Undoing Commits & Changes

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes)

In this section, we will discuss the available 'undo' Git strategies and commands. It is first important to note that Git does not have a traditional 'undo' system like those found in a word processing application. It will be beneficial to refrain from mapping Git operations to any traditional 'undo' mental model. Additionally, Git has its own nomenclature for 'undo' operations that it is best to leverage in a discussion. This nomenclature includes terms like reset, revert, checkout, clean, and more.

A fun metaphor is to think of Git as a timeline management utility. Commits are snapshots of a point in time or points of interest along the timeline of a project's history. Additionally, multiple timelines can be managed through the use of branches. When 'undoing' in Git, you are usually moving back in time, or to another timeline where mistakes didn't happen.

This tutorial provides all of the necessary skills to work with previous revisions of a software project. First, it shows you how to explore old commits, then it explains the difference between reverting public commits in the project history vs. resetting unpublished changes on your local machine.

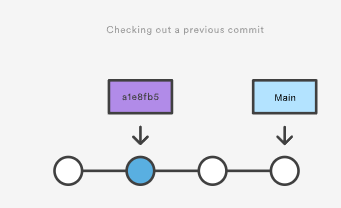
Finding what is lost: Reviewing old commits

The whole idea behind any version control system is to store “safe” copies of a project so that you never have to worry about irreparably breaking your code base. Once you’ve built up a project history of commits, you can review and revisit any commit in the history. One of the best utilities for reviewing the history of a Git repository is the git log command. In the example below, we use [git log](https://www.atlassian.com/git/tutorials/git-log) to get a list of the latest commits to a popular open-source graphics library.

git log --oneline  
e2f9a78fe Replaced FlyControls with OrbitControls  
d35ce0178 Editor: Shortcuts panel Safari support.  
9dbe8d0cf Editor: Sidebar.Controls to Sidebar.Settings.Shortcuts. Clean up.  
05c5288fc Merge pull request #12612 from TyLindberg/editor-controls-panel  
0d8b6e74b Merge pull request #12805 from harto/patch-1  
23b20c22e Merge pull request #12801 from gam0022/improve-raymarching-example-v2  
fe78029f1 Fix typo in documentation  
7ce43c448 Merge pull request #12794 from WestLangley/dev-x  
17452bb93 Merge pull request #12778 from OndrejSpanel/unitTestFixes  
b5c1b5c70 Merge pull request #12799 from dhritzkiv/patch-21  
1b48ff4d2 Updated builds.  
88adbcdf6 WebVRManager: Clean up.  
2720fbb08 Merge pull request #12803 from dmarcos/parentPoseObject  
9ed629301 Check parent of poseObject instead of camera  
219f3eb13 Update GLTFLoader.js  
15f13bb3c Update GLTFLoader.js  
6d9c22a3b Update uniforms only when onWindowResize  
881b25b58 Update ProjectionMatrix on change aspect

Each commit has a unique SHA-1 identifying hash. These IDs are used to travel through the committed timeline and revisit commits. By default, git log will only show commits for the currently selected branch. It is entirely possible that the commit you're looking for is on another branch. You can view all commits across all branches by executing git log --branches=\*. The command [git branch](https://www.atlassian.com/git/tutorials/using-branches) is used to view and visit other branches. Invoking the command, git branch -a will return a list of all known branch names. One of these branch names can then be logged using git log.

When you have found a commit reference to the point in history you want to visit, you can utilize the git checkout command to visit that commit. Git checkout is an easy way to “load” any of these saved snapshots onto your development machine. During the normal course of development, the HEAD usually points to main or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state, and it can be visualized as the following:



Checking out an old file does not move the HEAD pointer. It remains on the same branch and same commit, avoiding a 'detached head' state. You can then commit the old version of the file in a new snapshot as you would any other changes. So, in effect, this usage of git checkout on a file, serves as a way to revert back to an old version of an individual file. For more information on these two modes visit the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) page

Viewing an old revision

This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

git log --oneline

Let’s say your project history looks something like the following:

b7119f2 Continue doing crazy things  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import

You can use git checkout to view the “Make some import changes to hello.txt” commit as follows:

git checkout a1e8fb5

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. Nothing you do in here will be saved in your repository. To continue developing, you need to get back to the “current” state of your project:

git checkout main

This assumes that you're developing on the default main branch. Once you’re back in the main branch, you can use either [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert)or [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to undo any undesired changes.

Undoing a committed snapshot

There are technically several different strategies to 'undo' a commit. The following examples will assume we have a commit history that looks like:

git log --oneline  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import

We will focus on undoing the 872fa7e Try something crazy commit. Maybe things got a little too crazy.

How to undo a commit with git checkout

Using the git checkout command we can checkout the previous commit, a1e8fb5, putting the repository in a state before the crazy commit happened. Checking out a specific commit will put the repo in a "detached HEAD" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector. The garbage collector runs on a configured interval and permanently destroys orphaned commits. To prevent orphaned commits from being garbage collected, we need to ensure we are on a branch.

From the detached HEAD state, we can execute git checkout -b new\_branch\_without\_crazy\_commit. This will create a new branch named new\_branch\_without\_crazy\_commit and switch to that state. The repo is now on a new history timeline in which the 872fa7e commit no longer exists. At this point, we can continue work on this new branch in which the 872fa7e commit no longer exists and consider it 'undone'. Unfortunately, if you need the previous branch, maybe it was your main branch, this undo strategy is not appropriate. Let's look at some other 'undo' strategies. For more information and examples review our in-depth [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) discussion.

How to undo a public commit with git revert

Let's assume we are back to our original commit history example. The history that includes the 872fa7e commit. This time let's try a revert 'undo'. If we execute git revert HEAD, Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:

git log --oneline  
e2f9a78 Revert "Try something crazy"  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import

At this point, we have again technically 'undone' the 872fa7e commit. Although 872fa7e still exists in the history, the new e2f9a78 commit is an inverse of the changes in 872fa7e. Unlike our previous checkout strategy, we can continue using the same branch. This solution is a satisfactory undo. This is the ideal 'undo' method for working with public shared repositories. If you have requirements of keeping a curated and minimal Git history this strategy may not be satisfactory.

How to undo a commit with git reset

For this undo strategy we will continue with our working example. [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) is an extensive command with multiple uses and functions. If we invoke git reset --hard a1e8fb5 the commit history is reset to that specified commit. Examining the commit history with git log will now look like:

git log --oneline  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import

The log output shows the e2f9a78 and 872fa7e commits no longer exist in the commit history. At this point, we can continue working and creating new commits as if the 'crazy' commits never happened. This method of undoing changes has the cleanest effect on history. Doing a reset is great for local changes however it adds complications when working with a shared remote repository. If we have a shared remote repository that has the 872fa7e commit pushed to it, and we try to git push a branch where we have reset the history, Git will catch this and throw an error. Git will assume that the branch being pushed is not up to date because of it's missing commits. In these scenarios, git revert should be the preferred undo method.

Undoing the last commit

In the previous section, we discussed different strategies for undoing commits. These strategies are all applicable to the most recent commit as well. In some cases though, you might not need to remove or reset the last commit. Maybe it was just made prematurely. In this case you can amend the most recent commit. Once you have made more changes in the working directory and staged them for commit by using [git add](https://www.atlassian.com/git/tutorials/saving-changes), you can execute git commit --amend. This will have Git open the configured system editor and let you modify the last commit message. The new changes will be added to the amended commit.

Undoing uncommitted changes

Before changes are committed to the repository history, they live in the staging index and the working directory. You may need to undo changes within these two areas. The staging index and working directory are internal Git state management mechanisms. For more detailed information on how these two mechanisms operate, visit the [git reset](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting) page which explores them in depth.

The working directory

The working directory is generally in sync with the local file system. To undo changes in the working directory you can edit files like you normally would using your favorite editor. Git has a couple utilities that help manage the working directory. There is the [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) command which is a convenience utility for undoing changes to the working directory. Additionally, git reset can be invoked with the --mixed or --hard options and will apply a reset to the working directory.

The staging index

The [git add](https://www.atlassian.com/git/tutorials/saving-changes) command is used to add changes to the staging index. Git reset is primarily used to undo the staging index changes. A --mixed reset will move any pending changes from the staging index back into the working directory.

Undoing public changes

When working on a team with remote repositories, extra consideration needs to be made when undoing changes. Git reset should generally be considered a 'local' undo method. A reset should be used when undoing changes to a private branch. This safely isolates the removal of commits from other branches that may be in use by other developers. Problems arise when a reset is executed on a shared branch and that branch is then pushed remotely with git push. Git will block the push in this scenario complaining that the branch being pushed is out of date from the remote branch as it is missing commits.

The preferred method of undoing shared history is git revert. A revert is safer than a reset because it will not remove any commits from a shared history. A revert will retain the commits you want to undo and create a new commit that inverts the undesired commit. This method is safer for shared remote collaboration because a remote developer can then pull the branch and receive the new revert commit which undoes the undesired commit.

Summary

We covered many high-level strategies for undoing things in Git. It's important to remember that there is more than one way to 'undo' in a Git project. Most of the discussion on this page touched on deeper topics that are more thoroughly explained on pages specific to the relevant Git commands. The most commonly used 'undo' tools are [git checkout,](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset). Some key points to remember are:

* Once changes have been committed they are generally permanent
* Use git checkout to move around and review the commit history
* git revert is the best tool for undoing shared public changes
* git reset is best used for undoing local private changes

In addition to the primary undo commands, we took a look at other Git utilities: [git log](https://www.atlassian.com/git/tutorials/git-log) for finding lost commits [git clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) for undoing uncommitted changes [git add](https://www.atlassian.com/git/tutorials/saving-changes) for modifying the staging index.

Each of these commands has its own in-depth documentation. To learn more about a specific command mentioned here, visit the corresponding links.

Git Clean

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean)

In this section, we will focus on a detailed discussion of the git clean command. Git clean is to some extent an 'undo' command. Git clean can be considered complementary to other commands like [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) and [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout). Whereas these other commands operate on files previously added to the Git tracking index, the git clean command operates on untracked files. Untracked files are files that have been created within your repo's working directory but have not yet been added to the repository's tracking index using the [git add](https://www.atlassian.com/git/tutorials/saving-changes) command. To better demonstrate the difference between tracked and untracked files consider the following command line example

$ mkdir git\_clean\_test  
$ cd git\_clean\_test/  
$ git init .  
Initialized empty Git repository in /Users/kev/code/git\_clean\_test/.git/  
$ echo "tracked" > ./tracked\_file  
$ git add ./tracked\_file  
$ echo "untracked" > ./untracked\_file  
$ mkdir ./untracked\_dir && touch ./untracked\_dir/file  
$ git status  
On branch master  
  
Initial commit  
  
Changes to be committed: (use "git rm --cached <file>..." to unstage)  
  
new file: tracked\_file  
  
Untracked files: (use "git add <file>..." to include in what will be committed) untracked\_dir/ untracked\_file

The example creates a new Git repository in the git\_clean\_test directory. It then proceeds to create a tracked\_file which is added to the Git index, additionally, an untracked\_file is created, and an untracked\_dir. The example then invokes git status which displays output indicating Git's internal state of tracked and untracked changes. With the repository in this state, we can execute the git clean command to demonstrate its intended purpose.

$ git clean fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean

At this point, executing the default git clean command may produce a fatal error. The example above demonstrates what this may look like. By default, Git is globally configured to require that git clean be passed a "force" option to initiate. This is an important safety mechanism. When finally executed git clean is not undo-able. When fully executed, git clean will make a hard filesystem deletion, similar to executing the command line rm utility. Make sure you really want to delete the untracked files before you run it.

Common options and usage

Given the previous explanation of the default git clean behaviors and caveats, the following content demonstrates various git clean use cases and the accompanying command line options required for their operation.

-n

The -n option will perform a “dry run” of git clean. This will show you which files are going to be removed without actually removing them. It is a best practice to always first perform a dry run of git clean. We can demonstrate this option in the demo repo we created earlier.

$ git clean -n  
Would remove untracked\_file

The output tells us that untracked\_file will be removed when the git clean command is executed. Notice that the untracked\_dir is not reported in the output here. By default git clean will not operate recursively on directories. This is another safety mechanism to prevent accidental permanent deletion.

-f or --force

The force option initiates the actual deletion of untracked files from the current directory. Force is required unless the clean.requireForce configuration option is set to false. This will not remove untracked folders or files specified by .gitignore. Let us now execute a live git clean in our example repo.

$ git clean -f   
Removing untracked\_file

The command will output the files that are removed. You can see here that untracked\_file has been removed. Executing git status at this point or doing a ls will show that untracked\_file has been deleted and is nowhere to be found. By default git clean -f will operate on all the current directory untracked files. Additionally, a < path > value can be passed with the -f option that will remove a specific file.

git clean -f <path>  
-d include directories

The -d option tells git clean that you also want to remove any untracked directories, by default it will ignore directories. We can add the -d option to our previous examples:

$ git clean -dn  
Would remove untracked\_dir/  
$ git clean -df  
Removing untracked\_dir/

Here we have executed a 'dry run' using the -dn combination which outputs untracked\_dir is up for removal. Then we execute a forced clean, and receive output that untracked\_dir is removed.

-x force removal of ignored files

A common software release pattern is to have a build or distribution directory that is not committed to the repositories tracking index. The build directory will contain ephemeral build artifacts that are generated from the committed source code. This build directory is usually added to the repositories .gitignore file. It can be convenient to also clean this directory with other untracked files. The -x option tells git clean to also include any ignored files. As with previous git clean invocations, it is a best practice to execute a 'dry run' first, before the final deletion. The -x option will act on all ignored files, not just project build specific ones. This could be unintended things like ./.idea IDE configuration files.

git clean -xf

Like the -d option -x can be passed and composed with other options. This example demonstrates a combination with -f that will remove untracked files from the current directory as well as any files that Git usually ignores.

Interactive mode or git clean interactive

In addition to the ad-hoc command line execution we have demonstrated so far, git clean has an "interactive" mode that you can initiate by passing the -i option. Let us revisit the example repo from the introduction of this document. In that initial state, we will start an interactive clean session.

$ git clean -di  
Would remove the following items:  
  untracked\_dir/  untracked\_file  
\*\*\* Commands \*\*\*  
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit                 6: help  
What now>

We have initiated the interactive session with the -d option so it will also act upon our untracked\_dir. The interactive mode will display a What now> prompt that requests a command to apply to the untracked files. The commands themselves are fairly self explanatory. We'll take a brief look at each in a random order starting with command 6: help. Selecting command 6 will further explain the other commands:

What now> 6  
clean               - start cleaning  
filter by pattern   - exclude items from deletion  
select by numbers   - select items to be deleted by numbers  
ask each            - confirm each deletion (like "rm -i")  
quit                - stop cleaning  
help                - this screen  
?                   - help for prompt selection

5: quit

Is straight forward and will exit the interactive session.

1: clean

Will delete the indicated items. If we were to execute 1: clean at this point untracked\_dir/ untracked\_file would be removed

4: ask each

will iterate over each untracked file and display a Y/N prompt for a deletion. It looks like the following:

\*\*\* Commands \*\*\*  
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit                 6: help  
What now> 4  
Remove untracked\_dir/ [y/N]? N  
Remove untracked\_file [y/N]? N

2: filter by pattern

Will display an additional prompt that takes input used to filter the list of untracked files.

Would remove the following items:  
  untracked\_dir/  untracked\_file  
\*\*\* Commands \*\*\*  
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit                 6: help  
What now> 2  
  untracked\_dir/  untracked\_file  
Input ignore patterns>> \*\_file  
  untracked\_dir/

Here we input the \*\_file wildcard pattern which then restricts the untracked file list to just untracked\_dir.

3: select by numbers

Similar to command 2, command 3 works to refine the list of untracked file names. The interactive session will prompt for numbers that correspond to an untracked file name.

Would remove the following items:  
  untracked\_dir/  untracked\_file  
\*\*\* Commands \*\*\*  
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit                 6: help  
What now> 3  
    1: untracked\_dir/    2: untracked\_file  
Select items to delete>> 2  
    1: untracked\_dir/  \* 2: untracked\_file  
Select items to delete>>  
Would remove the following item:  
  untracked\_file  
\*\*\* Commands \*\*\*  
    1: clean                2: filter by pattern    3: select by numbers    4: ask each             5: quit                 6: help

Summary

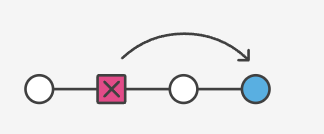
To recap, git clean is a convenience method for deleting untracked files in a repo's working directory. Untracked files are those that are in the repo's directory but have not yet been added to the repo's index with [git add](https://www.atlassian.com/git/tutorials/saving-changes). Overall the effect of git clean can be accomplished using [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) and the operating systems native deletion tools. Git clean can be used alongside [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to fully undo any additions and commits in a repository.

Git Revert

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert)

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

Reverting should be used when you want to apply the inverse of a commit from your project history. This can be useful, for example, if you’re tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do all of this for you.



How it works

The git revert command is used for undoing changes to a repository's commit history. Other 'undo' commands like, [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), move the HEAD and branch ref pointers to a specified commit. Git revert also takes a specified commit, however, git revert does not move ref pointers to this commit. A revert operation will take the specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.  
  
To demonstrate let’s create an example repo using the command line examples below:

$ mkdir git\_revert\_test  
$ cd git\_revert\_test/  
$ git init .  
Initialized empty Git repository in /git\_revert\_test/.git/  
$ touch demo\_file  
$ git add demo\_file  
$ git commit -am"initial commit"  
[main (root-commit) 299b15f] initial commit  
 1 file changed, 0 insertions(+), 0 deletions(-)  
 create mode 100644 demo\_file  
$ echo "initial content" >> demo\_file  
$ git commit -am"add new content to demo file"  
[main 3602d88] add new content to demo file  
n 1 file changed, 1 insertion(+)  
$ echo "prepended line content" >> demo\_file  
$ git commit -am"prepend content to demo file"  
[main 86bb32e] prepend content to demo file  
 1 file changed, 1 insertion(+)  
$ git log --oneline  
86bb32e prepend content to demo file  
3602d88 add new content to demo file  
299b15f initial commit

Here we have initialized a repo in a newly created directory named git\_revert\_test. We have made 3 commits to the repo in which we have added a file demo\_file and modified its content twice. At the end of the repo setup procedure, we invoke git log to display the commit history, showing a total of 3 commits. With the repo in this state, we are ready to initiate a git revert.

$ git revert HEAD [main b9cd081] Revert "prepend content to demo file" 1 file changed, 1 deletion(-)

Git revert expects a commit ref was passed in and will not execute without one. Here we have passed in the HEAD ref. This will revert the latest commit. This is the same behavior as if we reverted to commit 3602d8815dbfa78cd37cd4d189552764b5e96c58. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now examine the state of the repo using git log and see that there is a new commit added to the previous log:

$ git log --oneline 1061e79 Revert "prepend content to demo file" 86bb32e prepend content to demo file 3602d88 add new content to demo file 299b15f initial commit

Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 2nd and 4th commits represent the exact same code base and the 3rd commit is still in our history just in case we want to go back to it down the road.

Common options

-e  
--edit

This is a default option and doesn't need to be specified. This option will open the configured system editor and prompts you to edit the commit message prior to committing the revert

--no-edit

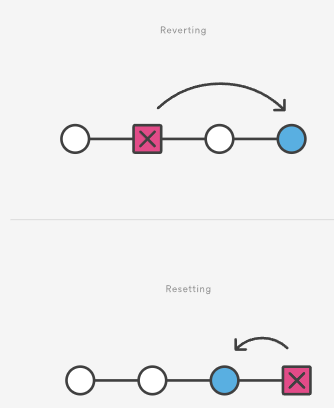
This is the inverse of the -e option. The revert will not open the editor.

-n  
--no-commit

Passing this option will prevent git revert from creating a new commit that inverses the target commit. Instead of creating the new commit this option will add the inverse changes to the Staging Index and Working Directory. These are the other trees Git uses to manage the state of the repository. For more info visit the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.

Resetting vs. reverting

It's important to understand that git revert undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a reset, not a revert.



Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository. For details about why altering shared history is dangerous, please see the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.  
  
Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backward from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution. For a more detailed discussion on the differences between git revert and other 'undo' commands see [Resetting, Checking Out and Reverting.](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting)

Summary

The git revert command is a forward-moving undo operation that offers a safe method of undoing changes. Instead of deleting or orphaning commits in the commit history, a revert will create a new commit that inverses the changes specified. Git revert is a safer alternative to git reset in regards to losing work. To demonstrate the effects of git revert we leveraged other commands that have more in-depth documentation on their individual pages: [git log](https://www.atlassian.com/git/tutorials/git-log), [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset).

Git Reset

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset)

The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments --soft, --mixed, --hard. The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

Git Reset & Three Trees of Git

To properly understand git reset usage, we must first understand Git's internal state management systems. Sometimes these mechanisms are called Git's "three trees". Trees may be a misnomer, as they are not strictly traditional tree data-structures. They are, however, node and pointer-based data structures that Git uses to track a timeline of edits. The best way to demonstrate these mechanisms is to create a changeset in a repository and follow it through the three trees.

To get started we will create a new repository with the commands below:

$ mkdir git\_reset\_test  
$ cd git\_reset\_test/  
$ git init .  
Initialized empty Git repository in /git\_reset\_test/.git/  
$ touch reset\_lifecycle\_file  
$ git add reset\_lifecycle\_file  
$ git commit -m"initial commit"  
[main (root-commit) d386d86] initial commit  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 reset\_lifecycle\_file

The above example code creates a new git repository with a single empty file, reset\_lifecycle\_file. At this point, the example repository has a single commit (d386d86) from adding reset\_lifecycle\_file.

The working directory

The first tree we will examine is "The Working Directory". This tree is in sync with the local filesystem and is representative of the immediate changes made to content in files and directories.

$ echo 'hello git reset' > reset\_lifecycle\_file  
 $ git status   
 On branch main  
 Changes not staged for commit:   
 (use "git add ..." to update what will be committed)   
 (use "git checkout -- ..." to discard changes in working directory)   
 modified: reset\_lifecycle\_file

In our demo repository, we modify and add some content to the reset\_lifecycle\_file. Invoking git status shows that Git is aware of the changes to the file. These changes are currently a part of the first tree, "The Working Directory". Git status can be used to show changes to the Working Directory. They will be displayed in the red with a 'modified' prefix.

Staging index

Next up is the 'Staging Index' tree. This tree is tracking Working Directory changes, that have been promoted with git add, to be stored in the next commit. This tree is a complex internal caching mechanism. Git generally tries to hide the implementation details of the Staging Index from the user.

To accurately view the state of the Staging Index we must utilize a lesser known Git command git ls-files. The git ls-files command is essentially a debug utility for inspecting the state of the Staging Index tree.

git ls-files -s  
100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0   reset\_lifecycle\_file

Here we have executed git ls-files with the -s or --stage option. Without the -s option the git ls-files output is simply a list of file names and paths that are currently part of the index. The -s option displays additional metadata for the files in the Staging Index. This metadata is the staged contents' mode bits, object name, and stage number. Here we are interested in the object name, the second value (d7d77c1b04b5edd5acfc85de0b592449e5303770). This is a standard Git object SHA-1 hash. It is a hash of the content of the files. The Commit History stores its own object SHA's for identifying pointers to commits and refs and the Staging Index has its own object SHA's for tracking versions of files in the index.

Next, we will promote the modified reset\_lifecycle\_file into the Staging Index.

$ git add reset\_lifecycle\_file

$ git status

On branch main Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

Here we have invoked git add reset\_lifecycle\_file which adds the file to the Staging Index. Invoking git status now shows reset\_lifecycle\_file in green under "Changes to be committed". It is important to note that git status is not a true representation of the Staging Index. The git status command output displays changes between the Commit History and the Staging Index. Let us examine the Staging Index content at this point.

$ git ls-files -s 100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

We can see that the object SHA for reset\_lifecycle\_file has been updated from e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 to d7d77c1b04b5edd5acfc85de0b592449e5303770.

Commit history

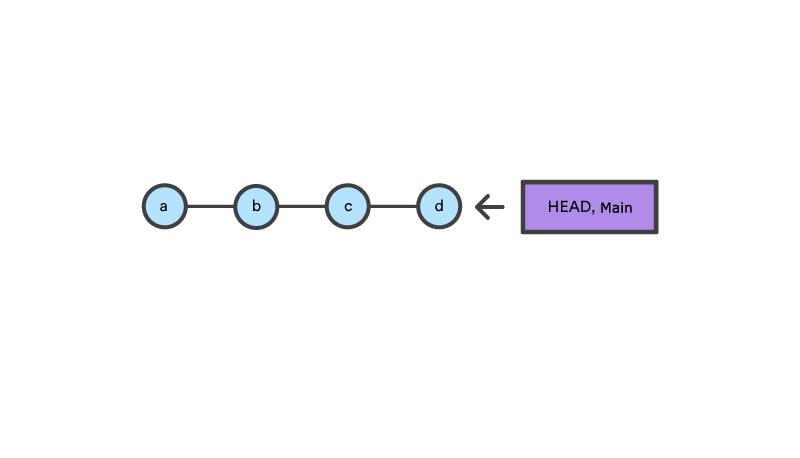
The final tree is the Commit History. The git commit command adds changes to a permanent snapshot that lives in the Commit History. This snapshot also includes the state of the Staging Index at the time of commit.

$ git commit -am"update content of reset\_lifecycle\_file"  
[main dc67808] update content of reset\_lifecycle\_file  
1 file changed, 1 insertion(+)  
$ git status  
On branch main  
nothing to commit, working tree clean

Here we have created a new commit with a message of "update content of resetlifecyclefile". The changeset has been added to the Commit History. Invoking git status at this point shows that there are no pending changes to any of the trees. Executing git log will display the Commit History. Now that we have followed this changeset through the three trees we can begin to utilize git reset.

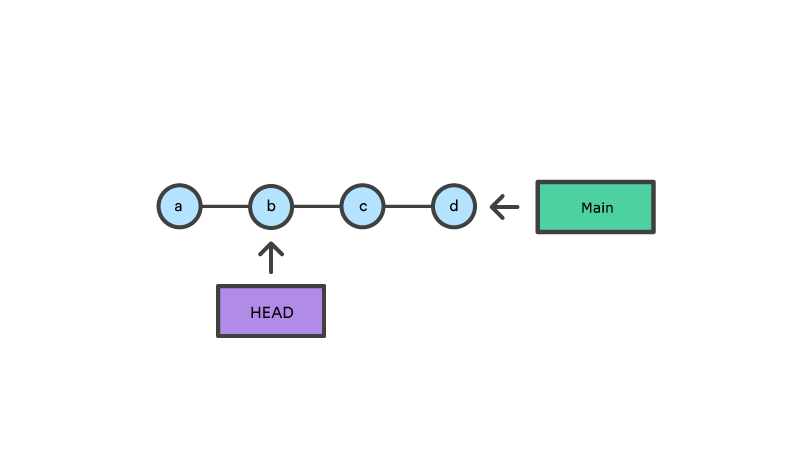
How it works

At a surface level, git reset is similar in behavior to git checkout. Where git checkout solely operates on the HEAD ref pointer, git reset will move the HEAD ref pointer and the current branch ref pointer. To better demonstrate this behavior consider the following example:



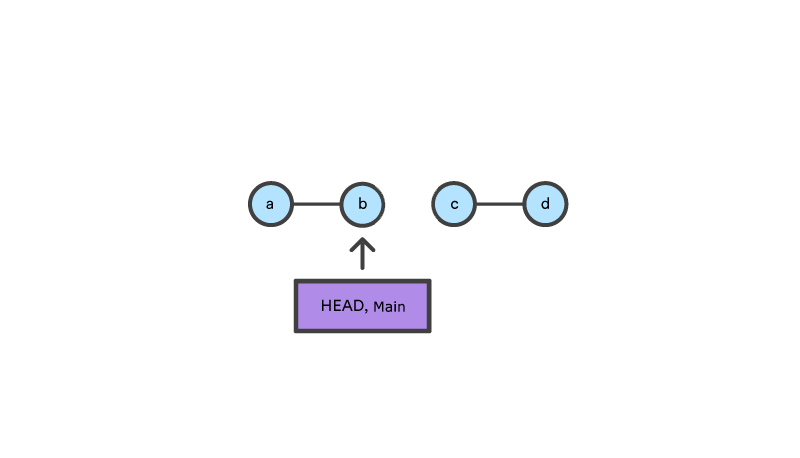
This example demonstrates a sequence of commits on the main branch. The HEAD ref and main branch ref currently point to commit d. Now let us execute and compare, both git checkout b and git reset b.

git checkout b



With git checkout, the main ref is still pointing to d. The HEAD ref has been moved, and now points at commit b. The repo is now in a 'detached HEAD' state.

git reset b

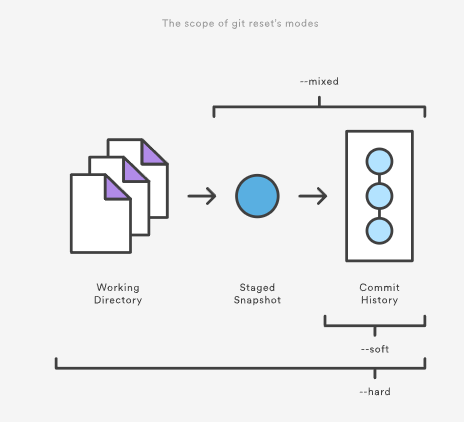


Comparatively, git reset, moves both the HEAD and branch refs to the specified commit.

In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.

Main Options

The default invocation of git reset has implicit arguments of --mixed and HEAD. This means executing git reset is equivalent to executing git reset --mixed HEAD. In this form HEAD is the specified commit. Instead of HEAD any Git SHA-1 commit hash can be used.



--hard

This is the most direct, DANGEROUS, and frequently used option. When passed --hard The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

To demonstrate this, let's continue with the three tree example repo we established earlier. First let's make some modifications to the repo. Execute the following commands in the example repo:

$ echo 'new file content' > new\_file  
$ git add new\_file  
$ echo 'changed content' >> reset\_lifecycle\_file

These commands have created a new file named new\_file and added it to the repo. Additionally, the content of reset\_lifecycle\_file will be modified. With these changes in place let us now examine the state of the repo using git status.

$ git status  
On branch main  
Changes to be committed:  
   (use "git reset HEAD ..." to unstage)  
  
new file: new\_file  
  
Changes not staged for commit:  
   (use "git add ..." to update what will be committed)  
   (use "git checkout -- ..." to discard changes in working directory)  
  
modified: reset\_lifecycle\_file

We can see that there are now pending changes to the repo. The Staging Index tree has a pending change for the addition of new\_file and the Working Directory has a pending change for the modifications to reset\_lifecycle\_file.

Before moving forward let us also examine the state of the Staging Index:

$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

We can see that new\_file has been added to the index. We have made updates to reset\_lifecycle\_file but the Staging Index SHA (d7d77c1b04b5edd5acfc85de0b592449e5303770) remains the same. This is expected behavior because have not used git add to promote these changes to the Staging Index. These changes exist in the Working Directory.

Let us now execute a git reset --hard and examine the new state of the repository.

$ git reset --hard  
HEAD is now at dc67808 update content of reset\_lifecycle\_file  
$ git status  
On branch main  
nothing to commit, working tree clean  
$ git ls-files -s  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "hard reset" using the --hard option. Git displays output indicating that HEAD is pointing to the latest commit dc67808. Next, we check the state of the repo with git status. Git indicates there are no pending changes. We also examine the state of the Staging Index and see that it has been reset to a point before new\_file was added. Our modifications to reset\_lifecycle\_file and the addition of new\_file have been destroyed. This data loss cannot be undone, this is critical to take note of.

--mixed

This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.

$ echo 'new file content' > new\_file  
$ git add new\_file  
$ echo 'append content' >> reset\_lifecycle\_file  
$ git add reset\_lifecycle\_file  
$ git status  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
new file: new\_file  
modified: reset\_lifecycle\_file  
  
  
$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file  
100644 7ab362db063f9e9426901092c00a3394b4bec53d 0 reset\_lifecycle\_file

In the example above we have made some modifications to the repository. Again, we have added a new\_file and modified the contents of reset\_lifecycle\_file. These changes are then applied to the Staging Index with git add. With the repo in this state, we will now execute the reset.

$ git reset --mixed  
$ git status  
On branch main  
Changes not staged for commit:  
    (use "git add ..." to update what will be committed)  
    (use "git checkout -- ..." to discard changes in working directory)  
  
modified: reset\_lifecycle\_file  
  
Untracked files:  
    (use "git add ..." to include in what will be committed)  
  
new\_file  
  
  
no changes added to commit (use "git add" and/or "git commit -a")  
$ git ls-files -s  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "mixed reset". To reiterate, --mixed is the default mode and the same effect as executing git reset. Examining the output from git status and git ls-files, shows that the Staging Index has been reset to a state where reset\_lifecycle\_file is the only file in the index. The object SHA for reset\_lifecycle\_file has been reset to the previous version.

The important things to take note of here is that git status shows us that there are modifications to reset\_lifecycle\_file and there is an untracked file: new\_file. This is the explicit --mixed behavior. The Staging Index has been reset and the pending changes have been moved into the Working Directory. Compare this to the --hard reset case where the Staging Index was reset and the Working Directory was reset as well, losing these updates.

--soft

When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched. This behavior can be hard to clearly demonstrate. Let's continue with our demo repo and prepare it for a soft reset.

$ git add reset\_lifecycle\_file

$ git ls-files -s

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

$ git status

On branch main

Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

Untracked files:

(use "git add ..." to include in what will be committed)

new\_file

Here we have again used git add to promote the modified reset\_lifecycle\_file into the Staging Index. We confirm that the index has been updated with the git ls-files output. The output from git status now displays the "Changes to be committed" in green. The new\_file from our previous examples is floating around in the Working Directory as an untracked file. Lets quickly execute rm new\_file to delete the file as we will not need it for the upcoming examples.

With the repository in this state we now execute a soft reset.

$ git reset --soft  
$ git status  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
modified: reset\_lifecycle\_file  
$ git ls-files -s  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

We have executed a 'soft reset'. Examining the repo state with git status and git ls-files shows that nothing has changed. This is expected behavior. A soft reset will only reset the Commit History. By default, git reset is invoked with HEAD as the target commit. Since our Commit History was already sitting on HEAD and we implicitly reset to HEAD nothing really happened.

To better understand and utilize --soft we need a target commit that is not HEAD. We have reset\_lifecycle\_file waiting in the Staging Index. Let's create a new commit.

$ git commit -m"prepend content to reset\_lifecycle\_file"

At this point, our repo should have three commits. We will be going back in time to the first commit. To do this we will need the first commit's ID. This can be found by viewing output from git log.

$ git log  
commit 62e793f6941c7e0d4ad9a1345a175fe8f45cb9df  
Author: bitbucket   
Date: Fri Dec 1 15:03:07 2017 -0800  
prepend content to reset\_lifecycle\_file  
  
commit dc67808a6da9f0dec51ed16d3d8823f28e1a72a  
Author: bitbucket   
Date: Fri Dec 1 10:21:57 2017 -0800  
  
update content of reset\_lifecycle\_file  
  
commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4  
  
Author: bitbucket   
Date: Thu Nov 30 16:50:39 2017 -0800  
  
initial commit

Keep in mind that Commit History ID's will be unique to each system. This means the commit ID's in this example will be different from what you see on your personal machine. The commit ID we are interested in for this example is 780411da3b47117270c0e3a8d5dcfd11d28d04a4. This is the ID that corresponds to the "initial commit". Once we have located this ID we will use it as the target for our soft reset.

Before we travel back in time lets first check the current state of the repo.

$ git status && git ls-files -s  
On branch main  
nothing to commit, working tree clean  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

Here we execute a combo command of git status and git ls-files -s this shows us there are pending changes to the repo and reset\_lifecycle\_file in the Staging Index is at a version of 67cc52710639e5da6b515416fd779d0741e3762e. With this in mind lets execute a soft reset back to our first commit.

$git reset --soft 780411da3b47117270c0e3a8d5dcfd11d28d04a4  
$ git status && git ls-files -s  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
modified: reset\_lifecycle\_file  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

The code above executes a "soft reset" and also invokes the git status and git ls-files combo command, which outputs the state of the repository. We can examine the repo state output and note some interesting observations. First, git status indicates there are modifications to reset\_lifecycle\_file and highlights them indicating they are changes staged for the next commit. Second, the git ls-files input indicates that the Staging Index has not changed and retains the SHA 67cc52710639e5da6b515416fd779d0741e3762e we had earlier.

To further clarify what has happened in this reset let us examine the git log:

$ git log commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4 Author: bitbucket Date: Thu Nov 30 16:50:39 2017 -0800 initial commit

The log output now shows that there is a single commit in the Commit History. This helps to clearly illustrate what --soft has done. As with all git reset invocations, the first action reset takes is to reset the commit tree. Our previous examples with --hard and --mixed have both been against the HEAD and have not moved the Commit Tree back in time. During a soft reset, this is all that happens.

This may then be confusing as to why git status indicates there are modified files. --soft does not touch the Staging Index, so the updates to our Staging Index followed us back in time through the commit history. This can be confirmed by the output of git ls-files -s showing that the SHA for reset\_lifecycle\_file is unchanged. As a reminder, git status does not show the state of 'the three trees', it essentially shows a diff between them. In this case, it is displaying that the Staging Index is ahead of the changes in the Commit History as if we have already staged them.

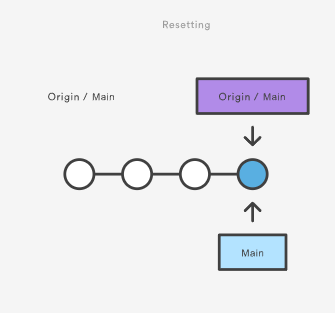
Resetting vs Reverting

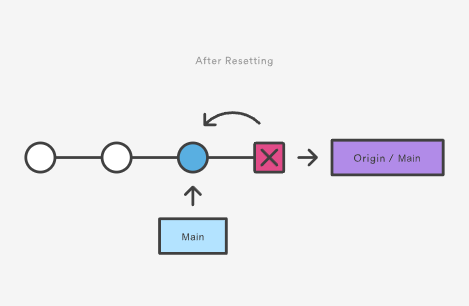
If [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) is a “safe” way to undo changes, you can think of git reset as the dangerous method. There is a real risk of losing work with git reset. Git reset will never delete a commit, however, commits can become 'orphaned' which means there is no direct path from a ref to access them. These orphaned commits can usually be found and restored using [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog). Git will permanently delete any orphaned commits after it runs the internal garbage collector. By default, Git is configured to run the garbage collector every 30 days. Commit History is one of the 'three git trees' the other two, Staging Index and Working Directory are not as permanent as Commits. Care must be taken when using this tool, as it’s one of the only Git commands that have the potential to lose your work.  
  
Whereas reverting is designed to safely undo a public commit, git reset is designed to undo local changes to the Staging Index and Working Directory. Because of their distinct goals, the two commands are implemented differently: resetting completely removes a changeset, whereas reverting maintains the original changeset and uses a new commit to apply the undo.

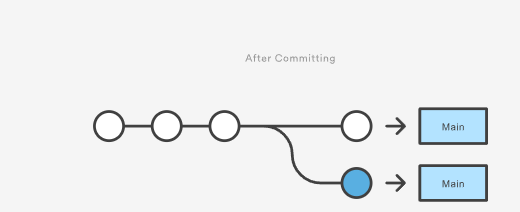
Don't Reset Public History

You should never use git reset when any snapshots after have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration. When they try to sync up with your repository, it will look like a chunk of the project history abruptly disappeared. The sequence below demonstrates what happens when you try to reset a public commit. The origin/main branch is the central repository’s version of your local main branch.







As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/main, and the merge commit required to synchronize your repositories is likely to confuse and frustrate your team.  
  
The point is, make sure that you’re using git reset on a local experiment that went wrong—not on published changes. If you need to fix a public commit, the git revert command was designed specifically for this purpose.

Examples

git reset ＜file＞

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

git reset

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

git reset --hard

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. Put another way: this obliterates all uncommitted changes, so make sure you really want to throw away your local developments before using it.

git reset

Move the current branch tip backward to commit, reset the staging area to match, but leave the working directory alone. All changes made since  will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

git reset --hard

Move the current branch tip backward to   and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after, as well.

Unstaging a file

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you’ve already added to the repository.

# Edit both hello.py and main.py  
  
# Stage everything in the current directory  
git add .  
  
# Realize that the changes in hello.py and main.py  
# should be committed in different snapshots  
  
# Unstage main.py  
git reset main.py  
  
# Commit only hello.py  
git commit -m "Make some changes to hello.py"  
  
# Commit main.py in a separate snapshot  
git add main.py  
git commit -m "Edit main.py"

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren’t related to the next commit.

Removing Local Commits

The next example shows a more advanced use case. It demonstrates what happens when you’ve been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

# Create a new file called `foo.py` and add some code to it  
  
# Commit it to the project history  
git add foo.py  
git commit -m "Start developing a crazy feature"  
  
# Edit `foo.py` again and change some other tracked files, too  
  
# Commit another snapshot  
git commit -a -m "Continue my crazy feature"  
  
# Decide to scrap the feature and remove the associated commits  
git reset --hard HEAD~2

The git reset HEAD~2 command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on unpublished commits. Never perform the above operation if you’ve already pushed your commits to a shared repository.

Summary

To review, git reset is a powerful command that is used to undo local changes to the state of a Git repo. Git reset operates on "The Three Trees of Git". These trees are the Commit History (HEAD), the Staging Index, and the Working Directory. There are three command line options that correspond to the three trees. The options --soft, --mixed, and --hard can be passed to git reset.  
  
In this article we leveraged several other Git commands to help demonstrate the reset processes. Learn more about those commands on their individual pages at: [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), [git log](https://www.atlassian.com/git/tutorials/git-log), [git add](https://www.atlassian.com/git/tutorials/saving-changes), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert).

Git RM

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

 A common question when getting started with Git is "How do I tell Git not to track a file (or files) any more?" The git rm command is used to remove files from a Git repository. It can be thought of as the inverse of the [git add](https://www.atlassian.com/git/tutorials/saving-changes) command.

Git rm Overview

The git rm command can be used to remove individual files or a collection of files. The primary function of git rm is to remove tracked files from the Git index. Additionally, git rm can be used to remove files from both the staging index and the working directory. There is no option to remove a file from only the working directory. The files being operated on must be identical to the files in the current HEAD. If there is a discrepancy between the HEAD version of a file and the staging index or working tree version, Git will block the removal. This block is a safety mechanism to prevent removal of in-progress changes.

Note that git rm does not remove branches. Learn more about [using git branches](https://www.atlassian.com/git/tutorials/using-branches)

Usage

<file>…​

Specifies the target files to remove. The option value can be an individual file, a space delimited list of files file1 file2 file3, or a wildcard file glob (~./directory/\*).

-f  
--force

The -foption is used to override the safety check that Git makes to ensure that the files in HEAD match the current content in the staging index and working directory.

-n  
--dry-run

The "dry run" option is a safeguard that will execute the git rm command but not actually delete the files. Instead it will output which files it would have removed.

-r

The -r option is shorthand for 'recursive'. When operating in recursive mode git rm will remove a target directory and all the contents of that directory.

--

The separator option is used to explicitly distinguish between a list of file names and the arguments being passed to git rm. This is useful if some of the file names have syntax that might be mistaken for other options.

--cached

The cached option specifies that the removal should happen only on the staging index. Working directory files will be left alone.

--ignore-unmatch

This causes the command to exit with a 0 sigterm status even if no files matched. This is a Unix level status code. The code 0 indicates a successful invocation of the command. The --ignore-unmatch option can be helpful when using git rm as part of a greater shell script that needs to fail gracefully.

-q  
--quiet

The quiet option hides the output of the git rm command. The command normally outputs one line for each file removed.

How to undo git rm

Executing git rm is not a permanent update. The command will update the staging index and the working directory. These changes will not be persisted until a new commit is created and the changes are added to the commit history. This means that the changes here can be "undone" using common Git commands.

git reset HEAD

A reset will revert the current staging index and working directory back to the HEAD commit. This will undo a git rm.

git checkout .

A checkout will have the same effect and restore the latest version of a file from HEAD.

In the event that git rm was executed and a new commit was created which persist the removal, git reflog can be used to find a ref that is before the git rm execution. Learn more about [using git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

Discussion

The <file> argument given to the command can be exact paths, wildcard file glob patterns, or exact directory names. The command removes only paths currently commited to the Git repository.

Wildcard file globbing matches across directories. It is important to be cautious when using wildcard globs. Consider the examples: directory/\* and directory\*. The first example will remove all sub files of directory/ whereas the second example will remove all sibling directories like directory1directory2directory\_whatever which may be an unexpected result.

The scope of git rm

The git rm command operates on the current branch only. The removal event is only applied to the working directory and staging index trees. The file removal is not persisted to the repository history until a new commit is created.

Why use git rm instead of rm

A Git repository will recognize when a regular shell rm command has been executed on a file it is tracking. It will update the working directory to reflect the removal. It will not update the staging index with the removal. An additional git add command will have to be executed on the removed file paths to add the changes to the staging index. The git rm command acts a shortcut in that it will update the working directory and the staging index with the removal.

Examples

git rm Documentation/\\*.txt

This example uses a wildcard file glob to remove all \*.txt files that are children of the Documentation directory and any of its subdirectories.

Note that the asterisk \* is escaped with slashes in this example; this is a guard that prevents the shell from expanding the wildcard. The wildcard then expands the pathnames of files and subdirectories under the Documentation/ directory.

git rm -f git-\*.sh

This example uses the force option and targets all wildcard git-\*.sh files. The force option explicitly removes the target files from both the working directory and staging index.

How to remove files no longer in the filesystem

As stated above in "Why use git rm instead of rm" , git rm is actually a convenience command that combines the standard shell rm and git add to remove a file from the working directory and promote that removal to the staging index. A repository can get into a cumbersome state in the event that several files have been removed using only the standard shell rm command.

If intentions are to record all the explicitly removed files as part of the next commit, git commit -a will add all the removal events to the staging index in preparation of the next commit.

If however, intentions are to persistently remove the files that were removed with the shell rm, use the following command:

git diff --name-only --diff-filter=D -z | xargs -0 git rm --cached

This command will generate a list of the removed files from the working directory and pipe that list to git rm --cached which will update the staging index.

Git rm summary

git rm is a command that operates on two of the primary Git [internal state management trees](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset): the working directory, and staging index. git rm is used to remove a file from a Git repository. It is a convenience method that combines the effect of the default shell rm command with git add. This means that it will first remove a target from the filesystem and then add that removal event to the staging index. The command is one of many that can be used for [undoing changes in Git.](https://www.atlassian.com/git/tutorials/undoing-changes)