The Ultimate GIT Cheat Sheet

Over 180 git commands on your fingertips!

Core Git Commands

These commands are fundamental for almost all Git operations:

1. `git init`: Initialize a new Git repository.
2. `git clone [url]`: Clone an existing repository.
3. `git add [file]`: Stage changes for commit.
4. `git commit -m "[commit message]"`: Commit changes to the repository.
5. `git status`: Check the status of changes.
6. `git branch`: Manage branches.
7. `git checkout [branch-name]`: Switch branches.
8. `git merge [branch]`: Merge branches.
9. `git remote add [variable name] [Remote Server Link]`: Connect to a remote repository.
10. `git push [variable name] [branch]`: Push changes to a remote repository.
11. `git pull`: Fetch and merge changes from a remote repository.
12. `git log`: Display commit logs.
13. `git fetch [remote]`: Download changes from a remote repository.
14. `git reset [file]`: Unstage changes.
15. `git diff`: Show changes between commits, commit and working tree, etc.
16. `git config --global user.name "[name]"`: Set a name for Git user.
17. `git config --global user.email "[email address]":` Set an email for Git user.

Git Branching Commands

These commands are used for creating, managing, and manipulating branches in Git, providing essential tools for efficient version control and collaborative development.

1. `git branch`: Lists all the local branches in the current repository.
2. `git branch [branch-name]`: Creates a new branch.
3. `git branch -d [branch-name]`: Deletes the specified branch. This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes.
4. `git branch -D [branch-name]`: Force deletes the specified branch, even if it has unmerged changes.

5. `git branch -m [old-branch-name] [new-branch-name]`: Renames a branch. If you are on the branch, you can omit the [old-branch-name].
6. `git branch -a`: Lists all branches, local and remote.
7. `git checkout [branch-name]`: Switches to the specified branch.
8. `git checkout -b [branch-name]`: Creates a new branch and then checks it out.
9. `git checkout -b [branch-name] [start-point]`: Creates a new branch starting from the specified commit, tag, or branch.
10. `git merge [branch-name]`: Merges the specified branch into the current branch.
11. `git merge --abort`: Aborts the merge process and tries to reconstruct the pre-merge state.
12. `git rebase [base]`: Rebase the current branch onto the [base] branch.
13. `git rebase --abort`: Aborts a rebase operation and returns to the original branch.
14. `git rebase --continue`: Continues the rebase process after resolving conflicts.
15. `git rebase -i [base]`: Performs an interactive rebase.
16. `git branch --set-upstream-to=[remote/branch]`: Sets the upstream branch for the current branch.
17. `git branch --unset-upstream`: Removes the upstream information for the current branch.
18. `git cherry-pick [commit-hash]`: Applies the changes introduced by the specified commit on the current branch.
19. `git branch -vv`: Lists all branches with more information including their upstream.
20. `git branch --no-merged`: Lists branches that have not been merged.
21. `git branch --merged`: Lists branches that have been merged.
22. `git checkout --track [remote/branch]`: Creates a new branch that tracks a remote branch.
23. `git checkout --orphan [branch-name]`: Creates a new orphan branch, which is a branch with no history.
24. `git branch -f [branch-name] [start-point]`: Force moves a branch to a new start point.
25. `git branch --show-current`: Displays the name of the current branch.
26. `git branch [branch-name] [start-point]`: Creates a new branch at the specified start point.

Git Stashing Commands

These commands are used to temporarily store and manage uncommitted changes in your working directory.

1. `git stash`: Temporarily stashes the changes you've made to your working directory. This allows you to save your current work state and revert to a clean working directory.

2. **`git stash pop`:** Applies the changes from the most recently stashed state and then removes it from the stash list. If there are conflicts, the stash is not removed unless you add --index.
3. **`git stash list`:** Lists all the stashed changesets. This command is useful for viewing all your stashed changes.
4. **`git stash apply`:** Applies the changes from the most recently stashed state (or specified stash) to your working directory but keeps it in the stash list for possible later reuse.
5. **`git stash drop [stash@{}]`:** Discards the most recent stash (or a specified stash if you include the index) from the list of stashed changes.
6. **`git stash clear`:** Removes all the stashed states. This is helpful when you want to clean up and remove all saved stashes.
7. **`git stash create [message]`:** Creates a stash with an optional message without removing the changes from the working directory.
8. **`git stash save [message]`:** Similar to git stash, but allows you to provide a message for easy identification later. This command is deprecated in favor of git stash push.
9. **`git stash push [-u | --include-untracked]`:** Creates a stash and includes untracked files if the -u or --include-untracked option is used.
10. **git stash branch [branch-name] [stash@{}]:** Creates and checks out a new branch starting from the commit at which the stash was originally created, applying the stashed changes to the new working directory. If no stash is specified, the latest one is used.
11. **`git stash show [stash@{}]`:** Displays the changes recorded in the stash as a diff between the stashed state and its original parent. Shows the latest stash if no stash is specified.
12. **`git stash apply --index`:** Applies the changes and reinstates the index (staging area), so staged changes are still staged.

Git Remote Commands

These commands used for handling remote repositories in Git. They are essential for collaborating and managing code across different locations and teams.

1. **`git remote`:** Lists all remote repositories.
2. **`git remote -v:`** Lists all remote repositories along with their URLs.
3. **`git remote add [name] [url]`:** Adds a new remote repository.
4. **`git remote remove [name]`:** Removes a remote repository.
5. **`git remote rename [old-name] [new-name]`:** Renames a remote repository.
6. **`git fetch [remote]`:** Downloads new data from a remote repository but doesn't integrate any of this new data into your working files.
7. **`git fetch --all`:** Fetches changes from all configured remotes.

8. `git pull [remote] [branch]`: Fetches changes from the remote repository and merges them into the current branch.
9. `git push [remote] [branch]`: Uploads all local branch commits to the remote repository.
10. `git push --all [remote]`: Pushes all branches to the remote repository.
11. `git push [remote] --delete [branch]`: Deletes a branch on the remote repository.
12. `git clone [url]`: Copies a remote repository into a new directory, automatically creating a local master branch.
13. `git clone --mirror [url]`: Creates a bare repository that is a mirror of the remote repository.
14. `git remote show [remote]`: Gives detailed information about a particular remote.
15. `git remote update`: Fetches the most up-to-date objects from the remote repository.
16. `git remote set-url [remote] [new-url]`: Changes an existing remote repository URL.
17. `git remote set-head [remote] [-a | -d | [branch]]`: Sets or deletes the default branch (HEAD) of the remote.
18. `git remote prune [remote]`: Deletes all stale remote-tracking branches under [remote].
19. `git remote set-branches [remote] [branch]`: Changes the branches tracked by the remote.
20. `git push --tags [remote]`: Pushes tags to the remote repository.
21. `git ls-remote [remote]`: Lists all references, such as branches and tags, available in a remote repository.
22. `git push [remote] [local-branch]:[remote-branch]`: Pushes a local branch to a specific branch on the remote repository.
23. `git push [remote] :[remote-branch]`: Deletes a remote branch (equivalent to git push [remote] --delete [branch]).

Git Configuration Commands

These commands are used for configuring Git settings. These allow for customization at global, system, and local levels to optimize user experience and workflow efficiency.

1. `git config --global [key] [value]`: Sets a configuration value at the global level, which is applicable across all repositories for the current user.
2. `git config --system [key] [value]`: Sets a configuration value at the system level, which is applicable to all users on the machine.
3. `git config --local [key] [value]`: Sets a configuration value at the local repository level.
4. `git config --global --unset [key]`: Unsets a global configuration value.

5. **`git config --system --unset [key]`:** Unsets a system-level configuration value.
6. **`git config --local --unset [key]`:** Unsets a local repository configuration value.
7. **`git config --list`:** Lists all the configuration settings.
8. **`git config --global --list`:** Lists global configuration settings.
9. **`git config --system --list`:** Lists system-level configuration settings.
10. **`git config --local --list`:** Lists local repository configuration settings.
11. **`git config [key]`:** Gets the value for a given key.
12. **`git config --global user.name "[name]"`:** Sets the global username.
13. **`git config --global user.email "[email]"`:** Sets the global email.
14. **`git config --global core.editor "[editor]"`:** Sets the text editor to be used with Git commands.
15. **`git config --global alias.[alias-name] [git-command]`:** Creates an alias for a Git command.
16. **`git config --global --edit`:** Opens the global configuration file in a text editor for manual editing.
17. **`git config --system --edit`:** Opens the system-level configuration file in a text editor for manual editing.
18. **`git config --local --edit`:** Opens the local repository configuration file in a text editor for manual editing.
19. **`git config --global color.ui auto`:** Enables automatic command line coloring for Git for easy reviewing.
20. **`git config --global core.autocrlf [true|false|input]`:** Sets the handling of line endings in Git.
21. **`git config --global push.default [matching|simple|upstream|current]`:** Sets the default mode for git push.
22. **`git config --global merge.conflictstyle [merge|diff3]`:** Sets the style of conflict markers in merge conflicts.
23. **`git config --global pull.rebase false`:** Configures the default behavior of git pull to merge.
24. **`git config --global rebase.autoStash true`:** Enables automatic stashing before rebase operations.
25. **`git config --global credential.helper cache`:** Enables temporary caching of credentials.

Git Plumbing Commands

Git plumbing commands are the low-level commands that provide the core functionality of Git. These commands are the foundational building blocks upon which the more user-friendly "porcelain" commands (like `git commit`, `git merge`, etc.) are built. Git's lower-level plumbing

commands, offering insights into the advanced and foundational operations that power Git's internal mechanisms.

1. **`git cat-file [type] [object]`:** Provides content or type and size information for repository objects.
2. **`git checkout-index [options] [--] [paths…]`:** Copies files from the index to the working tree.
3. **`git commit-tree [tree] [-p parent] [-m message]`:** Creates a new commit object from the provided tree object and log message and prints its SHA-1 hash.
4. **`git diff-tree [options] [<commit> | <tree>]`:** Compares the content and mode of trees found via two tree objects.
5. **`git for-each-ref [options] [--] [pattern]`:** Lists references in a local repository, optionally filtered by a pattern.
6. **`git hash-object [options] [--] <path>`:** Computes object ID and optionally creates a blob from a file.
7. **`git ls-files [options] [--] [path]`:** Shows information about files in the index and the working tree.
8. **`git ls-remote [options] [<repository> [<refs>...]]`:** Lists references in a remote repository.
9. **`git merge-tree [base-tree] [branch1-tree] [branch2-tree]`:** Runs a three-way tree merge.
10. **`git read-tree [options]<tree-ish>` :** Reads tree information into the index.
11. **`git rev-parse [options] [--] [args]`:** Parses revision specifications, command line options, and file names.
12. **`git show-branch [--topics] [commit]`:** Shows branches and their commits.
13. **`git show-ref [options] [--] [pattern]`:** Lists references in a local repository.
14. **`git symbolic-ref [options] name [ref]`:** Reads, modifies, or deletes symbolic refs.
15. **`git tag --list`:** Lists tags in a repository.
16. **`git update-ref [options] <refname> <newvalue> [<oldvalue>]`:** Updates the object name stored in a ref safely.
17. **`git mktree [--missing] [--batch] [--normalize]`:** Reads standard input and creates a tree object.
18. **`git pack-objects [--stdout | -o <file> ] [--all-progress]`:** Creates a packed archive of the objects in the repository.
19. **`git unpack-objects [--strict] [-n] [-q] [--recover] < <packfile>`:** Unpacks objects from a packed archive.
20. **`git verify-pack [-v] <pack>…` :** Validates the packed Git archive.
21. **`git write-tree [--missing-ok] [--prefix=/]`:** Creates a tree object from the current index.

Git Porcelain

Git porcelain commands are the higher-level commands that provide a user-friendly interface for the majority of Git operations. These commands abstract the complexities of Git's internal workings (handled by plumbing commands) into a more accessible and easy-to-use format.

1. **`git blame [file]`**: Shows what revision and author last modified each line of a file.
2. **`git bisect:`** Use binary search to find the commit that introduced a bug.
3. **`git grep`:** Prints lines matching a pattern.
4. **`git reset [commit]`:** Resets the current HEAD to the specified state.
5. **`git show [object]`:** Shows various types of objects.
6. **`git tag [tagname] [commit]`:** Creates, lists, deletes or verifies a tag object signed with GPG.
7. **`git rm [file]`:** Removes files from the working tree and from the index.
8. **`git mv [file] [new file]`:** Moves or renames a file, a directory, or a symlink.
9. **`git clone [url]`:** Clones a repository into a new directory.

Git Experimental

Git experimental commands are those that are either in the experimental stage of development or are less commonly used, often for very specific purposes. They might not be as stable or widely documented as the standard Git commands. While these commands are not as commonly used as the standard Git commands, they provide specialized functionalities that can be crucial for certain advanced workflows. It's important to use them with an understanding of their specific use cases and potential effects on your repository.

1. **`git annex`:** Manages large files without storing them in the Git repository.
2. **`git am`:** Applies a series of patches from a mailbox.
3. **`git cherry-pick --upstream`:** Experimental command for upstream cherry-picking.
4. **`git describe`:** Describes a commit using the most recent tag reachable from it.
5. **`git format-patch`:** Formats and exports a series of commits as emailed patches.
6. **`git fsck`:** Verifies the connectivity and validity of the objects in the database.
7. **`git gc`:** Cleans up unnecessary files and optimizes the local repository.
8. **`git help`:** Shows help for Git commands.
9. **`git log --merges`:** Lists commit logs that are merges.
10. **`git log --oneline`:** Displays the commit logs in a short, one-line format.
11. **`git log --pretty=`:** Formats the log output using a variety of user-specified formats.
12. **`git log --short-commit`:** Shows a shorter version of the commit ID.
13. **`git log --stat`:** Shows stats (file changes, insertions, deletions) for the commit log.

14. **`git log --topo-order`:** Orders commits in topological order.
15. **`git merge-ours`:** Merges using the 'ours' strategy.
16. **`git merge-recursive`:** Merges using the 'recursive' strategy.
17. **`git merge-subtree`:** Merges using the 'subtree' strategy.
18. **`git mergetool`:** Invokes a merge resolution tool.
19. **`git mktag`:** Creates a tag object.
20. **`git mv`:** Moves or renames a file, a directory, or a symlink.
21. **`git patch-id`:** Generates patch identifiers.
22. **`git p4`:** Synchronizes between Perforce and Git.
23. **`git prune`:** Prunes all unreachable objects from the object database.
24. **`git pull --rebase`:** Fetches from and rebases on top of another repository or a local branch.
25. **`git push --mirror`:** Pushes a mirror of the local repository.
26. **`git push --tags`:** Pushes tags to the remote repository.
27. **`git reflog`:** Manages reflog information.
28. **`git replace`:** Creates, lists, deletes, or verifies replace refs.
29. **`git reset --hard`:** Resets the current HEAD to the specified state, discarding all changes.
30. **`git reset --mixed`:** Resets the index but not the working tree.
31. **`git revert`:** Reverts existing commits.
32. **`git rm`:** Removes files from the working tree and the index.
33. **`git show-branch`:** Shows branches and their commits.
34. **`git show-ref`:** Lists references in a local repository.
35. **`git stash save`:** Saves your local modifications away and reverts the working directory to match the HEAD commit.
36. **`git subtree`:** Merges and splits subtrees from your project.
37. **`git tag --delete`:** Deletes a tag.
38. **`git tag --force`:** Replaces an existing tag.
39. **`git tag --sign`:** Creates a signed tag.
40. **`git tag -f`:** Forcefully creates or updates a tag.
41. **`git tag -l`:** Lists tags.
42. **`git tag --verify`:** Verifies a signed tag.
43. **`git unpack-file`:** Creates a new file with the content of a blob object.
44. **`git update-index`:** Registers file contents in the working tree to the index.
45. **`git verify-pack`:** Validates packed Git archives.
46. **`git worktree`:** Manages multiple working trees attached to the same repository.