# Git stash

# Undoing Commits & Changes

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

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old commits, then it explains the difference between reverting public commits in the project history vs. resetting unpublished changes on your local machine.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

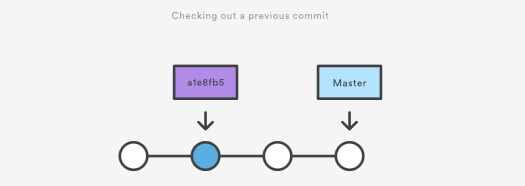
15f13bb3c Update GLTFLoader.js

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 <branch\_name>.

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 master 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 master

This assumes that you're developing on the default master branch. Once you’re back in the master 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 master 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](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

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

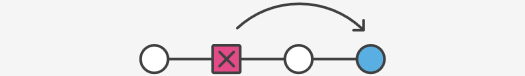
To recap, git clean is a convenience method for deleting untracked files in a repo's working directory. Untracked files are those that are in the repo's directory but have not yet been added to the repo's index with [git add](https://www.atlassian.com/git/tutorials/saving-changes). Overall the effect 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 clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [git rm](https://www.atlassian.com/git/tutorials/undoing-changes/git-rm)

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"

[master (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"

[master 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"

[master 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

[master 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.

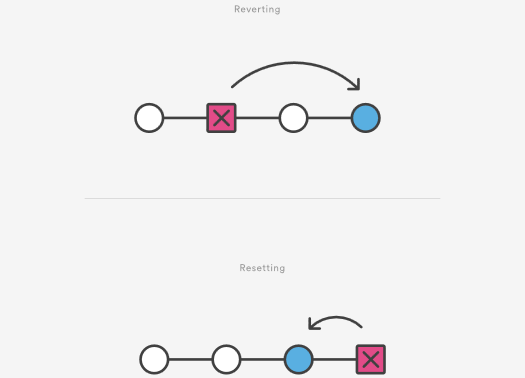
-n

--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 state 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 clean](https://www.atlassian.com/git/tutorials/undoing-changes/git-clean) [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) [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"

[master (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 master 

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 master 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"

[master dc67808] update content of reset\_lifecycle\_file

1 file changed, 1 insertion(+)

$ git status

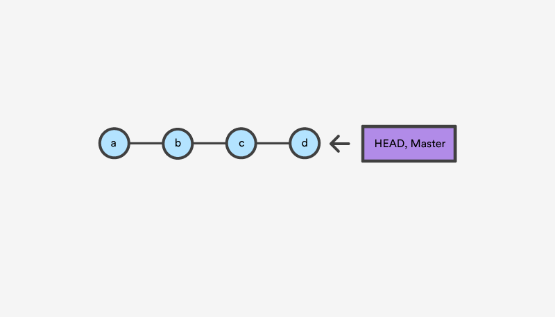
On branch master

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.

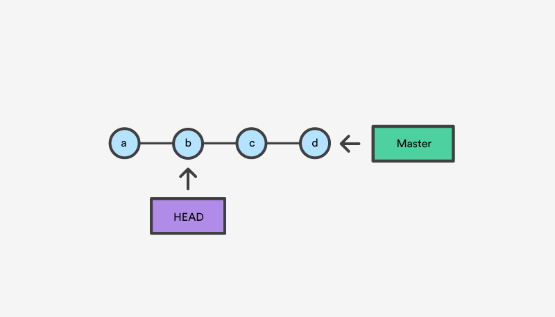
## How it works

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:



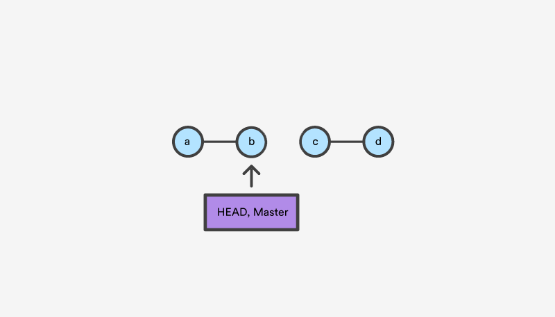
This example demonstrates a sequence of commits on the master branch. The HEAD ref and master branch ref currently point to commit d. Now let us execute and compare, both git checkout b and git reset b.

### git checkout b



With git checkout, the master 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

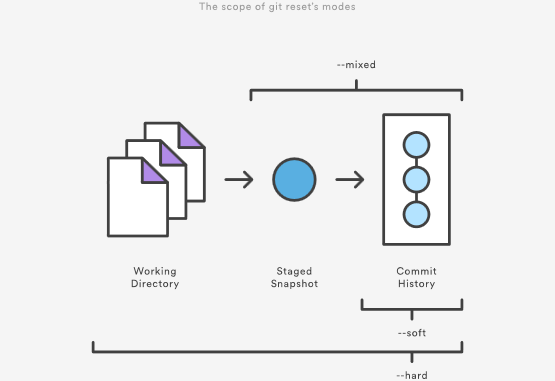


Comparatively, git reset, moves both the HEAD and branch refs to the specified commit.

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 master

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 master

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 master

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 master

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 master

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 master

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 master

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 master

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.

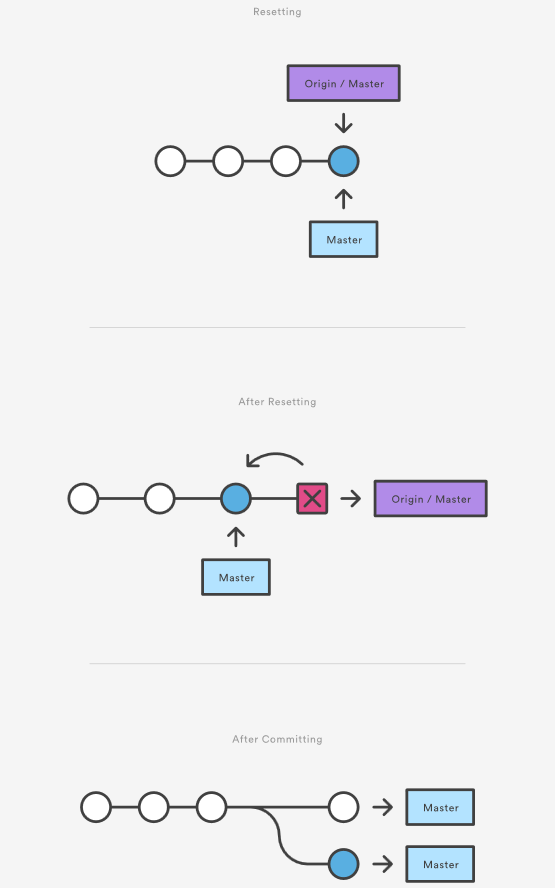
## Resetting vs Reverting

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 <commit> when any snapshots after <commit> 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/master branch is the central repository’s version of your local master branch.



As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/master, 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 <commit> 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 <commit>

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

git reset --hard <commit>

Move the current branch tip backward to <commit>  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).

# Rewriting history

###### Git commit --amend and other methods of rewriting history

## Intro

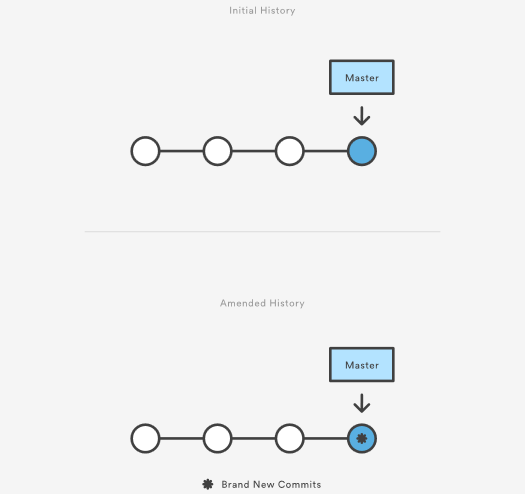
This tutorial will cover various methods of rewriting and altering Git history. Git uses a few different methods to record changes. We will discuss the strengths and weaknesses of the different methods and give examples of how to work with them. This tutorial discusses some of the most common reasons for overwriting committed snapshots and shows you how to avoid the pitfalls of doing so.

Git's main job is to make sure you never lose a committed change. But it's also designed to give you total control over your development workflow. This includes letting you define exactly what your project history looks like; however, it also creates the potential of losing commits. Git provides its history-rewriting commands under the disclaimer that using them may result in lost content.

Git has several mechanisms for storing history and saving changes. These mechanisms include: Commit --amend, git rebase and git reflog. These options give you powerful work flow customization options. By the end of this tutorial, you'll be familiar with commands that will let you restructure your Git commits, and be able to avoid pitfalls that are commonly encountered when rewriting history.

## Changing the Last Commit: git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram below. There are a few common scenarios for using git commit --amend. We'll cover usage examples in the following sections.



### Change most recent Git commit message

git commit --amend

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.

Premature commits happen all the time in the course of your everyday development. It’s easy to forget to stage a file or to format your commit message the wrong way. The --amend flag is a convenient way to fix these minor mistakes.

git commit --amend -m "an updated commit message"

Adding the -m option allows you to pass in a new message from the command line without being prompted to open an editor.

### Changing committed files

The following example demonstrates a common scenario in Git-based development. Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --amend flag:

# Edit hello.py and main.py git add hello.py git commit

# Realize you forgot to add the changes from main.py git add main.py

git commit --amend --no-edit

The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. The resulting commit will replace the incomplete one, and it will look like we committed the changes to hello.py and main.py in a single snapshot.

### Don’t amend public commits

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

### Recap

To review, git commit --amend lets you take the most recent commit and add new staged changes to it. You can add or remove changes from the Git staging area to apply with a --amend commit. If there are no changes staged, a --amend will still prompt you to modify the last commit message log. Be cautious when using --amend on commits shared with other team members. Amending a commit that is shared with another user will potentially require confusing and lengthy merge conflict resolutions.

## Changing older or multiple commits

To modify older or multiple commits, you can use git rebase to combine a sequence of commits into a new base commit. In standard mode, git rebase allows you to literally rewrite history — automatically applying commits in your current working branch to the passed branch head. Since your new commits will be replacing the old, it's important to not use git rebase on commits that have been pushed public, or it will appear that your project history disappeared.

In these or similar instances where it's important to preserve a clean project history, adding the -i option to git rebase allows you to run rebase interactive. This gives you the opportunity to alter individual commits in the process, rather than moving all commits. You can learn more about interactive rebasing and additional rebase commands on the [git rebase page](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

#### Changing committed files

During a rebase, the edit or e command will pause the rebase playback on that commit and allow you to make additional changes with git commit --amend Git will interrupt the playback and present a message:

Stopped at 5d025d1... formatting

You can amend the commit now, with

git commit --amend

Once you are satisfied with your changes, run

git rebase --continue

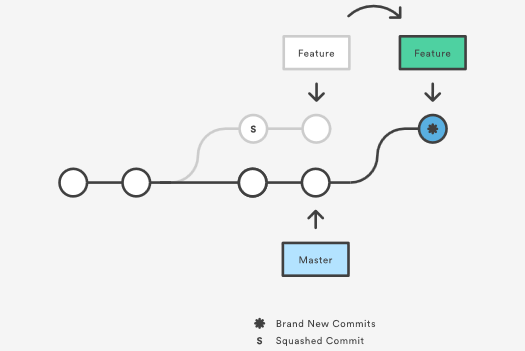
#### Multiple messages

Each regular Git commit will have a log message explaining what happened in the commit. These messages provide valuable insight into the project history. During a rebase, you can run a few commands on commits to modify commit messages.

* Reword or 'r' will stop rebase playback and let you rewrite the individual commit message during.
* Squash or 's' during rebase playback, any commits marked s will be paused on and you will be prompted to edit the separate commit messages into a combined message. More on this in the squash commits section below.
* Fixup or 'f' has the same combining effect as squash. Unlike squash, fixup commits will not interrupt rebase playback to open an editor to combine commit messages. The commits marked 'f' will have their messages discarded in-favor of the previous commit's message.

#### Squash commits for a clean history

The s "squash" command is where we see the true utility of rebase. Squash allows you to specify which commits you want to merge into the previous commits. This is what enables a "clean history." During rebase playback, Git will execute the specified rebase command for each commit. In the case of squash commits, Git will open your configured text editor and prompt to combine the specified commit messages. This entire process can be visualized as follows:



Note that the commits modified with a rebase command have a different ID than either of the original commits. Commits marked with pick will have a new ID if the previous commits have been rewritten.

Modern Git hosting solutions like Bitbucket now offer "auto squashing" features upon merge. These features will automatically rebase and squash a branch's commits for you when utilizing the hosted solutions UI. For more info see "[Squash commits when merging a Git branch with Bitbucket](https://blog.bitbucket.org/2017/01/31/git-squash-commits-merging-bitbucket/)."

### Recap

Git rebase gives you the power to modify your history, and interactive rebasing allows you to do so without leaving a “messy” trail. This creates the freedom to make and correct errors and refine your work, while still maintaining a clean, linear project history.

## The safety net: git reflog

Reference logs, or "reflogs" are a mechanism Git uses to record updates applied to tips of branches and other commit references. Reflog allows you to go back to commits even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary. Every time your branch tip is updated for any reason (by switching branches, pulling in new changes, rewriting history or simply by adding new commits), a new entry will be added to the reflog. In this section we will take a high level look at the git reflog command and explore some common uses.

### Usage

git reflog

This displays the reflog for the local repository.

git reflog --relative-date

This shows the reflog with relative date information (e.g. 2 weeks ago).

### Example

To understand git reflog, let's run through an example.

0a2e358 HEAD@{0}: reset: moving to HEAD~2

0254ea7 HEAD@{1}: checkout: moving from 2.2 to master

c10f740 HEAD@{2}: checkout: moving from master to 2.2

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled HEAD@{0}.

If it turns out that you accidentally moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentally dropped 2 commits.

git reset --hard 0254ea7

Using Git reset, it is now possible to change master back to the commit it was before. This provides a safety net in case the history was accidentally changed.

It's important to note that the reflog only provides a safety net if changes have been committed to your local repository and that it only tracks movements of the repositories branch tip. Additionally reflog entries have an expiration date. The default expiration time for reflog entries is 90 days.

For additional information, see our [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog) page.

## Summary

In this article we discussed several methods of changing git history, and undoing git changes. We took a high level look at the git rebase process. Some Key takeaways are:

* There are many ways to rewrite history with git.
* Use git commit --amend to change your latest log message.
* Use git commit --amend to make modifications to the most recent commit.
* Use git rebase to combine commits and modify history of a branch.
* git rebase -i gives much more fine grained control over history modifications than a standard git rebase.

Learn more about the commands we covered at their individual pages:

* [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)
* [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog)

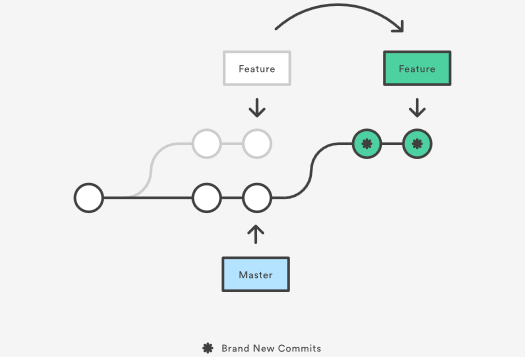
# git rebase

This document will serve as an in-depth discussion of the git rebase command. The Rebase command has also been looked at on the [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) and [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history) pages. This page will take a more detailed look at git rebase configuration and execution. Common Rebase use cases and pitfalls will be covered here.

Rebase is one of two Git utilities that specializes in integrating changes from one branch onto another. The other change integration utility is git merge. Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features. For a detailed look at Merge vs. Rebase, visit our [Merging vs Rebasing guide](https://www.atlassian.com/git/tutorials/merging-vs-rebasing). Rebase itself has 2 main modes: "manual" and "interactive" mode. We will cover the different Rebase modes in more detail below.

## What is git rebase?

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

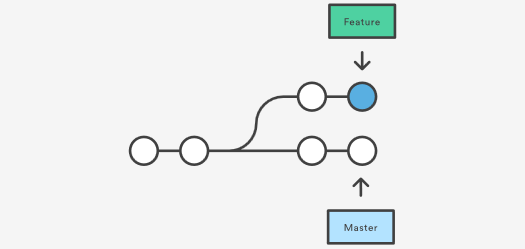
## Usage

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch. Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression. A more real-world scenario would be:

1. A bug is identified in the master branch. A feature that was working successfully is now broken.
2. A developer examines the history of the master branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
3. The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
4. Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

Learn more about [git log](https://www.atlassian.com/git/tutorials/git-log) and [git bisect](https://git-scm.com/docs/git-bisect) on their individual usage pages.

You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto the master branch facilitates a fast-forward merge.



Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with Git merge results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on what everybody has already done.”

### Don't rebase public history

As we've discussed previously in [rewriting history](https://www.atlassian.com/git/tutorials/rewriting-history), you should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

### Git Rebase Standard vs Git Rebase Interactive

Git rebase interactive is when git rebase accepts an -- i argument. This stands for "Interactive." Without any arguments, the command runs in standard mode. In both cases, let's assume we have created a separate feature branch.

# Create a feature branch based off of master

git checkout -b feature\_branch master

# Edit files

git commit -a -m "Adds new feature"

Git rebase in standard mode will automatically take the commits in your current working branch and apply them to the head of the passed branch.

git rebase

This automatically rebases the current branch onto <base>, which can be any kind of commit reference (for example an ID, a branch name, a tag, or a relative reference to HEAD).

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits. It's like Git commit --amend on steroids.

git rebase --interactive

This rebases the current branch onto <base> but uses an interactive rebasing session. This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves. Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands. The rebasing edit commands are as follows:

pick 2231360 some old commit

pick ee2adc2 Adds new feature

# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)

#

# Commands:

# p, pick = use commit

# r, reword = use commit, but edit the commit message

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

# f, fixup = like "squash", but discard this commit's log message

# x, exec = run command (the rest of the line) using shell

# d, drop = remove commit

#### Additional rebase commands

As detailed in the [rewriting history page](https://www.atlassian.com/git/tutorials/rewriting-history), rebasing can be used to change older and multiple commits, committed files, and multiple messages. While these are the most common applications, git rebase also has additional command options that can be useful in more complex applications.

* git rebase -- d means during playback the commit will be discarded from the final combined commit block.
* git rebase -- p leaves the commit as is. It will not modify the commit's message or content and will still be an individual commit in the branches history.
* git rebase -- x during playback executes a command line shell script on each marked commit. A useful example would be to run your codebase's test suite on specific commits, which may help identify regressions during a rebase.

### Recap

Interactive rebasing gives you complete control over what your project history looks like. This affords a lot of freedom to developers, as it lets them commit a "messy" history while they're focused on writing code, then go back and clean it up after the fact.

Most developers like to use an interactive rebase to polish a feature branch before merging it into the main code base. This gives them the opportunity to squash insignificant commits, delete obsolete ones, and make sure everything else is in order before committing to the “official” project history. To everybody else, it will look like the entire feature was developed in a single series of well-planned commits.

The real power of interactive rebasing can be seen in the history of the resulting master branch. To everybody else, it looks like you're a brilliant developer who implemented the new feature with the perfect amount of commits the first time around. This is how interactive rebasing can keep a project's history clean and meaningful.

### Configuration options

There are a few rebase properties that can be set using git config. These options will alter the git rebase output look and feel.

* **rebase.stat**: A boolean that is set to false by default. The option toggles display of visual diffstat content that shows what changed since the last debase.
* **rebase.autoSquash:** A boolean value that toggles the --autosquash behavior.
* **rebase.missingCommitsCheck:** Can be set to multiple values which change rebase behavior around missing commits.

|  |  |
| --- | --- |
| warn | Prints warning output in interactive mode which warns of removed commits |
| error | Stops the rebase and prints removed commit warning messages |
| ignore | Set by default this ignores any missing commit warnings |

* **rebase.instructionFormat:** A git log format string that will be used for formatting interactive rebase display

### Advanced rebase application

The command line argument --onto can be passed to git rebase. When in git rebase --onto mode the command expands to:

git rebase --onto

The --onto command enables a more powerful form or rebase that allows passing specific refs to be the tips of a rebase.  
Let’s say we have an example repo with branches like:

o---o---o---o---o master

\

o---o---o---o---o featureA

\

o---o---o featureB

featureB is based on featureA, however, we realize featureB is not dependent on any of the changes in featureA and could just be branched off master.

git rebase --onto master featureA featureB

featureA is the <oldbase>. master becomes the <newbase> and featureB is reference for what HEAD of the <newbase> will point to. The results are then:

o---o---o featureB

/

o---o---o---o---o master

\

o---o---o---o---o featureA

## Understanding the dangers of rebase

One caveat to consider when working with Git Rebase is merge conflicts may become more frequent during a rebase workflow. This occurs if you have a long-lived branch that has strayed from master. Eventually you will want to rebase against master and at that time it may contain many new commits that your branch changes may conflict with. This is easily remedied by rebasing your branch frequently against master, and making more frequent commits. The --continue and --abort command line arguments can be passed to git rebase to advance or reset the the rebase when dealing with conflicts.

A more serious rebase caveat is lost commits from interactive history rewriting. Running rebase in interactive mode and executing subcommands like squash or drop will remove commits from your branche's immediate log. At first glance this can appear as though the commits are permanently gone. Using git reflog these commits can be restored and the entire rebase can be undone. For more info on using git reflog to find lost commits, visit our [Git reflog documentation page](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog).

Git Rebase itself is not seriously dangerous. The real danger cases arise when executing history rewriting interactive rebases and force pushing the results to a remote branch that's shared by other users. This is a pattern that should be avoided as it has the capability to overwrite other remote users' work when they pull.

## Recovering from upstream rebase

If another user has rebased and force pushed to the branch that you’re committing to, a git pull will then overwrite any commits you have based off that previous branch with the tip that was force pushed. Luckily, using git reflog you can get the reflog of the remote branch. On the remote branch's reflog you can find a ref before it was rebased. You can then rebase your branch against that remote ref using the --onto option as discussed above in the Advanced Rebase Application section.

## Summary

In this article we covered git rebase usage. We discussed basic and advanced use cases and more advanced examples. Some key discussion points are:

* git rebase standard vs interactive modes
* git rebase configuration options
* git rebase --onto
* git rebase lost commits

We looked at git rebase usage with other tools like [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog), [git fetch](https://www.atlassian.com/git/tutorials/syncing#git-fetch), and [git push](https://www.atlassian.com/git/tutorials/syncing#git-push). Visit their corresponding pages for further information.

# git reflog

This page provides a detailed discussion of the git reflog command. Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit. Common examples include:

* git checkout
* git reset
* git merge

Reflogs track when Git refs were updated in the local repository. In addition to branch tip reflogs, a special reflog is maintained for the Git stash. Reflogs are stored in directories under the local repository's .git directory. git reflog directories can be found at .git/logs/refs/heads/., .git/logs/HEAD, and also .git/logs/refs/stash if the git stash has been used on the repo.

We discussed git reflog at a high level on the [Rewriting History Page](https://www.atlassian.com/git/tutorials/rewriting-history). This document will cover: extended configuration options of git reflog, common use-cases and pitfalls of git reflog, how to undo changes with git reflog, and more.

## Basic usage

The most basic Reflog use case is invoking:

git reflog

This is essentially a short cut that's equivalent to:

git reflog show HEAD

This will output the HEAD reflog. You should see output similar to:

eff544f HEAD@{0}: commit: migrate existing content

bf871fd HEAD@{1}: commit: Add Git Reflog outline

9a4491f HEAD@{2}: checkout: moving from master to git\_reflog

9a4491f HEAD@{3}: checkout: moving from Git\_Config to master

39b159a HEAD@{4}: commit: expand on git context

9b3aa71 HEAD@{5}: commit: more color clarification

f34388b HEAD@{6}: commit: expand on color support

9962aed HEAD@{7}: commit: a git editor -> the Git editor

Visit the [Rewriting History page](https://www.atlassian.com/git/tutorials/rewriting-history) for another example of common reflog access.

### Reflog references

By default, git reflog will output the reflog of the HEAD ref. HEAD is a symbolic reference to the currently active branch. Reflogs are available for other refs as well. The syntax to access a git ref is name@{qualifier}. In addition to HEAD refs, other branches, tags, remotes, and the Git stash can be referenced as well.

You can get a complete reflog of all refs by executing:

git reflog show --all

To see the reflog for a specific branch pass that branch name to git reflog show

git reflog show otherbranch

9a4491f otherbranch@{0}: commit: seperate articles into branch PRs

35aee4a otherbranch{1}: commit (initial): initial commit add git-init and setting-up-a-repo docs

Executing this example will show a reflog for the otherbranch branch. The following example assumes you have previously stashed some changes using the git stash command.

git reflog stash

0d44de3 stash@{0}: WIP on git\_reflog: c492574 flesh out intro

This will output a reflog for the Git stash. The returned ref pointers can be passed to other Git commands:

git diff stash@{0} otherbranch@{0}

When executed, this example code will display Git diff output comparing the stash@{0} changes against the otherbranch@{0} ref.

### Timed reflogs

Every reflog entry has a timestamp attached to it. These timestamps can be leveraged as the qualifier token of Git ref pointer syntax. This enables filtering Git reflogs by time. The following are some examples of available time qualifiers:

* 1.minute.ago
* 1.hour.ago
* 1.day.ago
* yesterday
* 1.week.ago
* 1.month.ago
* 1.year.ago
* 2011-05-17.09:00:00

Time qualifiers can be combined (e.g. 1.day.2.hours.ago), Additionally plural forms are accepted (e.g. 5.minutes.ago).

Time qualifier refs can be passed to other git commands.

git diff master@{0} master@{1.day.ago}

This example will diff the current master branch against master 1 day ago. This example is very useful if you want to know changes that have occurred within a time frame.

## Subcommands & configuration options

git reflog accepts few addition arguments which are considered subcommands.

### Show - git reflog show

show is implicitly passed by default. For example, the command:

git reflog master@{0}

is equivalent to the command:

git reflog show master@{0}

In addition, git reflog show is an alias for git log -g --abbrev-commit --pretty=oneline. Executing git reflog show will display the log for the passed <refid>.

### Expire - git reflog expire

The expire subcommand cleans up old or unreachable reflog entries. The expire subcommand has potential for data loss. This subcommand is not typically used by end users, but used by git internally. Passing a -n or --dry-run option to git reflog expire Will perform a "dry run" which will output which reflog entries are marked to be pruned, but will not actually prune them.

By default, the reflog expiration date is set to 90 days. An expire time can be specified by passing a command line argument --expire=time to git reflog expire or by setting a git configuration name of gc.reflogExpire.

### Delete - git reflog delete

The delete subcommand is self explanatory and will delete a passed in reflog entry. As with expire, delete has potential to lose data and is not commonly invoked by end users.

## Recovering lost commits

Git never really loses anything, even when performing history rewriting operations like rebasing or commit amending. For the next example let's assume that we have made some new changes to our repo. Our git log --pretty=oneline looks like the following:

338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content

1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content

c49257493a95185997c87e0bc3a9481715270086 flesh out intro

eff544f986d270d7f97c77618314a06f024c7916 migrate existing content

bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

We then commit those changes and execute the following:

#make changes to HEAD

git commit -am "some WIP changes"

With the addition of the new commit. The log now looks like:

37656e19d4e4f1a9b419f57850c8f1974f871b07 some WIP changes

338fbcb41de10f7f2e54095f5649426cb4bf2458 extended content

1e63ceab309da94256db8fb1f35b1678fb74abd4 bunch of content

c49257493a95185997c87e0bc3a9481715270086 flesh out intro

eff544f986d270d7f97c77618314a06f024c7916 migrate existing content

bf871fd762d8ef2e146d7f0226e81a92f91975ad Add Git Reflog outline

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

At this point we perform an interactive rebase against the master branch by executing...

git rebase -i origin/master

During the rebase we mark commits for squash with the s rebase subcommand. During the rebase, we squash a few commits into the most recent "some WIP changes" commit.

Because we squashed commits the git log output now looks like:

40dhsoi37656e19d4e4f1a9b419f57850ch87dah987698hs some WIP changes

35aee4a4404c42128bee8468a9517418ed0eb3dc initial commit add git-init and setting-up-a-repo docs

If we examine git log at this point it appears that we no longer have the commits that were marked for squashing. What if we want to operate on one of the squashed commits? Maybe to remove its changes from history? This is an opportunity to leverage the reflog.

git reflog

37656e1 HEAD@{0}: rebase -i (finish): returning to refs/heads/git\_reflog

37656e1 HEAD@{1}: rebase -i (start): checkout origin/master

37656e1 HEAD@{2}: commit: some WIP changes

We can see there are reflog entries for the start and finish of the rebase and prior to those is our "some WIP changes" commit. We can pass the reflog ref to git reset and reset to a commit that was before the rebase.

git reset HEAD@{2}

Executing this reset command will move HEAD to the commit where "some WIP changes" was added, essentially restoring the other squashed commits.

## Summary

In this tutorial we discussed the git reflog command. Some key points covered were:

* How to view reflog for specific branches
* How to undo a git rebase using the reflog
* How specify and view time based reflog entries

We briefly mentioned that git reflog can be used with other git commands like [git checkout](https://www.atlassian.com/git/tutorials/using-branches#git-checkout), [git reset](https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting), and [git merge](https://www.atlassian.com/git/tutorials/git-merge). Learn more at their respective pages. For additional discussion on refs and the reflog, [learn more here](https://www.atlassian.com/git/tutorials/refs-and-the-reflog).

# Advanced Git Tutorials

[Atlassian’s Git tutorials](https://www.atlassian.com/git/tutorials) introduce the most common Git commands, and our [Git Workflows](https://www.atlassian.com/git/tutorials/comparing-workflows) modules discuss how these commands are typically used to facilitate collaboration. Alone, these are enough to get a development team up and running with Git. But, if you really want to leverage the full power of Git, you’re ready to dive into our Advanced Git articles.

Each of these articles provide an in-depth discussion of an advanced feature of Git. Instead of presenting new commands and concepts, they refine your existing Git skills by explaining what’s going on under the hood. Armed with this knowledge, you’ll be able to use familiar Git commands more effectively. More importantly, you’ll never be scared of breaking your Git repository because you’ll understand why it broke and how to fix it.

## Merging vs. Rebasing

Git is all about working with divergent history. Its git merge and git rebase commands offer alternative ways to integrate commits from different branches, and both options come with their own advantages. In this article, we’ll discuss how and when a basic git merge operation can be replaced with a rebase.

## Resetting, Checking Out, and Reverting

The git reset, git checkout, and git revert commands are all similar in that they undo some type of change in your repository. But, they all affect different combinations of the working directory, staged snapshot, and commit history. This article clearly defines how these commands differ and when each of them should be used in the standard Git workflows.

## Advanced Git Log

The git log command is what makes your project history useful. Without it, you wouldn’t be able to access any of your commits. But, if you’re like most aspiring Git users, you’ve probably only scratched the surface of what’s possible with git log. This article walks you through its advanced formatting and filtering options, giving you the power to extract all sorts of interesting information from your Git repository.

## Git Hooks

If you want to perform custom actions when a certain event takes place in a Git repository, hooks are your tool of choice. They let you normalize commit messages, automate testing suites, notify continuous integration systems, and much more. After this article, you’ll understand the many ways in which Git hooks can streamline your workflow.

## Refs and the Reflog

A **ref** is Git’s internal way of referring to a commit. You’re already familiar with many categories of refs, including commit hashes and branch names. But, there are many other types of refs, and virtually every Git command utilizes them in some form or another. You’ll walk away from this article with an intimate knowledge of Git’s inner workings.

# Merging vs. Rebasing

[Conceptual Overview](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#conceptual-overview) [The Golden Rule of Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing) [Workflow Walkthrough](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#workflow-walkthrough) [Summary](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#summary)

The git rebase command has a reputation for being magical Git voodoo that beginners should stay away from, but it can actually make life much easier for a development team when used with care. In this article, we’ll compare git rebase with the related git merge command and identify all of the potential opportunities to incorporate rebasing into the typical Git workflow.

## Conceptual Overview

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

Now, let’s say that the new commits in master are relevant to the feature that you’re working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

### The Merge Option

The easiest option is to merge the master branch into the feature branch using something like the following:

git checkout feature

git merge master

Or, you can condense this to a one-liner:

git merge feature master

This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:

Merging is nice because it’s a non-destructive operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

### The Rebase Option

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

git checkout feature

git rebase master

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writes the project history by creating brand new commits for each commit in the original branch.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.

But, there are two trade-offs for this pristine commit history: safety and traceability. If you don’t follow the [Golden Rule of Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing), re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit—you can’t see when upstream changes were incorporated into the feature.

### Interactive Rebasing

Interactive rebasing gives you the opportunity to alter commits as they are moved to the new branch. This is even more powerful than an automated rebase, since it offers complete control over the branch’s commit history. Typically, this is used to clean up a messy history before merging a feature branch into master.

To begin an interactive rebasing session, pass the i option to the git rebase command:

git checkout feature

git rebase -i master

This will open a text editor listing all of the commits that are about to be moved:

pick 33d5b7a Message for commit #1

pick 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

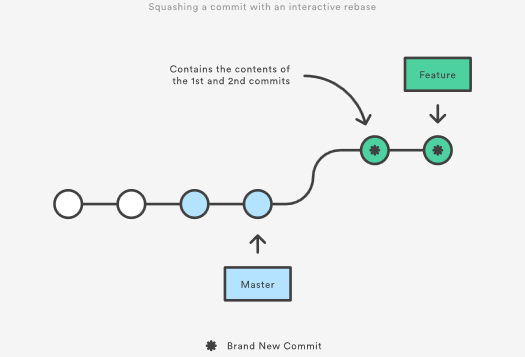
This listing defines exactly what the branch will look like after the rebase is performed. By changing the pick command and/or re-ordering the entries, you can make the branch’s history look like whatever you want. For example, if the 2nd commit fixes a small problem in the 1st commit, you can condense them into a single commit with the fixup command:

pick 33d5b7a Message for commit #1

fixup 9480b3d Message for commit #2

pick 5c67e61 Message for commit #3

When you save and close the file, Git will perform the rebase according to your instructions, resulting in project history that looks like the following:



Eliminating insignificant commits like this makes your feature’s history much easier to understand. This is something that git merge simply cannot do.

## The Golden Rule of Rebasing

Once you understand what rebasing is, the most important thing to learn is when not to do it. The golden rule of git rebase is to never use it on public branches.

For example, think about what would happen if you rebased master onto your feature branch:

The rebase moves all of the commits in master onto the tip of feature. The problem is that this only happened in your repository. All of the other developers are still working with the original master. Since rebasing results in brand new commits, Git will think that your master branch’s history has diverged from everybody else’s.

The only way to synchronize the two master branches is to merge them back together, resulting in an extra merge commit and two sets of commits that contain the same changes (the original ones, and the ones from your rebased branch). Needless to say, this is a very confusing situation.

So, before you run git rebase, always ask yourself, “Is anyone else looking at this branch?” If the answer is yes, take your hands off the keyboard and start thinking about a non-destructive way to make your changes (e.g., the git revert command). Otherwise, you’re safe to re-write history as much as you like.

### Force-Pushing

If you try to push the rebased master branch back to a remote repository, Git will prevent you from doing so because it conflicts with the remote master branch. But, you can force the push to go through by passing the --force flag, like so:

# Be very careful with this command!

git push --force

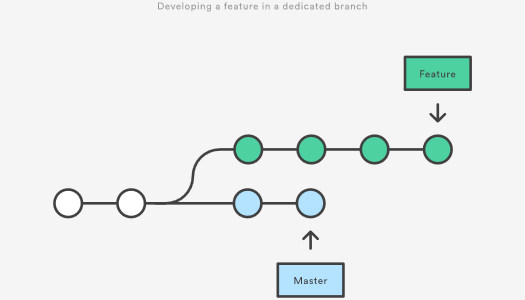
This overwrites the remote master branch to match the rebased one from your repository and makes things very confusing for the rest of your team. So, be very careful to use this command only when you know exactly what you’re doing.

One of the only times you should be force-pushing is when you’ve performed a local cleanup after you’ve pushed a private feature branch to a remote repository (e.g., for backup purposes). This is like saying, “Oops, I didn’t really want to push that original version of the feature branch. Take the current one instead.” Again, it’s important that nobody is working off of the commits from the original version of the feature branch.

## Workflow Walkthrough

Rebasing can be incorporated into your existing Git workflow as much or as little as your team is comfortable with. In this section, we’ll take a look at the benefits that rebasing can offer at the various stages of a feature’s development.

The first step in any workflow that leverages git rebase is to create a dedicated branch for each feature. This gives you the necessary branch structure to safely utilize rebasing:



### Local Cleanup

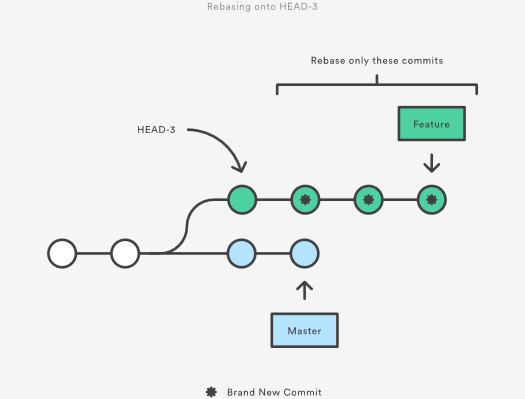
One of the best ways to incorporate rebasing into your workflow is to clean up local, in-progress features. By periodically performing an interactive rebase, you can make sure each commit in your feature is focused and meaningful. This lets you write your code without worrying about breaking it up into isolated commits—you can fix it up after the fact.

When calling git rebase, you have two options for the new base: The feature’s parent branch (e.g., master), or an earlier commit in your feature. We saw an example of the first option in the Interactive Rebasing section. The latter option is nice when you only need to fix up the last few commits. For example, the following command begins an interactive rebase of only the last 3 commits.

git checkout feature

git rebase -i HEAD~3

By specifying HEAD~3 as the new base, you’re not actually moving the branch—you’re just interactively re-writing the 3 commits that follow it. Note that this will not incorporate upstream changes into the feature branch.



If you want to re-write the entire feature using this method, the git merge-base command can be useful to find the original base of the feature branch. The following returns the commit ID of the original base, which you can then pass to git rebase:

git merge-base feature master

This use of interactive rebasing is a great way to introduce git rebase into your workflow, as it only affects local branches. The only thing other developers will see is your finished product, which should be a clean, easy-to-follow feature branch history.

But again, this only works for private feature branches. If you’re collaborating with other developers via the same feature branch, that branch is public, and you’re not allowed to re-write its history.

There is no git merge alternative for cleaning up local commits with an interactive rebase.

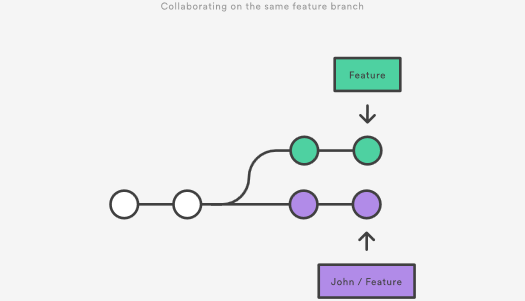
### Incorporating Upstream Changes Into a Feature

In the Conceptual Overview section, we saw how a feature branch can incorporate upstream changes from master using either git merge or git rebase. Merging is a safe option that preserves the entire history of your repository, while rebasing creates a linear history by moving your feature branch onto the tip of master.

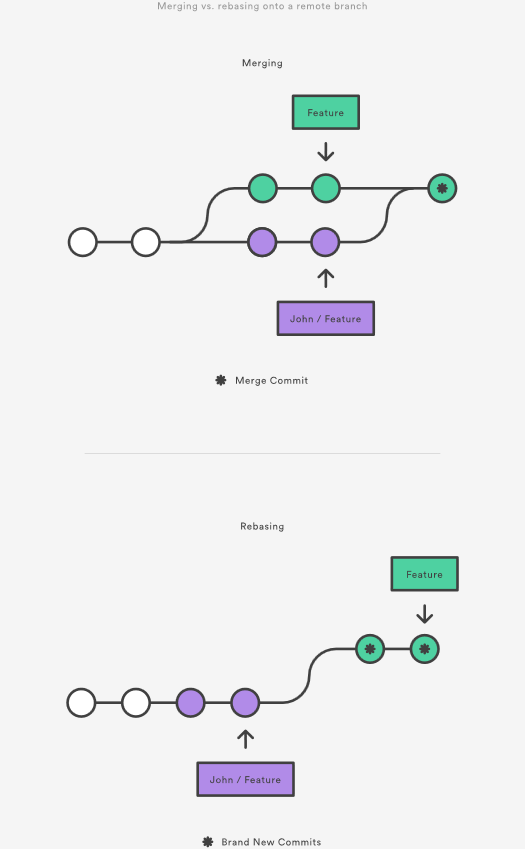
This use of git rebase is similar to a local cleanup (and can be performed simultaneously), but in the process it incorporates those upstream commits from master.

Keep in mind that it’s perfectly legal to rebase onto a remote branch instead of master. This can happen when collaborating on the same feature with another developer and you need to incorporate their changes into your repository.

For example, if you and another developer named John added commits to the feature branch, your repository might look like the following after fetching the remote feature branch from John’s repository:



You can resolve this fork the exact same way as you integrate upstream changes from master: either merge your local feature with john/feature, or rebase your local feature onto the tip of john/feature.



Note that this rebase doesn’t violate the Golden Rule of Rebasing because only your local feature commits are being moved—everything before that is untouched. This is like saying, “add my changes to what John has already done.” In most circumstances, this is more intuitive than synchronizing with the remote branch via a merge commit.

By default, the git pull command performs a merge, but you can force it to integrate the remote branch with a rebase by passing it the --rebase option.

### Reviewing a Feature With a Pull Request

If you use pull requests as part of your code review process, you need to avoid using git rebase after creating the pull request. As soon as you make the pull request, other developers will be looking at your commits, which means that it’s a public branch. Re-writing its history will make it impossible for Git and your teammates to track any follow-up commits added to the feature.

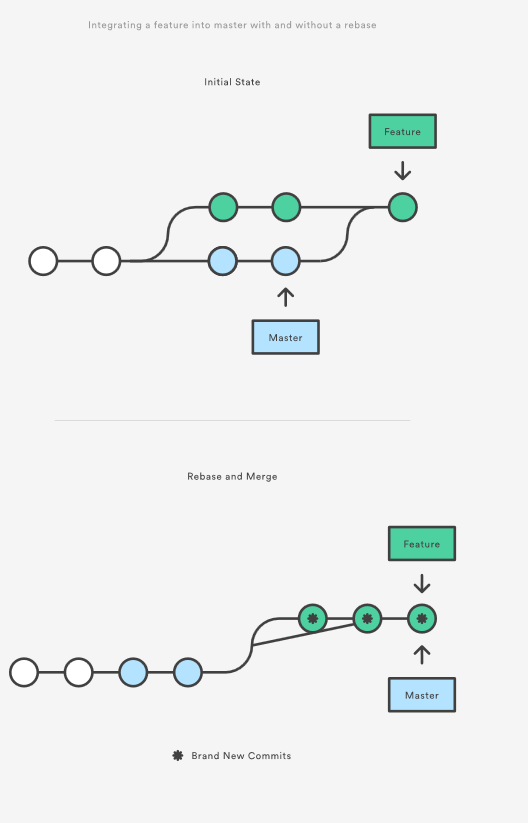
Any changes from other developers need to be incorporated with git merge instead of git rebase.

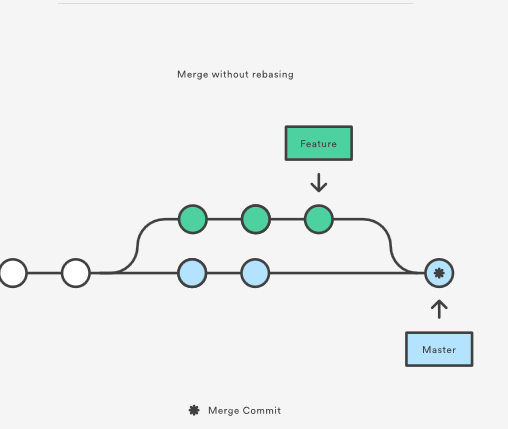
For this reason, it’s usually a good idea to clean up your code with an interactive rebase before submitting your pull request.

### Integrating an Approved Feature

After a feature has been approved by your team, you have the option of rebasing the feature onto the tip of the master branch before using git merge to integrate the feature into the main code base.

This is a similar situation to incorporating upstream changes into a feature branch, but since you’re not allowed to re-write commits in the master branch, you have to eventually use git merge to integrate the feature. However, by performing a rebase before the merge, you’re assured that the merge will be fast-forwarded, resulting in a perfectly linear history. This also gives you the chance to squash any follow-up commits added during a pull request.





If you’re not entirely comfortable with git rebase, you can always perform the rebase in a temporary branch. That way, if you accidentally mess up your feature’s history, you can check out the original branch and try again. For example:

git checkout feature

git checkout -b temporary-branch

git rebase -i master

# [Clean up the history]

git checkout master

git merge temporary-branch

## Summary

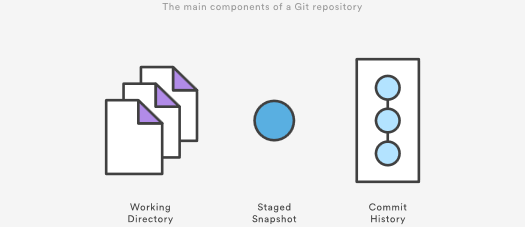
And that’s all you really need to know to start rebasing your branches. If you would prefer a clean, linear history free of unnecessary merge commits, you should reach for git rebase instead of git merge when integrating changes from another branch.

On the other hand, if you want to preserve the complete history of your project and avoid the risk of re-writing public commits, you can stick with git merge. Either option is perfectly valid, but at least now you have the option of leveraging the benefits of git rebase.

# Resetting, Checking Out & Reverting

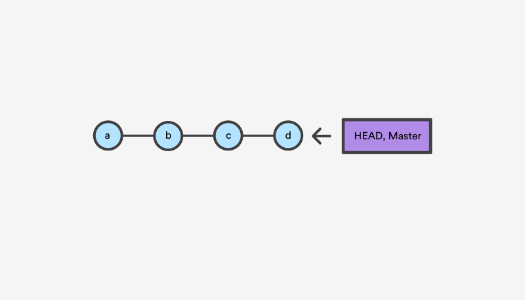
The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) commands are some of the most useful tools in your Git toolbox. They all let you undo some kind of change in your repository, and the first two commands can be used to manipulate either commits or individual files.

Because they’re so similar, it’s very easy to mix up which command should be used in any given development scenario. In this article, we’ll compare the most common configurations of git reset, git checkout, and git revert. Hopefully, you’ll walk away with the confidence to navigate your repository using any of these commands.

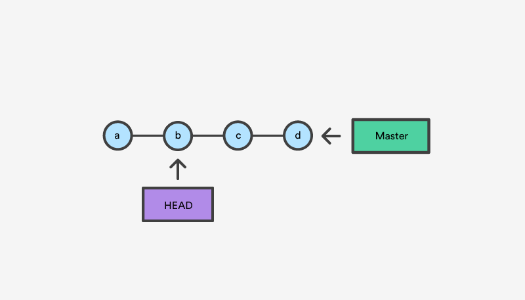


It helps to think about each command in terms of their effect on the three state management mechanisms of a Git repository: the working directory, the staged snapshot, and the commit history. These components are sometimes know as "The three trees" of Git. We explore the three trees in depth on the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page. Keep these mechanisms in mind as you read through this article.

A checkout is an operation that moves the HEAD ref pointer to a specified commit. To demonstrate this consider the following example.



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



This is an update to the "Commit History" tree. The git checkout command can be used in a commit, or file level scope. A file level checkout will change the file's contents to those of the specific commit.

A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit. git revert can only be run at a commit level scope and has no file level functionality.

A reset is an operation that takes a specified commit and resets the "three trees" to match the state of the repository at that specified commit. A reset can be invoked in three different modes which correspond to the three trees.

Checkout and reset are generally used for making local or private 'undos'. They modify the history of a repository that can cause conflicts when pushing to remote shared repositories. Revert is considered a safe operation for 'public undos' as it creates new history which can be shared remotely and doesn't overwrite history remote team members may be dependent on.

## Git Reset vs Revert vs Checkout reference

The table below sums up the most common use cases for all of these commands. Be sure to keep this reference handy, as you’ll undoubtedly need to use at least some of them during your Git career.

| **Command** | **Scope** | **Common use cases** |
| --- | --- | --- |
| git reset | Commit-level | Discard commits in a private branch or throw away uncommited changes |
| git reset | File-level | Unstage a file |
| git checkout | Commit-level | Switch between branches or inspect old snapshots |
| git checkout | File-level | Discard changes in the working directory |
| git revert | Commit-level | Undo commits in a public branch |
| git revert | File-level | (N/A) |

## Commit Level Operations

The parameters that you pass to git reset and git checkout determine their scope. When you don’t include a file path as a parameter, they operate on whole commits. That’s what we’ll be exploring in this section. Note that git revert has no file-level counterpart.

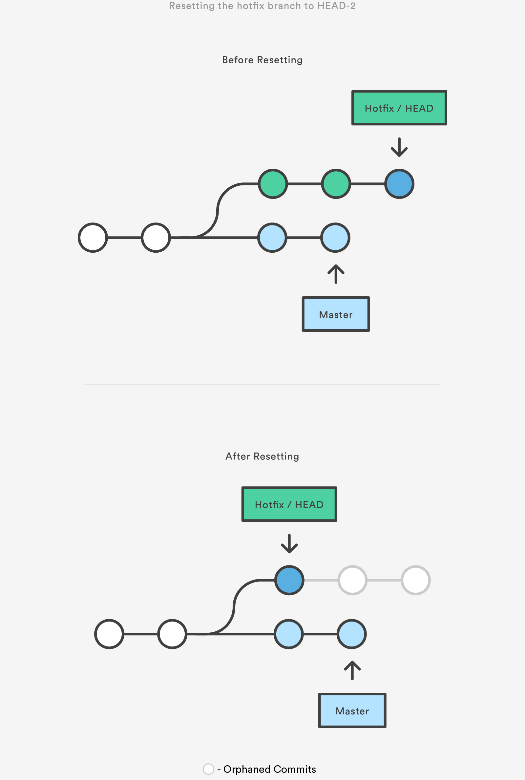
### Reset A Specific Commit

On the commit-level, resetting is a way to move the tip of a branch to a different commit. This can be used to remove commits from the current branch. For example, the following command moves the hotfix branch backwards by two commits.

git checkout hotfix

git reset HEAD~2

The two commits that were on the end of hotfix are now dangling, or orphaned commits. This means they will be deleted the next time Git performs a garbage collection. In other words, you’re saying that you want to throw away these commits. This can be visualized as the following:



This usage of git reset is a simple way to undo changes that haven’t been shared with anyone else. It’s your go-to command when you’ve started working on a feature and find yourself thinking, “Oh crap, what am I doing? I should just start over.”

In addition to moving the current branch, you can also get git reset to alter the staged snapshot and/or the working directory by passing it one of the following flags:

* --soft – The staged snapshot and working directory are not altered in any way.
* --mixed – The staged snapshot is updated to match the specified commit, but the working directory is not affected. This is the default option.
* --hard – The staged snapshot and the working directory are both updated to match the specified commit.

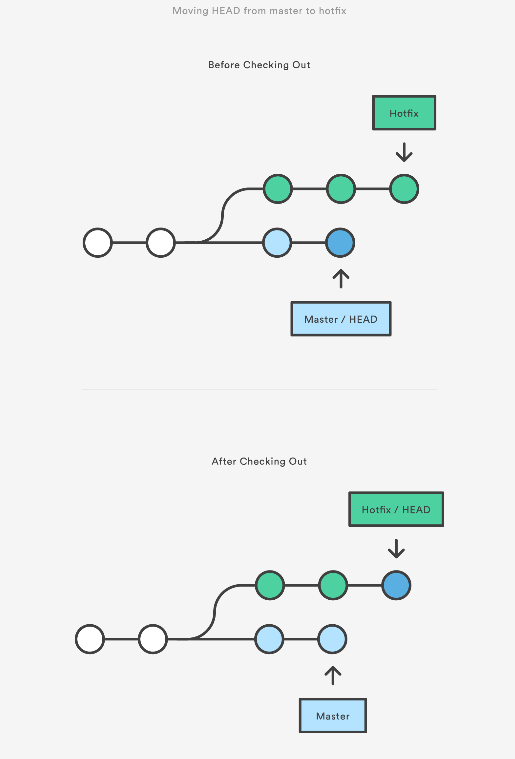
It’s easier to think of these modes as defining the scope of a git reset operation. For further detailed information visit the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page.

### Checkout old commits

The git checkout command is used to update the state of the repository to a specific point in the projects history. When passed with a branch name, it lets you switch between branches.

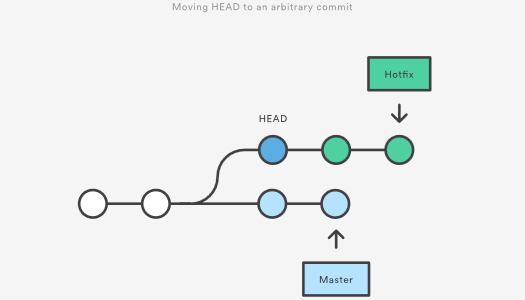
git checkout hotfix

Internally, all the above command does is move HEAD to a different branch and update the working directory to match. Since this has the potential to overwrite local changes, Git forces you to commit or [stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) any changes in the working directory that will be lost during the checkout operation. Unlike git reset, git checkout doesn’t move any branches around.



You can also check out arbitrary commits by passing the commit reference instead of a branch. This does the exact same thing as checking out a branch: it moves the HEAD reference to the specified commit. For example, the following command will check out the grandparent of the current commit:

git checkout HEAD~2



his is useful for quickly inspecting an old version of your project. However, since there is no branch reference to the current HEAD, this puts you in a detached HEAD state. This can be dangerous if you start adding new commits because there will be no way to get back to them after you switch to another branch. For this reason, you should always create a new branch before adding commits to a detached HEAD.

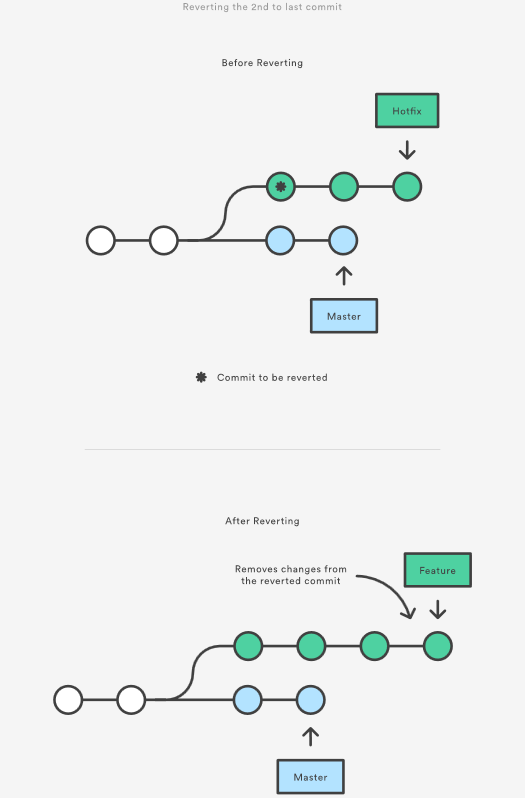
### Undo Public Commits with Revert

Reverting undoes a commit by creating a new commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history. For example, the following command will figure out the changes contained in the 2nd to last commit, create a new commit undoing those changes, and tack the new commit onto the existing project.

git checkout hotfix

git revert HEAD~2

This can be visualized as the following:



Contrast this with git reset, which does alter the existing commit history. For this reason, git revert should be used to undo changes on a public branch, and git reset should be reserved for undoing changes on a private branch.

You can also think of git revert as a tool for undoing committed changes, while git reset HEAD is for undoing uncommitted changes.

Like git checkout, git revert has the potential to overwrite files in the working directory, so it will ask you to commit or [stash changes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) that would be lost during the revert operation.

## File-level Operations

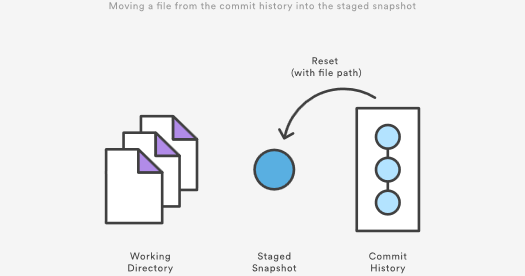
The git reset and git checkout commands also accept an optional file path as a parameter. This dramatically alters their behavior. Instead of operating on entire snapshots, this forces them to limit their operations to a single file.

### Git Reset A Specific File

When invoked with a file path, git reset updates the staged snapshot to match the version from the specified commit. For example, this command will fetch the version of foo.py in the 2nd-to-last commit and stage it for the next commit:

git reset HEAD~2 foo.py

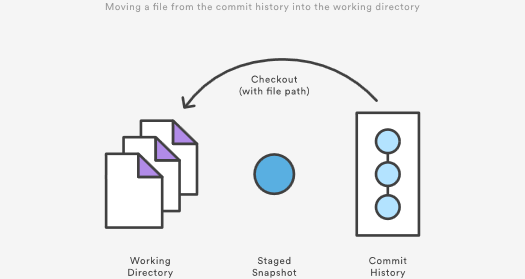
As with the commit-level version of git reset, this is more commonly used with HEAD rather than an arbitrary commit. Running git reset HEAD foo.py will unstage foo.py. The changes it contains will still be present in the working directory.



The --soft, --mixed, and --hard flags do not have any effect on the file-level version of git reset, as the staged snapshot is always updated, and the working directory is never updated.

### Git Checkout File

Checking out a file is similar to using git reset with a file path, except it updates the working directory instead of the stage. Unlike the commit-level version of this command, this does not move the HEAD reference, which means that you won’t switch branches.



For example, the following command makes foo.py in the working directory match the one from the 2nd-to-last commit:

git checkout HEAD~2 foo.py

Just like the commit-level invocation of git checkout, this can be used to inspect old versions of a project—but the scope is limited to the specified file.

If you stage and commit the checked-out file, this has the effect of “reverting” to the old version of that file. Note that this removes all of the subsequent changes to the file, whereas the git revert command undoes only the changes introduced by the specified commit.

Like git reset, this is commonly used with HEAD as the commit reference. For instance, git checkout HEAD foo.py has the effect of discarding unstaged changes to foo.py. This is similar behavior to git reset HEAD --hard, but it operates only on the specified file.

## Summary

You should now have all the tools you could ever need to undo changes in a Git repository. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git revert](https://www.atlassian.com/git/tutorials/undoing-changes/git-revert) commands can be confusing, but when you think about their effects on the working directory, staged snapshot, and commit history, it should be easier to discern which command fits the development task at hand.

# Advanced Git log

[Formatting Log Output](https://www.atlassian.com/git/tutorials/git-log#formatting-log-output) [Filtering the Commit History](https://www.atlassian.com/git/tutorials/git-log#filtering-the-commit-history) [Summary](https://www.atlassian.com/git/tutorials/git-log#summary)

The purpose of any version control system is to record changes to your code. This gives you the power to go back into your project history to see who contributed what, figure out where bugs were introduced, and revert problematic changes. But, having all of this history available is useless if you don’t know how to navigate it. That’s where the git log command comes in.

By now, you should already know the basic git log command for displaying commits. But, you can alter this output by passing many different parameters to git log.

The advanced features of git log can be split into two categories: formatting how each commit is displayed, and filtering which commits are included in the output. Together, these two skills give you the power to go back into your project and find any information that you could possibly need.

## Formatting Log Output

First, this article will take a look at the many ways in which git log’s output can be formatted. Most of these come in the form of flags that let you request more or less information from git log.

If you don’t like the default git log format, you can use git config’s aliasing functionality to create a shortcut for any of the formatting options discussed below. Please see in [The git config Command](https://www.atlassian.com/git/tutorials/setting-up-a-repository#git-config) for how to set up an alias.

### Oneline

The --oneline flag condenses each commit to a single line. By default, it displays only the commit ID and the first line of the commit message. Your typical git log --oneline output will look something like this:

0e25143 Merge branch 'feature'

ad8621a Fix a bug in the feature

16b36c6 Add a new feature

23ad9ad Add the initial code base

This is very useful for getting a high-level overview of your project.

### Decorating

Many times it’s useful to know which branch or tag each commit is associated with. The --decorate flag makes git log display all of the references (e.g., branches, tags, etc) that point to each commit.

This can be combined with other configuration options. For example, running git log --oneline --decorate will format the commit history like so:

0e25143 (HEAD, master) Merge branch 'feature'

ad8621a (feature) Fix a bug in the feature

16b36c6 Add a new feature

23ad9ad (tag: v0.9) Add the initial code base

This lets you know that the top commit is also checked out (denoted by HEAD) and that it is also the tip of the master branch. The second commit has another branch pointing to it called feature, and finally the 4th commit is tagged as v0.9.

Branches, tags, HEAD, and the commit history are almost all of the information contained in your Git repository, so this gives you a more complete view of the logical structure of your repository.

### Diffs

The git log command includes many options for displaying diffs with each commit. Two of the most common options are --stat and -p.

The --stat option displays the number of insertions and deletions to each file altered by each commit (note that modifying a line is represented as 1 insertion and 1 deletion). This is useful when you want a brief summary of the changes introduced by each commit. For example, the following commit added 67 lines to the hello.py file and removed 38 lines:

commit f2a238924e89ca1d4947662928218a06d39068c3

Author: John <john@example.com>

Date: Fri Jun 25 17:30:28 2014 -0500

Add a new feature

hello.py | 105 ++++++++++++++++++++++++-----------------

1 file changed, 67 insertion(+), 38 deletions(-)

The amount of + and - signs next to the file name show the relative number of changes to each file altered by the commit. This gives you an idea of where the changes for each commit can be found.

If you want to see the actual changes introduced by each commit, you can pass the -p option to git log. This outputs the entire patch representing that commit:

commit 16b36c697eb2d24302f89aa22d9170dfe609855b

Author: Mary <mary@example.com>

Date: Fri Jun 25 17:31:57 2014 -0500

Fix a bug in the feature

diff --git a/hello.py b/hello.py

index 18ca709..c673b40 100644

--- a/hello.py

+++ b/hello.py

@@ -13,14 +13,14 @@ B

-print("Hello, World!")

+print("Hello, Git!")

For commits with a lot of changes, the resulting output can become quite long and unwieldy. More often than not, if you’re displaying a full patch, you’re probably searching for a specific change. For this, you want to use the pickaxe option.

### The Shortlog

The git shortlog command is a special version of git log intended for creating release announcements. It groups each commit by author and displays the first line of each commit message. This is an easy way to see who’s been working on what.

For example, if two developers have contributed 5 commits to a project, the git shortlog output might look like the following:

Mary (2):

Fix a bug in the feature

Fix a serious security hole in our framework

John (3):

Add the initial code base

Add a new feature

Merge branch 'feature'

By default, git shortlog sorts the output by author name, but you can also pass the -n option to sort by the number of commits per author.

### Graphs

The --graph option draws an ASCII graph representing the branch structure of the commit history. This is commonly used in conjunction with the --oneline and --decorate commands to make it easier to see which commit belongs to which branch:

git log --graph --oneline --decorate

For a simple repository with just 2 branches, this will produce the following:

\* 0e25143 (HEAD, master) Merge branch 'feature'

|\

| \* 16b36c6 Fix a bug in the new feature

| \* 23ad9ad Start a new feature

\* | ad8621a Fix a critical security issue

|/

\* 400e4b7 Fix typos in the documentation

\* 160e224 Add the initial code base

The asterisk shows which branch the commit was on, so the above graph tells us that the 23ad9ad and 16b36c6 commits are on a topic branch and the rest are on the master branch.

While this is a nice option for simple repositories, you’re probably better off with a more full-featured visualization tool like gitk or [Sourcetree](https://www.atlassian.com/software/sourcetree/overview) for projects that are heavily branched.

### Custom Formatting

For all of your other git log formatting needs, you can use the --pretty=format:"<string>" option. This lets you display each commit however you want using printf-style placeholders.

For example, the %cn, %h and %cd characters in the following command are replaced with the committer name, abbreviated commit hash, and the committer date, respectively.

git log --pretty=format:"%cn committed %h on %cd"

This results in the following format for each commit:

John committed 400e4b7 on Fri Jun 24 12:30:04 2014 -0500

John committed 89ab2cf on Thu Jun 23 17:09:42 2014 -0500

Mary committed 180e223 on Wed Jun 22 17:21:19 2014 -0500

John committed f12ca28 on Wed Jun 22 13:50:31 2014 -0500

The complete list of placeholders can be found in the [Pretty Formats](https://www.kernel.org/pub/software/scm/git/docs/git-log.html#_pretty_formats) section of the git log manual page.

Aside from letting you view only the information that you’re interested in, the --pretty=format:"<string>" option is particularly useful when you’re trying to pipe git log output into another command.

## Filtering the Commit History

Formatting how each commit gets displayed is only half the battle of learning git log. The other half is understanding how to navigate the commit history. The rest of this article introduces some of the advanced ways to pick out specific commits in your project history using git log. All of these can be combined with any of the formatting options discussed above.

### By Amount

The most basic filtering option for git log is to limit the number of commits that are displayed. When you’re only interested in the last few commits, this saves you the trouble of viewing all the commits in a page.

You can limit git log’s output by including the -<n> option. For example, the following command will display only the 3 most recent commits.

git log -3

### By Date

If you’re looking for a commit from a specific time frame, you can use the --after or --before flags for filtering commits by date. These both accept a variety of date formats as a parameter. For example, the following command only shows commits that were created after July 1st, 2014 (inclusive):

git log --after="2014-7-1"

You can also pass in relative references like "1 week ago" and "yesterday":

git log --after="yesterday"

To search for a commits that were created between two dates, you can provide both a --before and --after date. For instance, to display all the commits added between July 1st, 2014 and July 4th, 2014, you would use the following:

git log --after="2014-7-1" --before="2014-7-4"

Note that the --since and --until flags are synonymous with --after and --before, respectively.

### By Author

When you’re only looking for commits created by a particular user, use the --author flag. This accepts a regular expression, and returns all commits whose author matches that pattern. If you know exactly who you’re looking for, you can use a plain old string instead of a regular expression:

git log --author="John"

This displays all commits whose author includes the name John. The author name doesn’t need to be an exact match—it just needs to contain the specified phrase.

You can also use regular expressions to create more complex searches. For example, the following command searches for commits by either Mary or John.

git log --author="John\|Mary"

Note that the author’s email is also included with the author’s name, so you can use this option to search by email, too.

If your workflow separates committers from authors, the --committer flag operates in the same fashion.

### By Message

To filter commits by their commit message, use the --grep flag. This works just like the --author flag discussed above, but it matches against the commit message instead of the author.

For example, if your team includes relevant issue numbers in each commit message, you can use something like the following to pull out all of the commits related to that issue:

git log --grep="JRA-224:"

You can also pass in the -i parameter to git log to make it ignore case differences while pattern matching.

### By File

Many times, you’re only interested in changes that happened to a particular file. To show the history related to a file, all you have to do is pass in the file path. For example, the following returns all commits that affected either the foo.py or the bar.py file:

git log -- foo.py bar.py

The -- parameter is used to tell git log that subsequent arguments are file paths and not branch names. If there’s no chance of mixing it up with a branch, you can omit the --.

### By Content

It’s also possible to search for commits that introduce or remove a particular line of source code. This is called a pickaxe, and it takes the form of -S"<string>". For example, if you want to know when the string Hello, World! was added to any file in the project, you would use the following command:

git log -S"Hello, World!"

If you want to search using a regular expression instead of a string, you can use the -G"<regex>" flag instead.

This is a very powerful debugging tool, as it lets you locate all of the commits that affect a particular line of code. It can even show you when a line was copied or moved to another file.

### By Range

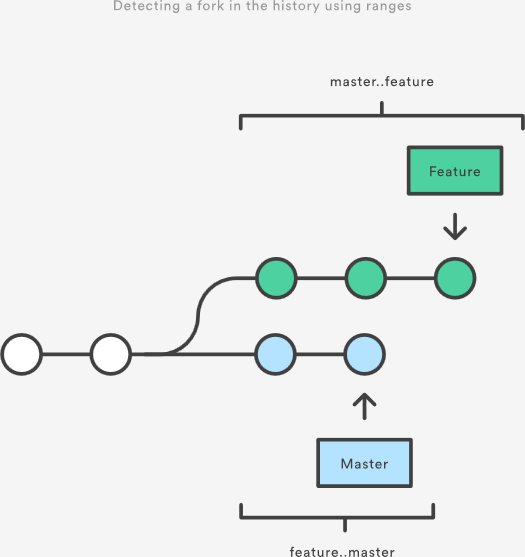
You can pass a range of commits to git log to show only the commits contained in that range. The range is specified in the following format, where <since> and <until> are commit references:

git log <since>..<until>

This command is particularly useful when you use branch references as the parameters. It’s a simple way to show the differences between 2 branches. Consider the following command:

git log master..feature

The master..feature range contains all of the commits that are in the feature branch, but aren’t in the master branch. In other words, this is how far feature has progressed since it forked off of master. You can visualize this as follows:



Note that if you switch the order of the range (feature..master), you will get all of the commits in master, but not in feature. If git log outputs commits for both versions, this tells you that your history has diverged.

### Filtering Merge Commits

By default, git log includes merge commits in its output. But, if your team has an always-merge policy (that is, you merge upstream changes into topic branches instead of rebasing the topic branch onto the upstream branch), you’ll have a lot of extraneous merge commits in your project history.

You can prevent git log from displaying these merge commits by passing the --no-merges flag:

git log --no-merges

On the other hand, if you’re only interested in the merge commits, you can use the --merges flag:

git log --merges

This returns all commits that have at least two parents.

## Summary

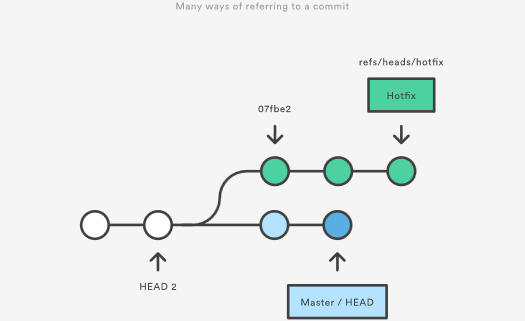
You should now be fairly comfortable using git log’s advanced parameters to format its output and select which commits you want to display. This gives you the power to pull out exactly what you need from your project history.

These new skills are an important part of your Git toolkit, but remember that git log is often used in conjunction other Git commands. Once you’ve found the commit you’re looking for, you typically pass it off to git checkout, git revert, or some other tool for manipulating your commit history. So, be sure to keep on learning about Git’s advanced features.

# Refs and the Reflog

[Hashes](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#hashes) [Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#refs) [Packed Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#packed-refs) [Special Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#special-refs) [Refspecs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#refspecs) [Relative Refs](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#relative-refs) [The Reflog](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#the-reflog) [Summary](https://www.atlassian.com/git/tutorials/refs-and-the-reflog#summary)

Git is all about commits: you stage commits, create commits, view old commits, and transfer commits between repositories using many different Git commands. The majority of these commands operate on a commit in some form or another, and many of them accept a commit reference as a parameter. For example, you can use git checkout to view an old commit by passing in a commit hash, or you can use it to switch branches by passing in a branch name.



By understanding the many ways to refer to a commit, you make all of these commands that much more powerful. In this chapter, we’ll shed some light on the internal workings of common commands like git checkout, git branch, and git push by exploring the many methods of referring to a commit.

We’ll also learn how to revive seemingly “lost” commits by accessing them through Git’s reflog mechanism.

## Hashes

The most direct way to reference a commit is via its SHA-1 hash. This acts as the unique ID for each commit. You can find the hash of all your commits in the git log output.

commit 0c708fdec272bc4446c6cabea4f0022c2b616eba

Author: Mary Johnson <mary@example.com>

Date: Wed Jul 9 16:37:42 2014 -0500

Some commit message

When passing the commit to other Git commands, you only need to specify enough characters to uniquely identify the commit. For example, you can inspect the above commit with git show by running the following command:

git show 0c708f

It’s sometimes necessary to resolve a branch, tag, or another indirect reference into the corresponding commit hash. For this, you can use the git rev-parse command. The following returns the hash of the commit pointed to by the master branch:

git rev-parse master

This is particularly useful when writing custom scripts that accept a commit reference. Instead of parsing the commit reference manually, you can let git rev-parse normalize the input for you.

## Refs

A **ref** is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git’s internal mechanism of representing branches and tags.

Refs are stored as normal text files in the .git/refs directory, where .git is usually called .git. To explore the refs in one of your repositories, navigate to .git/refs. You should see the following structure, but it will contain different files depending on what branches, tags, and remotes you have in your repo:

.git/refs/

heads/

master

some-feature

remotes/

origin/

master

tags/

v0.9

The heads directory defines all of the local branches in your repository. Each filename matches the name of the corresponding branch, and inside the file you’ll find a commit hash. This commit hash is the location of the tip of the branch. To verify this, try running the following two commands from the root of the Git repository:

# Output the contents of `refs/heads/master` file:

cat .git/refs/heads/master

# Inspect the commit at the tip of the `master` branch:

git log -1 master

The commit hash returned by the cat command should match the commit ID displayed by git log.

To change the location of the master branch, all Git has to do is change the contents of the refs/heads/master file. Similarly, creating a new branch is simply a matter of writing a commit hash to a new file. This is part of the reason why Git branches are so lightweight compared to SVN.

The tags directory works the exact same way, but it contains tags instead of branches. The remotes directory lists all remote repositories that you created with git remote as separate subdirectories. Inside each one, you’ll find all the remote branches that have been fetched into your repository.

### Specifying Refs

When passing a ref to a Git command, you can either define the full name of the ref, or use a short name and let Git search for a matching ref. You should already be familiar with short names for refs, as this is what you’re using each time you refer to a branch by name.

git show some-feature

The some-feature argument in the above command is actually a short name for the branch. Git resolves this to refs/heads/some-feature before using it. You can also specify the full ref on the command line, like so:

git show refs/heads/some-feature

This avoids any ambiguity regarding the location of the ref. This is necessary, for instance, if you had both a tag and a branch called some-feature. However, if you’re using proper naming conventions, ambiguity between tags and branches shouldn’t generally be a problem.

We’ll see more full ref names in the Refspecs section.

## Packed Refs

For large repositories, Git will periodically perform a garbage collection to remove unnecessary objects and compress refs into a single file for more efficient performance. You can force this compression with the garbage collection command:

git gc

This moves all of the individual branch and tag files in the refs folder into a single file called packed-refs located in the top of the .git directory. If you open up this file, you’ll find a mapping of commit hashes to refs:

00f54250cf4e549fdfcafe2cf9a2c90bc3800285 refs/heads/feature

0e25143693cfe9d5c2e83944bbaf6d3c4505eb17 refs/heads/master

bb883e4c91c870b5fed88fd36696e752fb6cf8e6 refs/tags/v0.9

On the outside, normal Git functionality won’t be affected in any way. But, if you’re wondering why your .git/refs folder is empty, this is where the refs went.

## Special Refs

In addition to the refs directory, there are a few special refs that reside in the top-level .git directory. They are listed below:

* HEAD – The currently checked-out commit/branch.
* FETCH\_HEAD – The most recently fetched branch from a remote repo.
* ORIG\_HEAD – A backup reference to HEAD before drastic changes to it.
* MERGE\_HEAD – The commit(s) that you’re merging into the current branch with git merge.
* CHERRY\_PICK\_HEAD – The commit that you’re cherry-picking.

These refs are all created and updated by Git when necessary. For example, The git pull command first runs git fetch, which updates the FETCH\_HEAD reference. Then, it runs git merge FETCH\_HEAD to finish pulling the fetched branches into the repository. Of course, you can use all of these like any other ref, as I’m sure you’ve done with HEAD.

These files contain different content depending on their type and the state of your repository. The HEAD ref can contain either a **symbolic ref**, which is simply a reference to another ref instead of a commit hash, or a commit hash. For example, take a look at the contents of HEAD when you’re on the master branch:

git checkout master

cat .git/HEAD

This will output ref: refs/heads/master, which means that HEAD points to the refs/heads/master ref. This is how Git knows that the master branch is currently checked out. If you were to switch to another branch, the contents of HEAD would be updated to reflect the new branch. But, if you were to check out a commit instead of a branch, HEAD would contain a commit hash instead of a symbolic ref. This is how Git knows that it’s in a detached HEAD state.

For the most part, HEAD is the only reference that you’ll be using directly. The others are generally only useful when writing lower-level scripts that need to hook into Git’s internal workings.

## Refspecs

A refspec maps a branch in the local repository to a branch in a remote repository. This makes it possible to manage remote branches using local Git commands and to configure some advanced git push and git fetch behavior.

A refspec is specified as [+]<src>:<dst>. The <src> parameter is the source branch in the local repository, and the <dst> parameter is the destination branch in the remote repository. The optional + sign is for forcing the remote repository to perform a non-fast-forward update.

Refspecs can be used with the git push command to give a different name to the remote branch. For example, the following command pushes the master branch to the origin remote repo like an ordinary git push, but it uses qa-master as the name for the branch in the origin repo. This is useful for QA teams that need to push their own branches to a remote repo.

git push origin master:refs/heads/qa-master

You can also use refspecs for deleting remote branches. This is a common situation for feature-branch workflows that push the feature branches to a remote repo (e.g., for backup purposes). The remote feature branches still reside in the remote repo after they are deleted from the local repo, so you get a build-up of dead feature branches as your project progresses. You can delete them by pushing a refspec that has an empty <src> parameter, like so:

git push origin :some-feature

This is very convenient, since you don’t need to log in to your remote repository and manually delete the remote branch. Note that as of Git v1.7.0 you can use the --delete flag instead of the above method. The following will have the same effect as the above command:

git push origin --delete some-feature

By adding a few lines to the Git configuration file, you can use refspecs to alter the behavior of git fetch. By default, git fetch fetches all of the branches in the remote repository. The reason for this is the following section of the .git/config file:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

The fetch line tells git fetch to download all of the branches from the origin repo. But, some workflows don’t need all of them. For example, many continuous integration workflows only care about the master branch. To fetch only the master branch, change the fetch line to match the following:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/master:refs/remotes/origin/master

You can also configure git push in a similar manner. For instance, if you want to always push the master branch to qa-master in the origin remote (as we did above), you would change the config file to:

[remote "origin"]

url = https://git@github.com:mary/example-repo.git

fetch = +refs/heads/master:refs/remotes/origin/master

push = refs/heads/master:refs/heads/qa-master

Refspecs give you complete control over how various Git commands transfer branches between repositories. They let you rename and delete branches from your local repository, fetch/push to branches with different names, and configure git push and git fetch to work with only the branches that you want.

## Relative Refs

You can also refer to commits relative to another commit. The ~ character lets you reach parent commits. For example, the following displays the grandparent of HEAD:

git show HEAD~2

But, when working with merge commits, things get a little more complicated. Since merge commits have more than one parent, there is more than one path that you can follow. For 3-way merges, the first parent is from the branch that you were on when you performed the merge, and the second parent is from the branch that you passed to the git merge command.

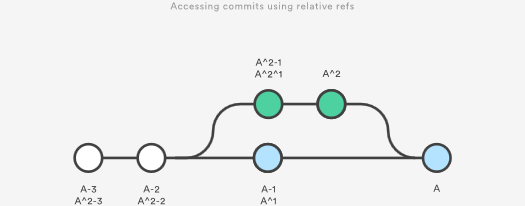
The ~ character will always follow the first parent of a merge commit. If you want to follow a different parent, you need to specify which one with the ^ character. For example, if HEAD is a merge commit, the following returns the second parent of HEAD.

git show HEAD^2

You can use more than one ^ character to move more than one generation. For instance, this displays the grandparent of HEAD (assuming it’s a merge commit) that rests on the second parent.

git show HEAD^2^1

To clarify how ~ and ^ work, the following figure shows you how to reach any commit from A using relative references. In some cases, there are multiple ways to reach a commit.



Relative refs can be used with the same commands that a normal ref can be used. For example, all of the following commands use a relative reference:

# Only list commits that are parent of the second parent of a merge commit

git log HEAD^2

# Remove the last 3 commits from the current branch

git reset HEAD~3

# Interactively rebase the last 3 commits on the current branch

git rebase -i HEAD~3

## The Reflog

The reflog is Git’s safety net. It records almost every change you make in your repository, regardless of whether you committed a snapshot or not. You can think of it as a chronological history of everything you’ve done in your local repo. To view the reflog, run the git reflog command. It should output something that looks like the following:

400e4b7 HEAD@{0}: checkout: moving from master to HEAD~2

0e25143 HEAD@{1}: commit (amend): Integrate some awesome feature into `master`

00f5425 HEAD@{2}: commit (merge): Merge branch ';feature';

ad8621a HEAD@{3}: commit: Finish the feature

This can be translated as follows:

* You just checked out HEAD~2
* Before that you amended a commit message
* Before that you merged the feature branch into master
* Before that you committed a snapshot

The HEAD{<n>} syntax lets you reference commits stored in the reflog. It works a lot like the HEAD~<n> references from the previous section, but the <n> refers to an entry in the reflog instead of the commit history.

You can use this to revert to a state that would otherwise be lost. For example, lets say you just scrapped a new feature with git reset. Your reflog might look something like this:

ad8621a HEAD@{0}: reset: moving to HEAD~3

298eb9f HEAD@{1}: commit: Some other commit message

bbe9012 HEAD@{2}: commit: Continue the feature

9cb79fa HEAD@{3}: commit: Start a new feature

The three commits before the git reset are now dangling, which means that there is no way to reference them—except through the reflog. Now, let’s say you realize that you shouldn’t have thrown away all of your work. All you have to do is check out the HEAD@{1} commit to get back to the state of your repository before you ran git reset.

git checkout HEAD@{1}

This puts you in a detached HEAD state. From here, you can create a new branch and continue working on your feature.

## Summary

You should now be quite comfortable referring to commits in a Git repository. We learned how branches and tags were stored as refs in the .git subdirectory, how to read a packed-refs file, how HEAD is represented, how to use refspecs for advanced pushing and fetching, and how to use the relative ~ and ^ operators to traverse a branch hierarchy.

We also took a look at the reflog, which is a way to reference commits that are not available through any other means. This is a great way to recover from those little “Oops, I shouldn’t have done that” situations.

The point of all this was to be able to pick out exactly the commit that you need in any given development scenario. It’s very easy to leverage the skills you learned in this article against your existing Git knowledge, as some of the most common commands accept refs as arguments, including git log, git show, git checkout, git reset, git revert, git rebase, and many others.

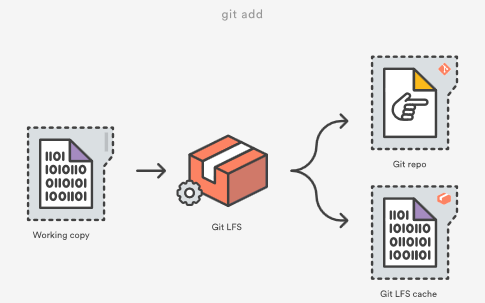
# Git LFS

### What is Git LFS?

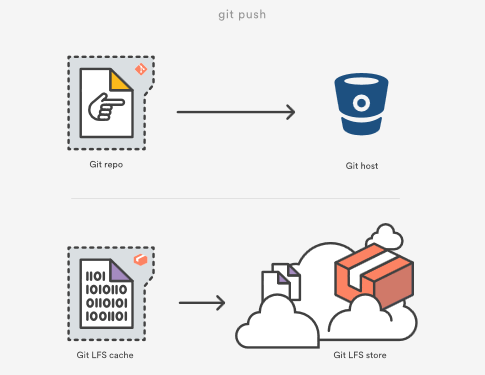
Git is a distributed version control system, meaning the entire history of the repository is transferred to the client during the cloning process. For projects containing large files, particularly large files that are modified regularly, this initial clone can take a huge amount of time, as every version of every file has to be downloaded by the client. Git LFS (Large File Storage) is a Git extension developed by Atlassian, GitHub, and a few other open source contributors, that reduces the impact of large files in your repository by downloading the relevant versions of them lazily. Specifically, large files are downloaded during the checkout process rather than during cloning or fetching.

Git LFS does this by replacing large files in your repository with tiny pointer files. During normal usage, you'll never see these pointer files as they are handled automatically by Git LFS:

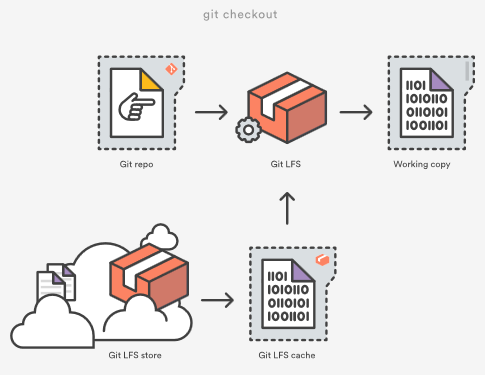
1. When you add a file to your repository, Git LFS replaces its contents with a pointer, and stores the file contents in a local Git LFS cache.



When you push new commits to the server, any Git LFS files referenced by the newly pushed commits are transferred from your local Git LFS cache to the remote Git LFS store tied to your Git repository.



When you checkout a commit that contains Git LFS pointers, they are replaced with files from your local Git LFS cache, or downloaded from the remote Git LFS store.



Git LFS is seamless: in your working copy you'll only see your actual file content. This means you can use Git LFS without changing your existing Git workflow; you simply git checkout, edit, git add, and git commit as normal. git clone and git pull operations will be significantly faster as you only download the versions of large files referenced by commits that you actually check out, rather than every version of the file that ever existed.

### racking files with Git LFS

When you add a new type of large file to your repository, you'll need to tell Git LFS to track it by specifying a pattern using the git lfs track command:

$ git lfs track "\*.ogg"

Tracking \*.ogg

Note that the quotes around "\*.ogg" are important. Omitting them will cause the wildcard to be expanded by your shell, and individual entries will be created for each .ogg file in your current directory:

# probably not what you want

$ git lfs track \*.ogg

Tracking explode.ogg

Tracking music.ogg

Tracking phaser.ogg

The patterns supported by Git LFS are the same as those supported by [.gitignore](https://www.atlassian.com/git/tutorials/gitignore), for example:

# track all .ogg files in any directory

$ git lfs track "\*.ogg"

# track files named music.ogg in any directory

$ git lfs track "music.ogg"

# track all files in the Assets directory and all subdirectories

$ git lfs track "Assets/"

# track all files in the Assets directory but \*not\* subdirectories

$ git lfs track "Assets/\*"

# track all ogg files in Assets/Audio

$ git lfs track "Assets/Audio/\*.ogg"

# track all ogg files in any directory named Music

$ git lfs track "\*\*/Music/\*.ogg"

# track png files containing "xxhdpi" in their name, in any directory

$ git lfs track "\*xxhdpi\*.png

After running git lfs track, you'll notice a new file named .gitattributes in the directory you ran the command from. .gitattributes is a Git mechanism for binding special behaviors to certain file patterns. Git LFS automatically creates or updates .gitattributes files to bind tracked file patterns to the Git LFS filter. However, you will need to commit any changes to the .gitattributes file to your repository yourself:

You can stop tracking a particular pattern with Git LFS by simply removing the appropriate line from your .gitattributes file, or by running the git lfs untrack command:

$ git lfs untrack "\*.ogg"

Untracking \*.ogg

After running git lfs untrack you will again have to commit the changes to .gitattributes yourself.

# Git gc

The git gc command is a repository maintenance command. The "gc" stands for garbage collection. Executing git gc is literally telling Git to clean up the mess it's made in the current repository. Garbage collection is a concept that originates from interpreted programming languages which do dynamic memory allocation. Garbage collection in interpreted languages is used to recover memory that has become inaccessible to the executing program.

Git repositories accumulate various types of garbage. One type of Git garbage is orphaned or inaccessible commits. Git commits can become inaccessible when performing history altering commands like [git resets](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) or [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase). In an effort to preserve history and avoid data loss Git will not delete detached commits. A detached commit can still be checked out, cherry picked, and examined through the git log.

In addition to detached commit clean up, git gc will also perform compression on stored Git Objects, freeing up precious disk space. When Git identifies a group of similar objects it will compress them into a 'pack'. Packs are like zip files of Git bjects and live in the ./git/objects/pack directory within a repository.

## What does git gc actually do?

Before execution, git gc first checks several [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) values. These values will help clarify the rest of git gc responsibility.

### git gc configuration

gc.reflogExpire

An optional variable that defaults to 90 days. It is used to set how long records in a branches reflog should be preserved.

gc.reflogExpireUnreachable

An optional variable that defaults to 30 days. It is used to set how long inaccessible reflog records should be preserved.

gc.aggressiveWindow

An optional variable that defaults to 250. It controls how much time is spent in the delta compression phase of object packing when git gc is executed with the --aggressive option.

gc.aggressiveDepth

Optional variable that defaults to 50. It controls the depth of compression git-repack uses during a git gc --aggresive execution

gc.pruneExpire

Optional variable that defaults to "2 weeks ago". It sets how long a inaccessible object will be preserved before pruning

gc.worktreePruneExpire

Optional variable that defaults to "3 months ago". It sets how long a stale working tree will be preserved before being deleted.

### git gc execution

Behind the scenes git gc actually executes a bundle of other internal subcommands like [git prune](https://www.atlassian.com/git/tutorials/git-prune), git repack, git pack and git rerere. The high-level responsibility of these commands is to identify any Git objects that are outside the threshold levels set from the git gc configuration. Once identified, these objects are then compressed, or pruned accordingly.

## git gc best practices and FAQS

Garbage collection is run automatically on several frequently used commands:

* [git pull](https://www.atlassian.com/git/tutorials/making-a-pull-request)
* [git merge](https://www.atlassian.com/git/tutorials/git-merge)
* [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)
* [git commit](https://www.atlassian.com/git/tutorials/saving-changes)

The frequency in which git gc should be manually executed corresponds to the activity level of a repository. A repository with a single contributing developer will need to execute git gc far less often than a frequently-updated multi-user repository.

## git gc vs git prune

git gc is a parent command and git prune is a child. git gc will internally trigger git prune. git prune is used to remove Git objects that have been deemed inaccessible by the git gc configuration. Learn more about [git prune](https://www.atlassian.com/git/tutorials/git-prune).

## What is git gc aggressive?

git gc can be invoked with the --aggressive command line option. The --aggressive option causes git gc to spend more time on its optimization effort. This causes git gc to run slower but will save more disk space after its completion. The effects of --aggressive are persistent and only need to be run after a large volume of changes to a repository.

## What is git gc auto?

The git gc --auto command variant first checks if any housekeeping is required on the repo before executing. If it finds housekeeping is not needed it exits without doing any work. Some Git commands implicitly run git gc --auto after execution to clean up any loose objects they have created.

Before execution git gc --auto will check the git configuration for threshold values on loose objects and packing compression size. These values can be set with [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config). If the repository surpasses any of the housekeeping thresholds git gc --auto will be executed.

## Getting started with git gc

You're probably already using git gc without noticing. As discussed in the best practices section, it is automatically invoked through frequently used commands. If you want to manually invoke it simply execute git gc and you should see an output indicating the work it has performed.

# Git Prune

The git prune command is an internal housekeeping utility that cleans up unreachable or "orphaned" Git objects. Unreachable objects are those that are inaccessible by any refs. Any commit that cannot be accessed through a branch or tag is considered unreachable. git prune is generally not executed directly. Prune is considered a garbage collection command and is a child command of the [git gc](https://www.atlassian.com/git/tutorials/git-gc) command.

## Git Prune Overview

In order to understand the effects of git prune we need to simulate a scenario where a commit becomes unreachable. The following is a sequence of command line executions that will simulate this experience.

~ $ cd git-prune-demo/  
~/git-prune-demo $ git init .  
Initialized empty Git repository in /Users/kev/Dropbox/git-prune-demo/.git/  
~/git-prune-demo $ echo "hello git prune" > hello.txt  
~/git-prune-demo $ git add hello.txt  
~/git-prune-demo $ git commit -am "added hello.txt"

The preceding sequence of commands will create a new repository in a directory named git-prune-demo. One commit consisting of a new file hello.text is added to the repo with the basic content of "hello git prune". Next, we will create modify hello.txt and create a new commit from those modifications.

~/git-prune-demo $ echo "this is second line txt" >> hello.txt  
~/git-prune-demo $ cat hello.txt  
hello git prune  
this is second line txt  
~/git-prune-demo $ git commit -am "added another line to hello.txt"  
[master 5178bec] added another line to hello.txt  
1 file changed, 1 insertion(+)

We now have a 2 commit history in this demo repo. We can verify by using git log:

~/git-prune-demo $ git log  
commit 5178becc2ca965e1728554ce1cb8de2f2c2370b1  
Author: kevzettler <kevzettler@gmail.com>  
Date:   Sun Sep 30 14:49:59 2018 -0700  
  
        added another line to hello.txt  
  
commit 994b122045cf4bf0b97139231b4dd52ea2643c7e  
Author: kevzettler <kevzettler@gmail.com>  
Date:   Sun Sep 30 09:43:41 2018 -0700  
  
        added hello.txt

The [git log](https://www.atlassian.com/git/tutorials/git-log) output displays the 2 commits and corresponding commit messages about the edits made to hello.txt. The next step is for us to make one of the commits unreachable. We will do this by utilizing the [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command. We reset the state of the repo to the first commit. the "added hello.txt" commit.

~/git-prune-demo $ git reset --hard 994b122045cf4bf0b97139231b4dd52ea2643c7e  
HEAD is now at 994b122 added hello.txt

If we now use git log to examine the state of the repository we can see that we only have one commit

~/git-prune-demo $ git log  
commit 994b122045cf4bf0b97139231b4dd52ea2643c7e  
Author: kevzettler <kevzettler@gmail.com>  
Date:   Sun Sep 30 09:43:41 2018 -0700  
  
        added hello.txt

The demo repository is now in a state that contains a detached commit. The second commit we made with the message "added another line to hello.txt" is no longer displayed in the git log output and is now detached. It may appear as though we have lost or deleted the commit, but Git is very strict about not deleting history. We can confirm it is still available, but detached, by using [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) to visit it directly:

~/git-prune-demo $ git checkout 5178becc2ca965e1728554ce1cb8de2f2c2370b1  
Note: checking out '5178becc2ca965e1728554ce1cb8de2f2c2370b1'.  
  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by performing another checkout.  
  
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:  
  
      git checkout -b <new-branch-name>  
  
HEAD is now at 5178bec... added another line to hello.txt  
~/git-prune-demo $ git log  
commit 5178becc2ca965e1728554ce1cb8de2f2c2370b1  
Author: kevzettler <kevzettler@gmail.com>  
Date:   Sun Sep 30 14:49:59 2018 -0700  
  
      added another line to hello.txt  
  
commit 994b122045cf4bf0b97139231b4dd52ea2643c7e  
Author: kevzettler <kevzettler@gmail.com>  
Date:   Sun Sep 30 09:43:41 2018 -0700  
  
      added hello.txt

When we check out the detached commit, Git is thoughtful enough to give us a detailed message explaining that we are in a detached state. If we examine the log here we can see that the "added another line to hello.txt" commit is now back in the log output! Now that we know the repository is in a good simulation state with a detached commit we can practice using git prune. First though, let us return to the master branch using git checkout

~/git-prune-demo $ git checkout master  
Warning: you are leaving 1 commit behind, not connected to  
any of your branches:  
  
      5178bec added another line to hello.txt  
  
If you want to keep it by creating a new branch, this may be a good time  
to do so with:  
  
     git branch <new-branch-name> 5178bec  
  
Switched to branch 'master'

When returning to master via git checkout, Git is again thoughtful enough to let us know that we are leaving a detached commit behind. It's now time to prune the detached commit! Next, we will execute git prune but we must be sure to pass some options to it. --dry-run and --verbose will display output indicating what is set to be pruned but not actually prune it.

~/git-prune-demo $ git prune --dry-run --verbose

This command will most likely return empty output. Empty output implies that the prune will not actually delete anything. Why would this happen? Well, the commit is most likely not fully detached. Somewhere Git is still maintaining a reference to it. This is a prime example of why git prune is not to be used stand-alone outside of git gc. This is also a good example of how it is hard to fully lose data with Git.

Most likely Git is storing a reference to our detached commit in the reflog. We can investigate by running [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog). You should see some output describing the sequence of actions we took to get here. For more info on git reflog visit the [git reflog](https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog) page. In addition to preserving history in the reflog, Git has internal expiration dates on when it will prune detached commits. Again, these are all implementation details that git gc handles and git prune should not be used standalone.

To conclude our git prune simulation demo we must clear the reflog

~/git-prune-demo $ git reflog expire --expire=now --expire-unreachable=now --all

The above command will force expire all entries to the reflog that are older than now. This is a brutal and dangerous command that you should never have to use as casual Git user. We are executing this command to demonstrate a successful git prune. With the reflog totally wiped we can now execute git prune.

~/git-prune-demo $ git prune --dry-run --verbose --expire=now  
1782293bdfac16b5408420c5cb0c9a22ddbdd985 blob  
5178becc2ca965e1728554ce1cb8de2f2c2370b1 commit  
a1b3b83440d2aa956ad6482535cbd121510a3280 commit  
f91c3433eae245767b9cd5bdb46cd127ed38df26 tree

This command should output a list of Git SHA object references that looks like the above.

## Usage

git prune has a short list of options that we covered in the overview section.

-n --dry-run

Don't execute the prune. Just show an output of what it will do

-v --verbose

Display output of all objects and actions taken by the prune

--progress

Displays output that indicates the progress of the prune

--expire <time>

Force expiration of objects that are past <time>

<head>…

Specifying a <head> will preserve any options from that head ref

## Discussion

### What’s the Difference Between Git Prune, Git Fetch --prune, and Git Remote Prune?

git remote prune and git fetch --prune do the same thing: delete the refs to branches that don't exist on the remote. This is highly desirable when working in a team workflow in which remote branches are deleted after merge to master. The second command, git fetch --prune will connect to the remote and fetch the latest remote state before pruning. It is essentially a combination of commands:

git fetch --all && git remote prune

The generic git prune command is entirely different. As discussed in the overview section, git prune will delete locally detached commits.

### How Do I Clean Outdated Branches?

git fetch --prune is the best utility for cleaning outdated branches. It will connect to a shared remote repository remote and fetch all remote branch refs. It will then delete remote refs that are no longer in use on the remote repository.

### Does Git Remote Prune Origin Delete the Local Branch?

No git remote prune origin will only delete the refs to remote branches that no longer exist. Git stores both local and remote refs. A repository will have local/origin and remote/origin ref collections. git remote prune origin will only prune the refs in remote/origin. This safely leaves local work in local/origin.

## Git Prune Summary

The git prune command is intended to be invoked as a child command to git gc. It is highly unlikely you will ever need to invoke git prune in a day to day software engineering capacity. Other commands are needed to understand the effects of git prune. Some commands used in this article were git log, git reflog, and git checkout.

Artilces

# Simple Git workflow is simple



###### Nicola Paolucci

[Back to list](https://www.atlassian.com/git/articles)

Many teams have already migrated to git and many more are transitioning to it now. Apart from training single developers and appointing **Champions** to help with the adoption it is imperative to pick a nice and simple code collaboration practice that does not complicate things too much. With git one can definitely conjure very complicated workflows, I've seen them first hand.

A manual on workflows does not come pre-installed with git, but maybe it should seeing how many people have questions on the topic. The good news is that we're working hard to write material that helps.

## Recent webinars and guides on workflows

* [Git Branching for Agile Teams](https://www.youtube.com/watch?v=9SZ7kSQ2424)
* [Git Ready: Workflows Webinar, Atlassian, September 2013](https://www.youtube.com/watch?v=B78AdLNZBQQ)

If you prefer reading and pretty pictures, one of the most popular sections of our git tutorial site is [the workflows section](https://www.atlassian.com/git/tutorials/comparing-workflows).

But before you leave for those destinations please read on, because I have something really cool for you.

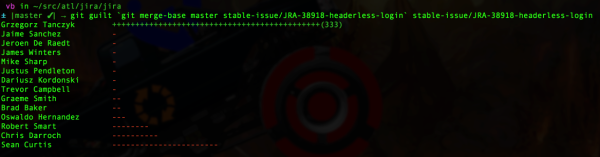
I want to detail a terse but complete description of a simple workflow for continuous delivery. The prerequisite is that you and your team are at least a little bit acquainted with git, and have good knowledge of the rebase command in the two forms (interactive and not).

## A basic basic branching workflow for continuous delivery

The simple workflow I want to describe has two guiding principles:

* master is always production-like and deployable.
* rebase during feature development, explicit (non fast-forward) merge when done.

Pulling change-sets using rebase rewrites the history of the branch you're working on and keeps your changes on top.



The rebase you want in this workflow is the one in the second picture.

Armed with these guiding principles let's breakdown the seven steps:

### 1. Start by pulling down the latest changes from master

This is done easily with the common git commands:

git checkout master

git fetch origin

git merge master

I like to be more explicit and use fetch/merge but the two commands are equivalent to: git pull origin master.

### 2. Branch off to isolate the feature or bug-fix work in a branch

Now create a branch for the feature or bug-fix:

git checkout -b PRJ-123-awesome-feature

The branch name structure I show here is just the one we use, but you can pick any convention you feel comfortable with.

### 3. Now you can work on the feature

Work on the feature as long as needed. Make sure your commits are meaningful and do not cluster separate changes together.

### 4. To keep your feature branch fresh and up to date with the latest changes in master, use rebase

Every once in a while during the development update the feature branch with the latest changes in master. You can do this with:

git fetch origin

git rebase origin/master

In the (somewhat less common) case where other people are also working on the same shared remote feature branch, also rebase changes coming from it:

git rebase origin/PRJ-123-awesome-feature

At this point solve any conflicts that come out of the rebase.

Resolving conflicts during the rebase allows you to have always clean merges at the end of the feature development. It also keeps your feature branch history clean and focused without spurious noise.

### 5. When ready for feedback push your branch remotely and create a pull request

When it's time to share your work and solicit feedback you can push your branch remotely with:

git push -u origin PRJ-123-awesome-feature

(if the branch is already set as 'upstream' and your remote is called 'origin', 'git push' is enough)

Now you can create a pull request on your favorite git server (for example [Bitbucket Server](https://www.atlassian.com/software/bitbucket/server) or [Bitbucket Cloud](https://bitbucket.org)).

After the initial push you can keep pushing updates to the remote branch multiple times throughout. This can happen in response to feedback, or because you're not done with the development of the feature.

### 6. Perform a final rebase cleanup after the pull request has been approved

After the review is done, it's good to perform a final cleanup and scrub of the feature branch commit history to remove spurious commits that are not providing relevant information. In some cases – if your team is experienced and they can handle it – you can rebase also during development, but I **strongly** advise against it.:

git rebase -i origin/master

(At this point if you have rewritten the history of a published branch and provided that no one else will commit to it or use it, you might need to push your changes using the --force flag).

### 7. When development is complete record an explicit merge

When finished with the development of the feature branch and reviewers have reviewed your work, merge using the flag --no-ff. This will preserve the context of the work and will make it easy to revert the whole feature if needed. Here are the commands:

git checkout master

git pull origin master

git merge --no-ff PRJ-123-awesome-feature

If you followed the advice above and you have used rebase to keep your feature branch up to date, the actual merge commit will not include any changes; this is cool! The merge commit becomes just a marker that stores the context about the feature branch.

For more information have a look at my recent article on [the pros and cons of enforcing a merge vs rebase workflow](https://blogs.atlassian.com/2013/10/git-team-workflows-merge-or-rebase/).

### Useful .gitconfig option to toggle:

You can instruct git so that any pull uses rebase instead than merge and it preserves while doing so:

git config --global branch.autosetuprebase always

git config --global pull.rebase preserve #(this is a very recent and useful addition that appeared in git 1.8.5)

Not everyone likes to change the default behavior of core commands so you should only incorporate the above if you understand its implications. See [Stack Overflow for details on preserve merges](http://stackoverflow.com/questions/15915430/what-exactly-does-gits-rebase-preserve-merges-do-and-why).

## Conclusions

This should give you plenty of material to get acquainted with workflows, branching models and code collaboration possibilities. For more git rocking follow me [@durdn](http://twitter.com/durdn) and the awesome [@AtlDevtools](http://twitter.com/AtlDevTools) team. **Credits:** Inspiration for this post comes partially from this [concise and well made gist](https://gist.github.com/jbenet/ee6c9ac48068889b0912).

# Titanium Armor: Recovering From Various Disasters



###### Nicola Paolucci

[Back to list](https://www.atlassian.com/git/articles)

[git](https://www.atlassian.com/git/tutorials/what-is-git) is an advanced tool. It features a philosophy that is dear to my heart: to treat developers as smart and responsible folks. This means that a lot of power is at your fingertips. The power to also shoot yourself in the foot - arguably with a titanium vest on - but shoot yourself nonetheless.

The topic of this post is to confirm what my colleague [Charles O'Farrell](https://twitter.com/charlesofarrell) said very well in the ever popular "[Why Git?](http://blogs.atlassian.com/2012/03/git-vs-mercurial-why-git/?_ga=2.6845883.889086498.1548178177-1094429246.1546445841)" article some time ago:

[...] Git is actually the safest of all the DVCS options. As we saw above, Git never actually lets you change anything, it just creates new objects.

[...] Git actually keeps track of every change you make, storing them in the reflog. Because every commit is unique and immutable, all the reflog has to do is store a reference to them. This means you're safe from harm and your code is always preserved, but there are situations where you might need some conjuring to get it back.

Let me give you a few real world examples of how to recover from trouble, going from simple to advanced:

## How To Undo reset --hard If You Only Staged Your Changes

I'll save you the speech to commit frequently - whether in a private short lived branch or a shared one - to protect you from heaps of problems.

But let's say you didn't commit this time and you find yourself in the following scenario: you've done some work, haven't committed it yet but have git added it to the staging area.

Then tragedy strikes: you type git reset --hard and immediately realize that you've zeroed your local changes(!!) and brought back (in it's entirety) the previous commit.

This is a tough situation to be in. How do you recover your work? Is it even possible? The answer is yes! The solution involves some low level searching and can be approached in a couple of ways.

Since git creates new objects in the .git/objects folder as soon as something is added to the staging area we can look there for the most recently created objects:

In OSX (and BSD systems) you can do:

find .git/objects/ -type f | xargs stat -f "%Sm %N" | sort | tail -5

On Linux(with GNU tools) this should work:

find .git/objects/ -type f -printf '%TY-%Tm-%Td %TT %p\n' | sort | tail -5

With tail -n you can cut to the last n results if your repository is very big:

The result should be something like:

Apr 2 12:26:33 2013 .git/objects//pack/pack-4cf657357973915f6fb6f90a41f69a88c0b08bb7.pack

Apr 2 12:29:56 2013 .git/objects//3d/d6845a69cd59dac0851e1ae0bd69dde950c46b

Apr 2 12:29:56 2013 .git/objects//68/c983f0cefb4bee2f753516ab6352ee2f2b7d29

Apr 2 12:35:23 2013 .git/objects//03/2c6a653cc00c302020293e020d8d68326b112e

Apr 2 15:34:55 2013 .git/objects//df/485610889e98ff773b4440a032ea2ede1338b9

In my case my sample file was lost exactly around 15:34 so I can inspect and retrieve it with:

git show df485610889e98ff773b4440a032ea2ede1338b9

Remember that the folder is part of the sha-1 reference, so remove the / slash to compute the correct id.

Note that git has specific low level command to explore dangling objects and to verify their connectivity: [git fsck](https://www.kernel.org/pub/software/scm/git/docs/git-fsck.html). For an alternative solution to this problem using fsck, check out this very cool [answer on Stack Overflow](http://stackoverflow.com/questions/7374069/undo-git-reset-hard/7376959#7376959).

## Conclusions

There are many more examples and scenarios to cover on this topic so I might come back later with more interesting cases.

One thing to take away from this should be the feeling that it's very very hard to screw up and lose your data when using git. That's all for now!

Follow the awesome [@Bitbucket](https://twitter.com/Bitbucket) team or me [@durdn](http://twitter.com/durdn) for more Git rocking.

## Advanced Branching And Merging

### Getting conflict-resolution help during a merge

All of the changes that git was able to merge automatically are already added to the index file, so git-diff shows only the conflicts.

Recall that the commit which will be committed after we resolve this conflict will have two parents instead of the usual one: one parent will be HEAD, the tip of the current branch; the other will be the tip of the other branch, which is stored temporarily in MERGE\_HEAD.

During the merge, the index holds three versions of each file. Each of these three "file stages" represents a different version of the file:

$ git show :1:file.txt # the file in a common ancestor of both branches

$ git show :2:file.txt # the version from HEAD.

$ git show :3:file.txt # the version from MERGE\_HEAD.

When you ask linkgit:git-diff[1] to show the conflicts, it runs a three-way diff between the conflicted merge results in the work tree with stages 2 and 3 to show only hunks whose contents come from both sides, mixed (in other words, when a hunk's merge results come only from stage 2, that part is not conflicting and is not shown. Same for stage 3).

Some special diff options allow diffing the working directory against any of these stages:

$ git diff -1 file.txt # diff against stage 1

$ git diff --base file.txt # same as the above

$ git diff -2 file.txt # diff against stage 2

$ git diff --ours file.txt # same as the above

$ git diff -3 file.txt # diff against stage 3

$ git diff --theirs file.txt # same as the above.

## Finding Issues - Git Blame

The git-blame command is really helpful for figuring out who changed which sections of a file. If you simple run 'git blame [filename]' you'll get an output of the entire file with the last commit sha, date and author for every line in the file.

$> git blame sha1\_file.c

This is often helpful if a file had a line reverted or a mistake that broke the build to help you see who changed that line last

$> git blame -L 160,+10 sha1\_file.c

## SCM Migration

So you've made the decision to move away from your existing system and convert your whole project to Git. How can you do that easily?

### Importing Subversion

Git comes with a script called git-svn that has a clone command that will import a subversion repository into a new git repository. There is also a free tool on the GitHub service that will do this for you.

$ git-svn clone http://my-project.googlecode.com/svn/trunk new-project

This will give you a new Git repository with all the history of the original Subversion repo. This takes a pretty good amount of time, generally, since it starts with version 1 and checks out and commits locally every single revision one by one.

## Submodules

To see how submodule support works, create (for example) four example repositories that can be used later as a submodule:

$ mkdir ~/git

$ cd ~/git

$ for i in a b c d

do

mkdir $i

cd $i

git init

echo "module $i" > $i.txt

git add $i.txt

git commit -m "Initial commit, submodule $i"

cd ..

done

Now create the superproject and add all the submodules:

$ mkdir super

$ cd super

$ git init

$ for i in a b c d

do

git submodule add ~/git/$i $i

done

See what files git-submodule created:

$ ls -a

. .. .git .gitmodules a b c d

The git-submodule add command does a couple of things:

* It clones the submodule under the current directory and by default checks out the master branch.
* It adds the submodule's clone path to the linkgit:gitmodules[5] file and adds this file to the index, ready to be committed.
* It adds the submodule's current commit ID to the index, ready to be committed.

Commit the superproject:

$ git commit -m "Add submodules a, b, c and d."

Now clone the superproject:

$ cd ..

$ git clone super cloned

$ cd cloned

The submodule directories are there, but they're empty:

Pulling down the submodules is a two-step process. First run git submodule init to add the submodule repository URLs to .git/config:

$ git submodule init

Now use git-submodule update to clone the repositories and check out the commits specified in the superproject:

$ git submodule update