

Git Logo by Jason Long is licensed under the

Creative Commons Attribution 3.0 Unported License.

Copied from git-scm.com/downloads/logos

**WHAT IS GIT**

Git is an open source version control system. Along with allowing sharing of a code base easily with others for them to use or contribute to, Git is the equivalent to running a manual backup on files every time an important change is made. This is desirable for a number of reasons, including the frighteningly common occurrence of needing to modify half a dozen files in order to test a change to the project’s code. One approach is to cowboy code, make the changes on the fly, cross the fingers and hope that nothing is then broken. A much wiser approach would be to make a copy of these files before modifying them in order to quickly revert changes if things go awry. But, what if there is an uncovered edge case and the code does not need to be reverted until a month later? It already is a huge workload to copy files by hand, but having to keep track of each version of each file in case problems arise months later seems like a gross misallocation of development time. Git was designed to make these situations manageable. Git allows developers to commit snapshots of files as they are changed, string them along in a history, and revert to any combination of the files at any point in that history with only few keystrokes.

**HOW IT IS USEFUL**

Git supports having multiple versions of the source files that can developed and tested independently of each other and merged together later. Often this merge is completely automated. Git can be useful in the situation where one developer on a team is working on a new feature and another working on a bug fix. Git can prevent disasters such as losing all of the new features from the past month when the hard drive meets its untimely end. With remote repositories hosted on a cloud, Git can prevent setbacks from all kinds of mishaps. The true power of Git comes from its facilitation for collaboration on projects of any scale. Git allows code changes to be easily shared with others for review, testing, instruction, or production. It allows large groups of people to be working on the code base at once and for their changes to be organized and coordinated so that their work results in more robust code. All of these benefits make Git one of the most crucial skills for a software developer to be familiar with in order to be a productive team member.

**COMMITS**

The base unit in Git is a commit. A commit is a snapshot of how one or more files differ from the last version in the file’s history. When a file is committed, Git performs several actions. First, it performs a diff of the file against the previous version of it. A diff is essentially subtracting the old version from the new, producing only the changes in between the versions. A hash of the diff is then generated using the SHA1 algorithm. This hash is used to name each commit. Commits contain a pointer to their ancestor commits. This means each commit knows its parent commit. The final part of a commit is the commit message. This is a message added by the author of the commit which is used to describe the changes it contains.

Viewing the Status

To see the current status of the repository perform a git status command. This will print the following to console:

On branch master

Your branch is up-to-date with ‘origin/master’.

Nothing to commit, working directory clean

The first line contains the current branch. The second compares the branch to its remote branch and displays whether the current branch is ahead, behind or up-to-date. The third line contains the status of the staging area. It will list files that have been changed or deleted since the last commit or new files not tracked by Git. In the above example, nothing has been changed. Below is an example of running Git status after changes have been made.

On branch master

Your branch is up-to-date with ‘origin/master’.

Changes not staged for commit:

modified: changed.txt

deleted: deleted.txt

Untracked files:

new.txt

This example shows how it displays when changes need to be staged for commit. These are listed as modified or deleted. Files without a previous commit history by Git are labelled untracked.

Adding a file

To add files to a commit they must first add be added to the staging area using

git add <path>. This will move files or directories to the staging area to be attached to the next commit. The example below illustrates the result of the command git add new.txt:

On branch master

Your branch is up-to-date with ‘origin/master’.

Changes to be committed:

new file: new.txt

Changes not staged for commit:

modified: changed.txt

deleted: deleted.txt

Removing a File

To remove a file from the staging area use git reset HEAD <file>. HEAD refers to the current checked out branch. This does not delete the file, but prevents it from being attached to the next commit.

Creating a Commit

To create a commit, use the git commit command. This will open a text editor where a message should be added describing what changes have been made. The commit will be saved when the text editor has been closed. Two shortcuts for committing are use of the a and m flags. git commit –a skips the staging steps and adds all of the changed files that Git is tracking to the staging area resulting in:

On branch master

Your branch is up-to-date with ‘origin/master’.

Changes to be committed:

modified: changed.txt

deleted: deleted.txt

Untracked files:

new.txt

git commit –m”<message>” uses the text in the message area as the commit message, bypassing the text editor. git commit –am”<message>” uses both flags in one command simultaneously, to quickly commit all tracked files with the included message.

**BRANCHES**

A history of commits in Git is called a branch, and the default branch is master. Branches can be checked out (selected for use and modification), created, merged or deleted.

Listing Branches

Use git branch to list all available local branches.

Selecting a Branch

Use git checkout <branch name> to select a branch to work on. When checking out a branch for modification, Git performs all the changes of the commits in the branch on the user’s file system. This causes the user’s files to mirror the files whenever the last commit on the branch was made.

Creating a Branch

To make a new branch and check it out, use git checkout –b <new branch name>. This will copy the current branch to a new name and allow changes to be made on the branches independently. It is a good habit to create a new branch for any new feature or bug fix before beginning work.

Deleting a Branch

Deleting a branch git branch –D <branch name>, also deletes the commits it contains, unless those commits have already been merged into another branch.

Merging a Branch

Merging is one of the most useful features of Git. Merging one branch into another creates a new commit belonging to the branch that is checked out. This commit contains the previous commit on the branch and all of the changes from the branch being merged. If a file has been changed in both branches, Git is most often able to merge them seamlessly. To merge a branch into the current branch, use git merge <other branch>.

Handling Merge Conflicts

Merge conflicts only occur if the same lines have been modified in both files. If this happens, Git will insert both changes side by side into the file, alert the user, and pause before the automatic commit. This allows the user to deconflict them. After manually merging the conflicted areas the user must finish the commit. If a conflict occurs the following will be printed to the console:

Automatic merge failed; fix conflicts and then commit the result.

A Git status lists the conflicted files. Simply open the conflicted files and look for the inserted text. Suppose there are two branches: hello and goodbye. Both contain the file message.txt which holds the text “hi” and “bye” in the respective branches. git checkout hello followed by git merge goodbye will result in a merge conflict because the same line has been changed in both files. Opening up the file message.txt after the merge fails will show:

<<<<<<< HEAD  
hi  
=======  
bye  
>>>>>>> goodbye

Git inserts conflict markers into the file to mark the start, middle and end of a conflict. The conflict starts with the marker <<<<<<< HEAD. This text is followed by the conflicted lines from the checked out branch ending with =======, which marks the middle. Then comes the text from the merged branch, and the >>>>>>> goodbye marker which signifies the end of the conflict. To resolve the conflict, change the conflicted files contents so that the desired functionality is achieved. Often this involves deleting the conflict markers, and leaving all of the changes from ether the HEAD or the merged branch. Alternatives are to leave both changes together if they do not conflict or to refactor them so that the changes can coexist. Once all of the conflicts have been reconciled, add and commit the results.

**REPOSITORIES**

Repositories or “repos” are the outer container of Git. They contain all the branches. One of the most powerful features of Git is the use of remote repositories, which are repos that exist on a system other than the users. Remote repositories allow users to store copies of their projects on the cloud either privately or publicly. Private repos allow a user or group of users to collaborate on a project privately. Public repos are open to everyone to view, use and contribute. They allow people from around the world to collaborate on projects or use other’s projects in their own work.

Cloning a Repo

Managing a project with even two collaborators can be a challenge. Git provides several ways to help manage a repository. The most basic is cloning, which copies the master branch of a repository onto the user’s local system. To clone a repo, use git clone <URL> with the url of the repo to be cloned.

Sharing and Receiving Commits

After cloning a repository to their system a user can make changes and then push their commits up to a remote branch so others can see them or to simply back them up. The command syntax is git push <remote name> <branch name>. If a remote branch contains different commits than the local one, the push will be rejected until the user has merged the commits that they are missing. This can be resolved through a pull using a pull command with the syntax: git pull <remote name> <branch name>.

A pull first compares the remotes most recent hash against its own. If the hashes differ, Git will gather all the missing commits from the remote repository and merge them into the current branch.

**HELP**

Git can be complicated. If you need assistance, simply append any command with the --help flag. This will display all the syntax and flags that can be used with a command and descriptions of each. It will also display an example of how to use the command. To exit the help screen, type q. Another great resource for help with Git is a Google search. Your problem has probably already been answered on the internet.