Links

<https://www.alexkras.com/category/git/>

<https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting>

# What is version control

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members

Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Version Control Systems (VCS) have seen great improvements over the past few decades and some are better than others. VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System). One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS, more on that later. Like many of the most popular VCS systems available today, Git is free and open source. Regardless of what they are called, or which system is used, the primary benefits you should expect from version control are as follows.

1. A complete long-term change history of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Different VCS tools differ on how well they handle renaming and moving of files. This history should also include the author, date and written notes on the purpose of each change. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software. If the software is being actively worked on, almost everything can be considered an "older version" of the software.
2. Branching and merging. Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes. Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict. Many software teams adopt a practice of branching for each feature or perhaps branching for each release, or both. There are many different workflows that teams can choose from when they decide how to make use of branching and merging facilities in VCS.
3. Traceability. Being able to trace each change made to the software and connect it to project management and bug tracking software such as [Jira](https://www.atlassian.com/software/jira), and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics. Having the annotated history of the code at your fingertips when you are reading the code, trying to understand what it is doing and why it is so designed can enable developers to make correct and harmonious changes that are in accord with the intended long-term design of the system. This can be especially important for working effectively with legacy code and is crucial in enabling developers to estimate future work with any accuracy.

# What is Git

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

In addition to being distributed, Git has been designed with performance, security and flexibility in mind.

## Performance

The raw performance characteristics of Git are very strong when compared to many alternatives. Committing new changes, branching, merging and comparing past versions are all optimized for performance. The algorithms implemented inside Git take advantage of deep knowledge about common attributes of real source code file trees, how they are usually modified over time and what the access patterns are.

Unlike some version control software, Git is not fooled by the names of the files when determining what the storage and version history of the file tree should be, instead, Git focuses on the file content itself. After all, source code files are frequently renamed, split, and rearranged. The object format of Git's repository files uses a combination of delta encoding (storing content differences), compression and explicitly stores directory contents and version metadata objects.

Being distributed enables significant performance benefits as well.

For example, say a developer, Alice, makes changes to source code, adding a feature for the upcoming 2.0 release, then commits those changes with descriptive messages. She then works on a second feature and commits those changes too. Naturally these are stored as separate pieces of work in the version history. Alice then switches to the version 1.3 branch of the same software to fix a bug that affects only that older version. The purpose of this is to enable Alice's team to ship a bug fix release, version 1.3.1, before version 2.0 is ready. Alice can then return to the 2.0 branch to continue working on new features for 2.0 and all of this can occur without any network access and is therefore fast and reliable. She could even do it on an airplane. When she is ready to send all of the individually committed changes to the remote repository, Alice can "push" them in one command.

## Security

Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

With Git, you can be sure you have an authentic content history of your source code.

Some other version control systems have no protections against secret alteration at a later date. This can be a serious information security vulnerability for any organization that relies on software development.

## Flexibility

One of Git's key design objectives is flexibility. Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

Git has been designed to support branching and tagging as first-class citizens (unlike SVN) and operations that affect branches and tags (such as merging or reverting) are also stored as part of the change history. Not all version control systems feature this level of tracking.

## Version control with Git

Git is the best choice for most software teams today. While every team is different and should do their own analysis, here are the main reasons why version control with Git is preferred over alternatives:

### **Git is good**

Git has the functionality, performance, security and flexibility that most teams and individual developers need. These attributes of Git are detailed above. In side-by-side comparisons with most other alternatives, many teams find that Git is very favorable.

### **Git is a de facto standard**

Git is the most broadly adopted tool of its kind. This is makes Git attractive for the following reasons. At Atlassian, nearly all of our project source code is managed in Git.

Vast numbers of developers already have Git experience and a significant proportion of college graduates may have experience with only Git. While some organizations may need to climb the learning curve when migrating to Git from another version control system, many of their existing and future developers do not need to be trained on Git.

In addition to the benefits of a large talent pool, the predominance of Git also means that many third party software tools and services are already integrated with Git including IDEs, and our own tools like DVCS desktop client [Sourcetree](https://www.atlassian.com/software/sourcetree), issue and project tracking software, [Jira](https://www.atlassian.com/software/jira), and code hosting service, [Bitbucket](https://bitbucket.org/).

If you are an inexperienced developer wanting to build up valuable skills in software development tools, when it comes to version control, Git should be on your list.

### **Git is a quality open source project**

Git is a very well supported open source project with over a decade of solid stewardship. The project maintainers have shown balanced judgment and a mature approach to meeting the long term needs of its users with regular releases that improve usability and functionality. The quality of the open source software is easily scrutinized and countless businesses rely heavily on that quality.

Git enjoys great community support and a vast user base. Documentation is excellent and plentiful, including books, tutorials and dedicated web sites. There are also podcasts and video tutorials.

Being open source lowers the cost for hobbyist developers as they can use Git without paying a fee. For use in open-source projects, Git is undoubtedly the successor to the previous generations of successful open source version control systems, SVN and CVS.

### **Criticism of Git**

One common criticism of Git is that it can be difficult to learn. Some of the terminology in Git will be novel to newcomers and for users of other systems, the Git terminology may be different, for example, revert in Git has a different meaning than in SVN or CVS. Nevertheless, Git is very capable and provides a lot of power to its users. Learning to use that power can take some time, however once it has been learned, that power can be used by the team to increase their development speed.

For those teams coming from a non-distributed VCS, having a central repository may seem like a good thing that they don't want to lose. However, while Git has been designed as a distributed version control system (DVCS), with Git, you can still have an official, canonical repository where all changes to the software must be stored. With Git, because each developer's repository is complete, their work doesn't need to be constrained by the availability and performance of the "central" server. During outages or while offline, developers can still consult the full project history. Because Git is flexible as well as being distributed, you can work the way you are accustomed to but gain the additional benefits of Git, some of which you may not even realise you're missing.

Now that you understand what version control is, what Git is and why software teams should use it, read on to discover the benefits Git can provide across the whole organization.

## Git for developers

### **Feature Branch Workflow**

One of the biggest advantages of Git is its branching capabilities. Unlike centralized version control systems, Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.

Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits. They let you represent development work at the same granularity as the your [agile backlog](https://www.atlassian.com/agile/backlogs). For example, you might implement a policy where each [Jira](https://www.atlassian.com/software/jira) ticket is addressed in its own feature branch.

### **Distributed Development**

In SVN, each developer gets a working copy that points back to a single central repository. Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits.

Having a full local history makes Git fast, since it means you don’t need a network connection to create commits, inspect previous versions of a file, or perform diffs between commits.

Distributed development also makes it easier to scale your engineering team. If someone breaks the production branch in SVN, other developers can’t check in their changes until it’s fixed. With Git, this kind of blocking doesn’t exist. Everybody can continue going about their business in their own local repositories.

And, similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else’s and start anew.

### **Pull Requests**

Many source code management tools such as [Bitbucket](https://bitbucket.org/) enhance core Git functionality with pull requests. A pull request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

Since they’re essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren’t destroying the entire project by treating pull requests as a formal code review.

### **Community**

In many circles, Git has come to be the expected version control system for new projects. If your team is using Git, odds are you won’t have to train new hires on your workflow, because they’ll already be familiar with distributed development.

In addition, Git is very popular among open source projects. This means it’s easy to leverage 3rd-party libraries and encourage others to fork your own open source code.

### **Faster Release Cycle**

The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle. These capabilities facilitate an [agile workflow](https://www.atlassian.com/agile/workflow) where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.

As you might expect, Git works very well with continuous integration and continuous delivery environments. Git hooks allow you to run scripts when certain events occur inside of a repository, which lets you automate deployment to your heart’s content. You can even build or deploy code from specific branches to different servers.

For example, you might want to configure Git to deploy the most recent commit from the develop branch to a test server whenever anyone merges a pull request into it. Combining this kind of build automation with peer review means you have the highest possible confidence in your code as it moves from development to staging to production.

Terminology

### Branch

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

### HEAD

Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

### Hook

A script that runs automatically every time a particular event occurs in a Git repository. Hooks let you customize Git’s internal behavior and trigger customizable actions at key points in the development life cycle.

### Master

The default development branch. Whenever you create a git repository, a branch named "master" is created, and becomes the active branch.

### Tag

A reference typically used to mark a particular point in the commit chain. In contrast to a head, a tag is not updated by the commit command.

### Version Control

A system that records changes to a file or set of files over time so that you can recall specific versions later.

### Working Tree

The tree of actual checked out files, normally containing the contents of the HEAD commit's tree and any local changes you've made but haven't yet committed.

# Git commands

### git add

Moves changes from the working directory to the staging area. This gives you the opportunity to prepare a snapshot before committing it to the official history.

### git branch

This command is your general-purpose branch administration tool. It lets you create isolated development environments within a single repository.

### git checkout

In addition to checking out old commits and old file revisions, git checkout is also the means to navigate existing branches. Combined with the basic Git commands, it’s a way to work on a particular line of development.

### git clean

Removes untracked files from the working directory. This is the logical counterpart to git reset, which (typically) only operates on tracked files.

### git clone

Creates a copy of an existing Git repository. Cloning is the most common way for developers to obtain a working copy of a central repository.

### git commit

Takes the staged snapshot and commits it to the project history. Combined with git add, this defines the basic workflow for all Git users.

### git commit --amend

Passing the --amend flag to git commit lets you amend the most recent commit. This is very useful when you forget to stage a file or omit important information from the commit message.

### git config

A convenient way to set configuration options for your Git installation. You’ll typically only need to use this immediately after installing Git on a new development machine.

### git fetch

Fetching downloads a branch from another repository, along with all of its associated commits and files. But, it doesn't try to integrate anything into your local repository. This gives you a chance to inspect changes before merging them with your project.

### git init

Initializes a new Git repository. If you want to place a project under revision control, this is the first command you need to learn.

### git log

Lets you explore the previous revisions of a project. It provides several formatting options for displaying committed snapshots.

### git merge

A powerful way to integrate changes from divergent branches. After forking the project history with git branch, git merge lets you put it back together again.

### git pull

Pulling is the automated version of git fetch. It downloads a branch from a remote repository, then immediately merges it into the current branch. This is the Git equivalent of svn update.

### git push

Pushing is the opposite of fetching (with a few caveats). It lets you move a local branch to another repository, which serves as a convenient way to publish contributions. This is like svn commit, but it sends a series of commits instead of a single changeset.

### git rebase

Rebasing lets you move branches around, which helps you avoid unnecessary merge commits. The resulting linear history is often much easier to understand and explore.

### git rebase -i

The -i flag is used to begin an interactive rebasing session. This provides all the benefits of a normal rebase, but gives you the opportunity to add, edit, or delete commits along the way

### git reflog

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag.

### git remote

A convenient tool for administering remote connections. Instead of passing the full URL to the fetch, pull, and push commands, it lets you use a more meaningful shortcut.

### git reset

Undoes changes to files in the working directory. Resetting lets you clean up or completely remove changes that have not been pushed to a public repository.

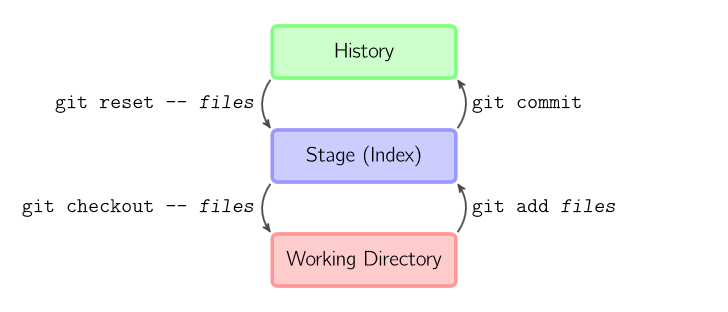
### git revert

Undoes a committed snapshot. When you discover a faulty commit, reverting is a safe and easy way to completely remove it from the code base.

### git status

Displays the state of the working directory and the staged snapshot. You’ll want to run this in conjunction with git add and git commit to see exactly what’s being included in the next snapshot.

<http://marklodato.github.io/visual-git-guide/index-en.html>

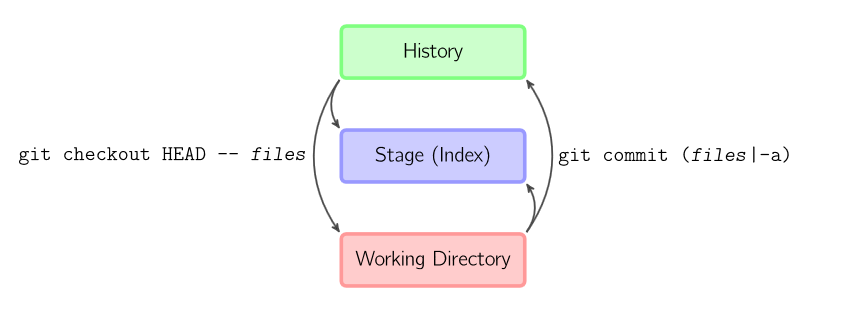


The four commands above copy files between the working directory, the stage (also called the index), and the history (in the form of commits).

* git add *files* copies *files* (at their current state) to the stage.
* git commit saves a snapshot of the stage as a commit.
* git reset -- *files* unstages files; that is, it copies *files* from the latest commit to the stage. Use this command to "undo" a git add *files*. You can also git reset to unstage everything.
* git checkout -- *files* copies *files* from the stage to the working directory. Use this to throw away local changes.

You can use git reset -p, git checkout -p, or git add -p instead of (or in addition to) specifying particular files to interactively choose which hunks copy.

It is also possible to jump over the stage and check out files directly from the history or commit files without staging first.

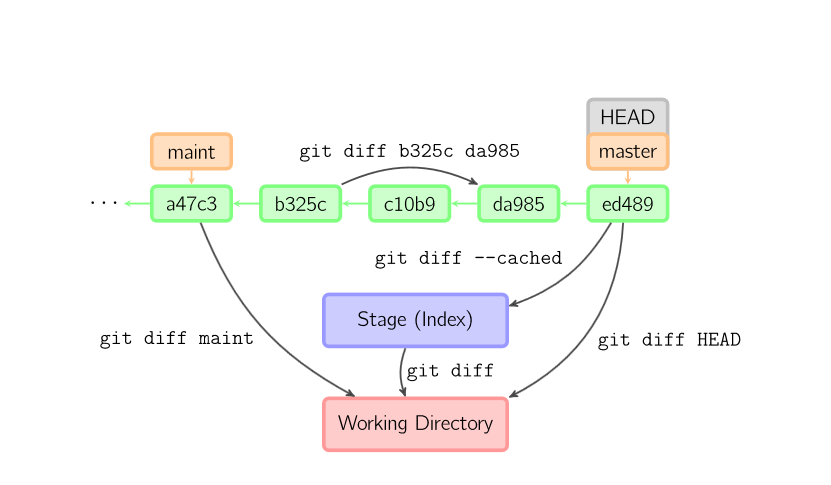


* git commit -a is equivalent to running git add on all filenames that existed in the latest commit, and then running git commit.
* git commit *files* creates a new commit containing the contents of the latest commit, plus a snapshot of *files* taken from the working directory. Additionally, *files* are copied to the stage.
* git checkout HEAD -- *files* copies *files* from the latest commit to both the stage and the working directory.

## Commands in Detail

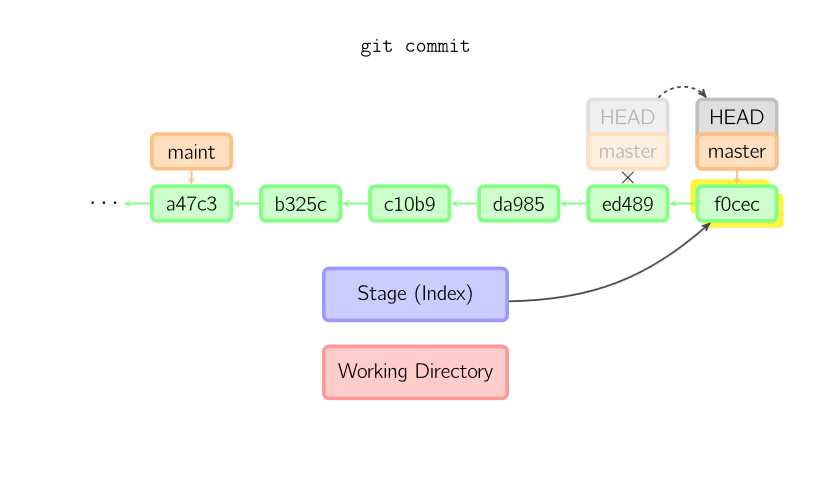
### Diff

There are various ways to look at differences between commits. Below are some common examples. Any of these commands can optionally take extra filename arguments that limit the differences to the named files

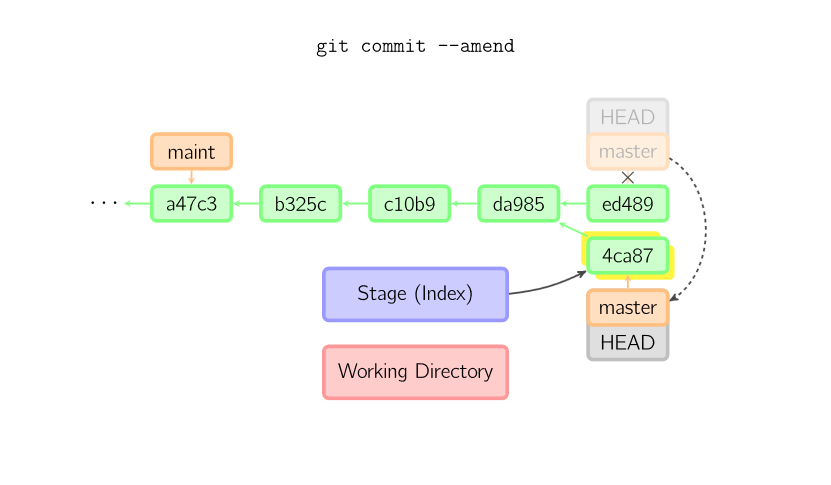


### Commit

When you commit, git creates a new commit object using the files from the stage and sets the parent to the current commit. It then points the current branch to this new commit. In the image below, the current branch is master. Before the command was run, master pointed to ed489. Afterward, a new commit, f0cec, was created, with parent ed489, and then master was moved to the new commit



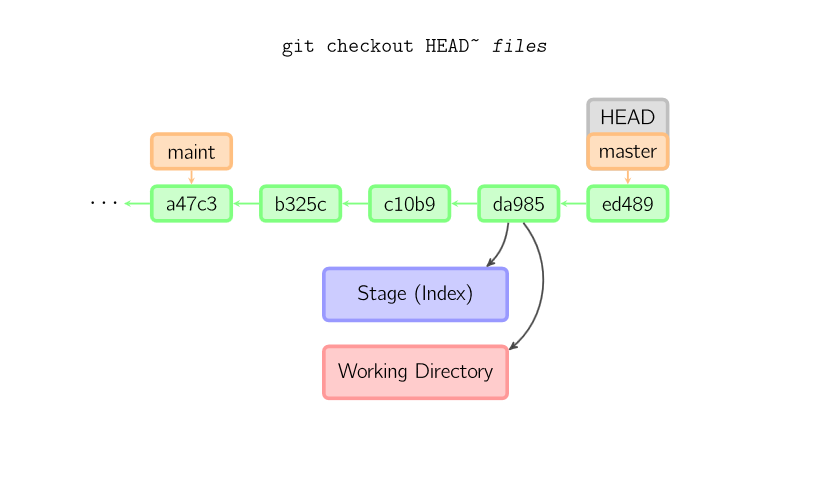
Sometimes a mistake is made in a commit, but this is easy to correct with git commit --amend. When you use this command, git creates a new commit with the same parent as the current commit. (The old commit will be discarded if nothing else references it.)



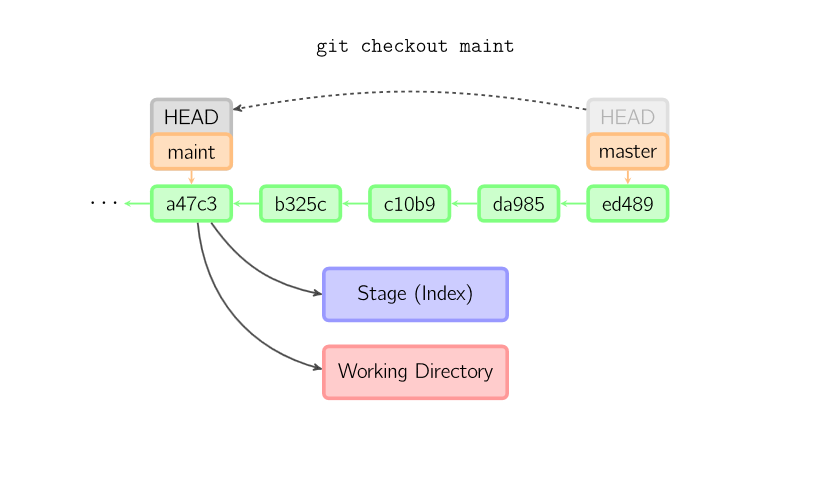
### Checkout

The checkout command is used to copy files from the history (or stage) to the working directory, and to optionally switch branches.

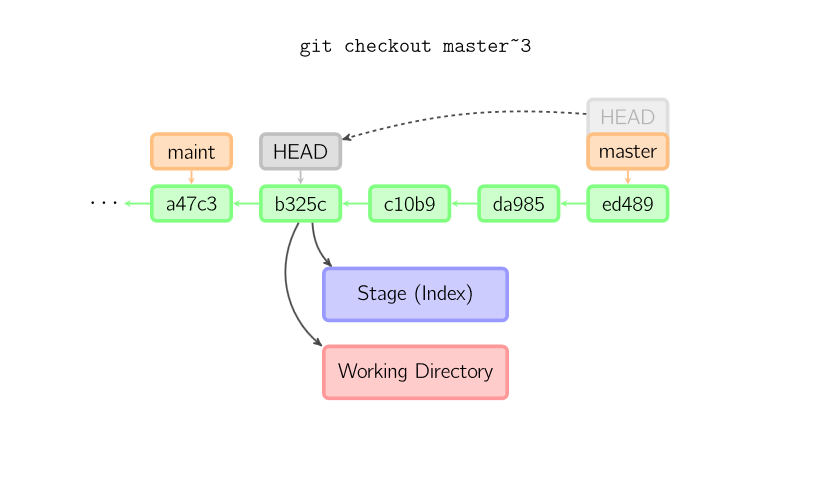
When a filename (and/or -p) is given, git copies those files from the given commit to the stage and the working directory. For example, git checkout HEAD~ foo.c copies the file foo.c from the commit called HEAD~ (the parent of the current commit) to the working directory, and also stages it. (If no commit name is given, files are copied from the stage.) Note that the current branch is not changed.



When a filename is not given but the reference is a (local) branch, HEAD is moved to that branch (that is, we "switch to" that branch), and then the stage and working directory are set to match the contents of that commit. Any file that exists in the new commit (a47c3 below) is copied; any file that exists in the old commit (ed489) but not in the new one is deleted; and any file that exists in neither is ignored.

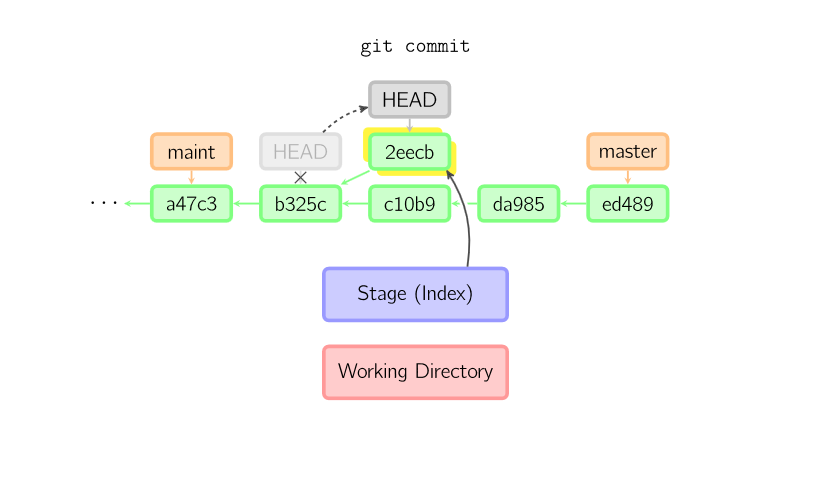


When a filename is not given and the reference is not a (local) branch — say, it is a tag, a remote branch, a SHA-1 ID, or something like master~3 — we get an anonymous branch, called a detached HEAD. This is useful for jumping around the history. Say you want to compile version 1.6.6.1 of git. You can git checkout v1.6.6.1 (which is a tag, not a branch), compile, install, and then switch back to another branch, say git checkout master. However, committing works slightly differently with a detached HEAD; this is covered [below](http://marklodato.github.io/visual-git-guide/index-en.html#detached).

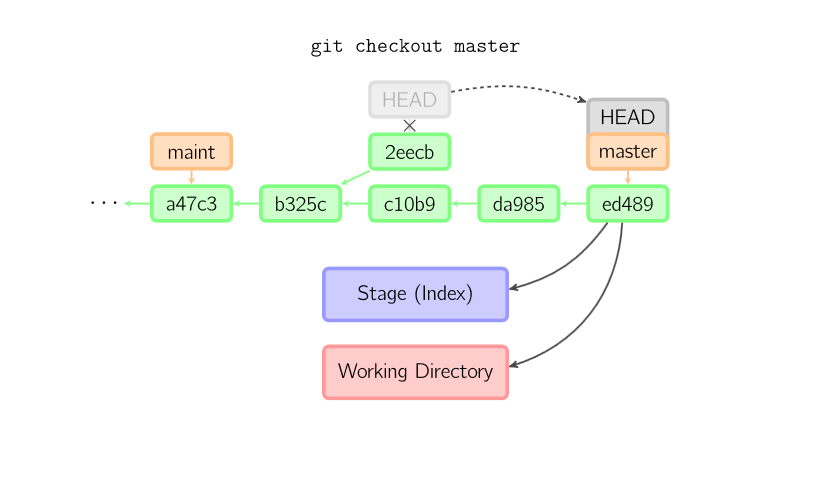


### Committing with a Detached HEAD

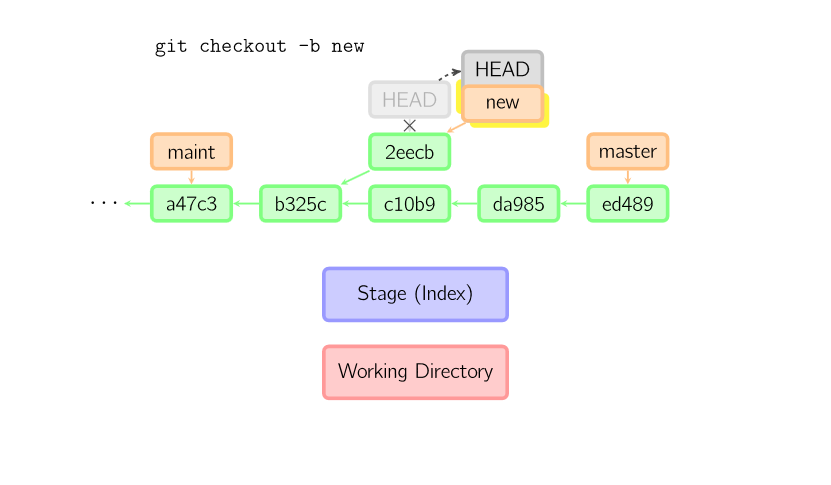
When HEAD is detached, commits work like normal, except no named branch gets updated. (You can think of this as an anonymous branch.)



Once you check out something else, say master, the commit is (presumably) no longer referenced by anything else, and gets lost. Note that after the command, there is nothing referencing 2eecb.



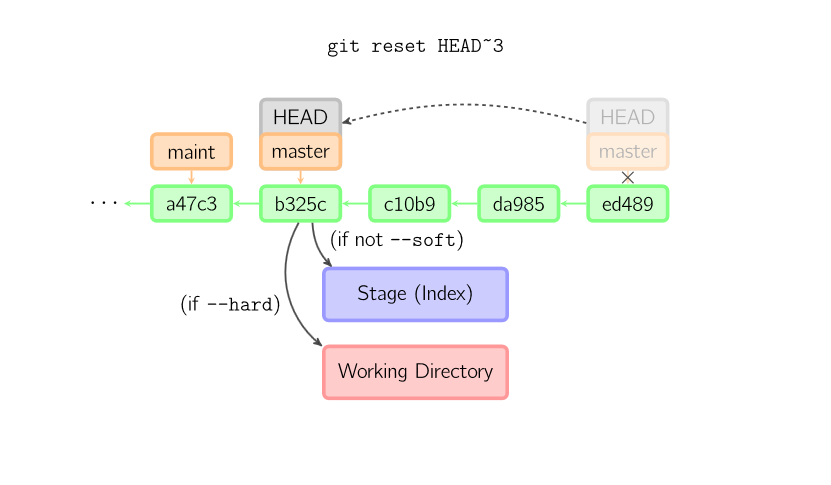
If, on the other hand, you want to save this state, you can create a new named branch using git checkout -b name



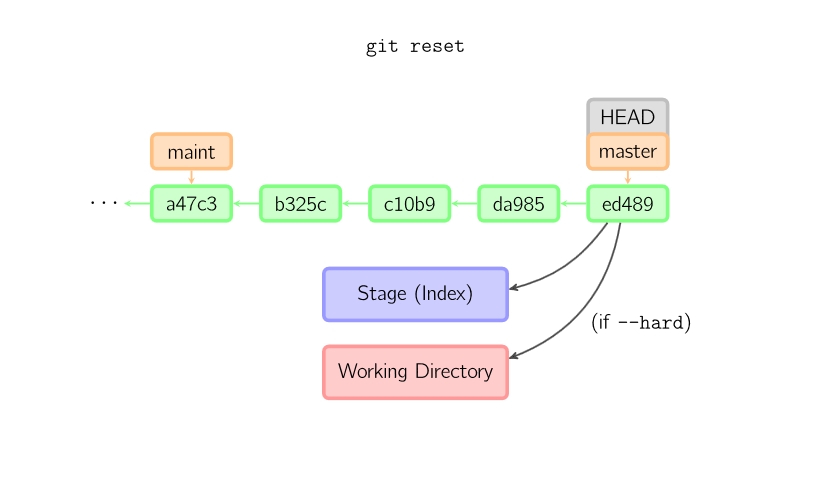
### Reset

The reset command moves the current branch to another position, and optionally updates the stage and the working directory. It also is used to copy files from the history to the stage without touching the working directory.

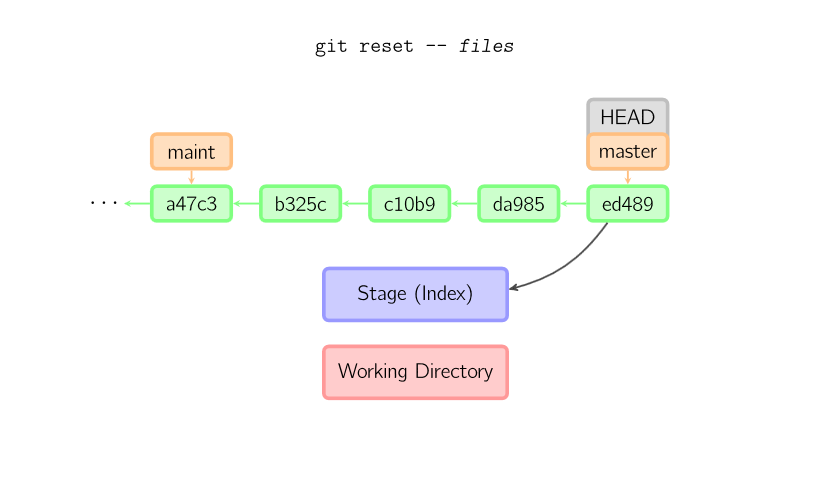
If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If --hard is given, the working directory is also updated. If --soft is given, neither is updated.



If a commit is not given, it defaults to HEAD. In this case, the branch is not moved, but the stage (and optionally the working directory, if --hard is given) are reset to the contents of the last commit

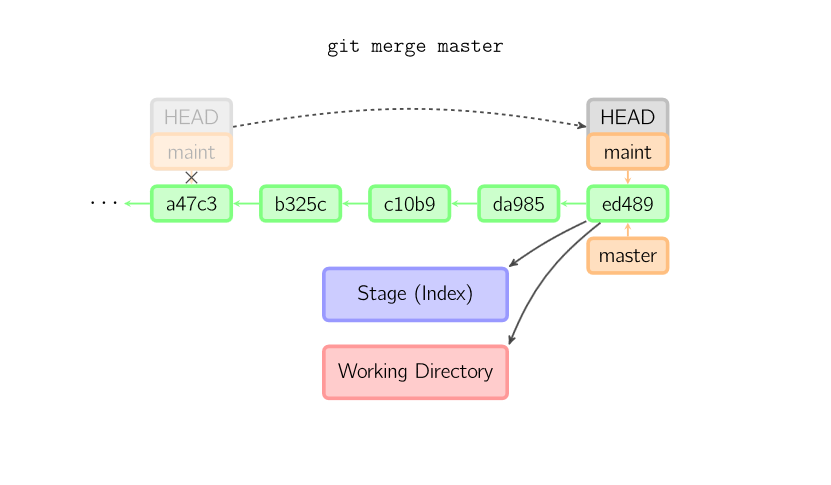


If a filename (and/or -p) is given, then the command works similarly to [checkout](http://marklodato.github.io/visual-git-guide/index-en.html#checkout) with a filename, except only the stage (and not the working directory) is updated. (You may also specify the commit from which to take files, rather than HEAD.)

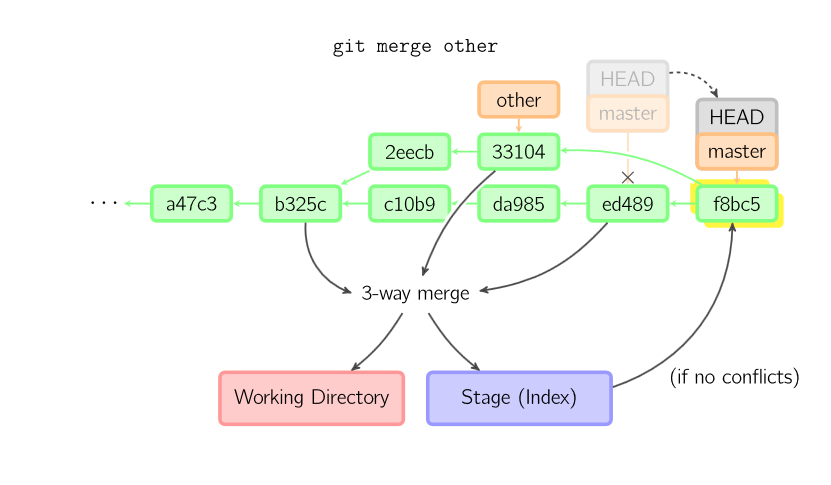


### Merge

A merge creates a new commit that incorporates changes from other commits. Before merging, the stage must match the current commit. The trivial case is if the other commit is an ancestor of the current commit, in which case nothing is done. The next most simple is if the current commit is an ancestor of the other commit. This results in a fast-forward merge. The reference is simply moved, and then the new commit is checked out.

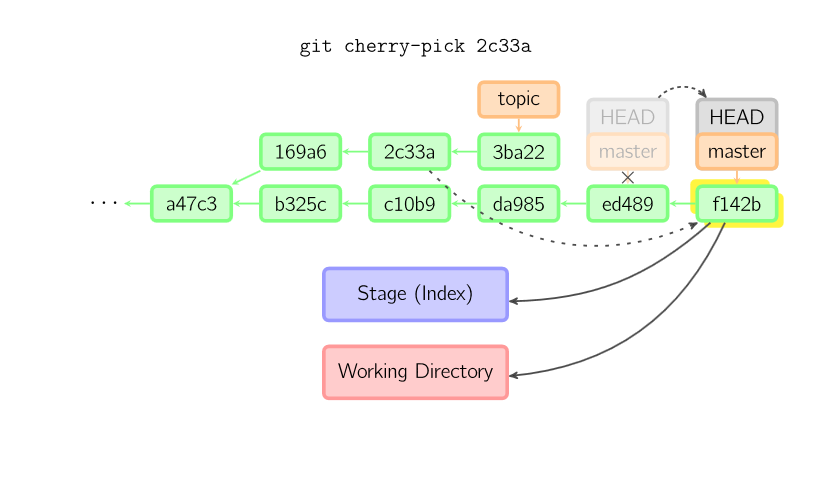


Otherwise, a "real" merge must occur. You can choose other strategies, but the default is to perform a "recursive" merge, which basically takes the current commit (ed489 below), the other commit (33104), and their common ancestor (b325c), and performs a [three-way merge](http://en.wikipedia.org/wiki/Three-way_merge). The result is saved to the working directory and the stage, and then a commit occurs, with an extra parent (33104) for the new commit.



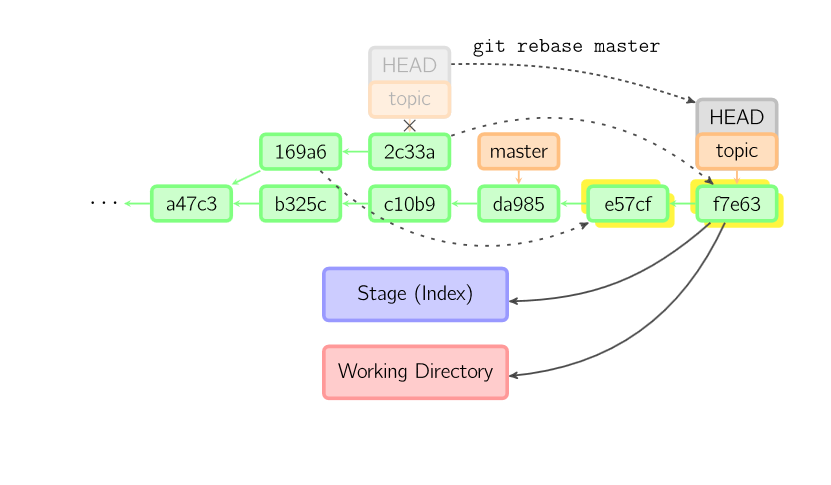
### Cherry Pick

The cherry-pick command "copies" a commit, creating a new commit on the current branch with the same message and patch as another commit.



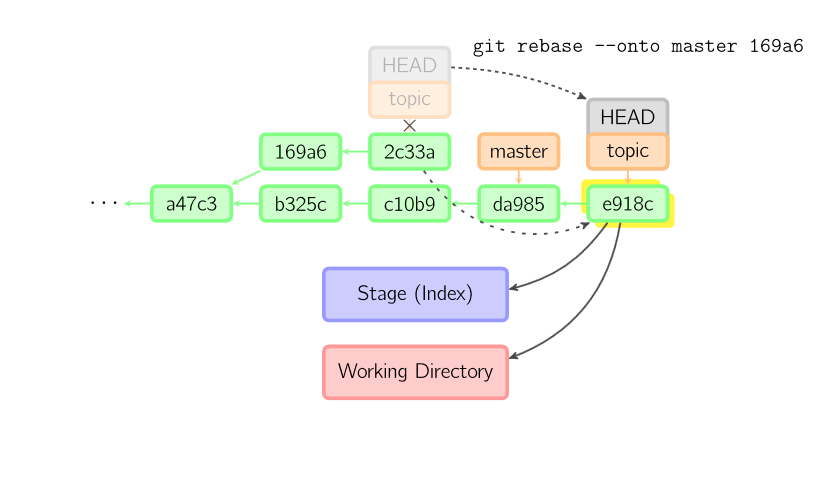
### Rebase

A rebase is an alternative to a [merge](http://marklodato.github.io/visual-git-guide/index-en.html#merge) for combining multiple branches. Whereas a merge creates a single commit with two parents, leaving a non-linear history, a rebase replays the commits from the current branch onto another, leaving a linear history. In essence, this is an automated way of performing several [cherry-pick](http://marklodato.github.io/visual-git-guide/index-en.html#cherry-pick)s in a row.



The above command takes all the commits that exist in topic but not in master (namely 169a6 and 2c33a), replays them onto master, and then moves the branch head to the new tip. Note that the old commits will be garbage collected if they are no longer referenced.

To limit how far back to go, use the --onto option. The following command replays onto master the most recent commits on the current branch since 169a6 (exclusive), namely 2c33a



There is also git rebase --interactive, which allows one to do more complicated things than simply replaying commits, namely dropping, reordering, modifying, and squashing commits. There is no obvious picture to draw for this; see [git-rebase(1)](http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html#_interactive_mode) for more details.

The contents of files are not actually stored in the index (.git/index) or in commit objects. Rather, each file is stored in the object database (.git/objects) as a blob, identified by its SHA-1 hash. The index file lists the filenames along with the identifier of the associated blob, as well as some other data. For commits, there is an additional data type, a tree, also identified by its hash. Trees correspond to directories in the working directory, and contain a list of trees and blobs corresponding to each filename within that directory. Each commit stores the identifier of its top-level tree, which in turn contains all of the blobs and other trees associated with that commit.

If you make a commit using a detached HEAD, the last commit really is referenced by something: the reflog for HEAD. However, this will expire after a while, so the commit will eventually be garbage collected, similar to commits discarded with git commit --amend or git rebase.

## Walkthrough: Watching the effect of commands

The following walks you through changes to a repository so you can see the effect of the command in real time, similar to how [Visualizing Git Concepts with D3](http://onlywei.github.io/explain-git-with-d3/) simulates them visually. Hopefully you find this useful.

Start by creating some repository:

$ **git init foo**

$ **cd foo**

$ **echo 1 > myfile**

$ **git add myfile**

$ **git commit -m "version 1"**

Now, define the following functions to help us show information:

show\_status() {

echo "HEAD: $(git cat-file -p HEAD:myfile)"

echo "Stage: $(git cat-file -p :myfile)"

echo "Worktree: $(cat myfile)"

}

initial\_setup() {

echo 3 > myfile

git add myfile

echo 4 > myfile

show\_status

}

Initially, everything is at version 1.

$ **show\_status**

HEAD: 1

Stage: 1

Worktree: 1

We can watch the state change as we add and commit.

$ **echo 2 > myfile**

$ **show\_status**

HEAD: 1

Stage: 1

Worktree: 2

$ **git add myfile**

$ **show\_status**

HEAD: 1

Stage: 2

Worktree: 2

$ **git commit -m "version 2"**

[master 4156116] version 2

1 file changed, 1 insertion(+), 1 deletion(-)

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 2

Now, let's create an initial state where the three are all different.

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

Let's watch what each command does. You will see that they match the diagrams above.

git reset -- myfile copies from HEAD to stage:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git reset -- myfile**

Unstaged changes after reset:

M myfile

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 4

git checkout -- myfile copies from stage to worktree:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git checkout -- myfile**

$ **show\_status**

HEAD: 2

Stage: 3

Worktree: 3

git checkout HEAD -- myfile copies from HEAD to both stage and worktree:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git checkout HEAD -- myfile**

$ **show\_status**

HEAD: 2

Stage: 2

Worktree: 2

git commit myfile copies from worktree to both stage and HEAD:

$ **initial\_setup**

HEAD: 2

Stage: 3

Worktree: 4

$ **git commit myfile -m "version 4"**

[master 679ff51] version 4

1 file changed, 1 insertion(+), 1 deletion(-)

$ **show\_status**

HEAD: 4

Stage: 4

Worktree: 4

<https://davidwalsh.name/tutorials/git>

How to Batch Update Git Commit Messages

**Prepending to Commit Messages**

To prepend text to every commit message in a given range, you'd execute a message like:

git filter-branch --msg-filter 'echo "bug ###### - \c" && cat' master..HEAD

You can also sed to achieve this:

git filter-branch -f --msg-filter 'sed "s/^/bug ###### - /"' master..HEAD

## Appending to Commit Messages

The case for appending to commit messages could be where you want to add the reviewer name(s) to the message.  Appending is roughly the same:

git filter-branch -f --msg-filter 'cat && echo "[Reviewer Walsh]"' master..HEAD

Quick git Commit Searching

One frequent git task is searching a list of commits on master branch which match a given keyword.  Here's how I do that:

git log -i --grep='ckeditor'

Create a Repository Archive with git

By [David Walsh](http://davidwalsh.name) on September 28, 2015

One feature I recently found out about is its archive feature which allows for exporting an entire repository to a zip or tar file.

# Format: git archive {branchname} --format={compression} --output={filename}

git archive master --format=tar --output=kuma.tar

git archive some-feature-branch --format=tar --output=kuma.tar

Sure you could use any archiving utility to archive a given repo, this feature allows for quick archiving of any branch or repository state!

List Recent git Commits from Command Line

By [David Walsh](http://davidwalsh.name) on January 5, 2016

.  I found a useful command for listing commits newest to oldest so on [commandlinefu](http://www.commandlinefu.com/commands/view/15064/show-a-git-log-with-offsets-relative-to-head):

git log --oneline | nl -v0 | sed 's/^ \+/&HEAD~/'

That command will render a listing with the latest commits first and original commit last:

Track Empty Directories with git

# Track Empty Directories with git

By [David Walsh](http://davidwalsh.name) on May 30, 2017

There are times when you'd like to track an empty directory within git but there's a problem: git wont allow you to add a directory that doesn't have a file in it.  The easy solution is putting an empty stub file within the directory, and the industry standard for that stub file name is .gitkeep

You can quickly create the file and commit the "empty" directory from command line:

touch my-empty-dir/.gitkeep

git add my-empty-dir/.gitkeep

git commit -m "Adding my empty directory"

The problem is simple, the solution is easy, but I wanted to highlight that .gitkeep is the industry standard.

Checkout the Previous Branch with git

By [David Walsh](http://davidwalsh.name) on June 29, 2017

I recently found out that you can switch the the previous branch you were on using the following command:

git checkout master

# Do whatever

git pull remote master

# Go back to the previous branch

git checkout -

Using - references the previous branch name, thus allowing you to navigate branches with ease!

Undo File Changes with Git

# Undo File Changes with Git

By [David Walsh](http://davidwalsh.name) on July 10, 2017

* To quickly undo file changes with git, execute the following two commands:
* git reset HEAD path/to/file.ext
* git checkout path/to/file.ext
* The second command (checkout) is required or you'll still see the file listed when running git status again.  With both of those executions, you'll no longer see the file listed with git status.
* git makes version control easy but the two steps needed to essentially revert changes to a file aren't intuitive, thus I thought I would share on this blog.  Happy coding!
* How to Delete a git Remote Branch

# How to Delete a git Remote Branch

* By [David Walsh](http://davidwalsh.name) on March 4, 2018

Once a branch is merged, for example, [we no longer need it around](https://davidwalsh.name/delete-merged-branches-git).

Deleting a branch on a local host machine repo is easy:

git branch -d <branch\_name>

To remove a branch from the remote git repository, like a GitHub-hosted repository, you can execute:

git push <remote\_name> --delete <branch\_name>

Reset File Changes with git

# Reset File Changes with git

By [David Walsh](http://davidwalsh.name) on April 24, 2018

You can restore a file's contents before a patch with the following:

git reset origin/master path/to/file-to-be-changed.ext

Once this shell snippet is executed, the file's contents are restored and can be re-commited to restore the file contents.

Search Git Commits Between Dates

# Search Git Commits Between Dates

By [David Walsh](http://davidwalsh.name) on July 11, 2018

You can use the following git command to list commits between two dates:

$ git log --after="2018-06-30" --before="2018-07-03" --oneline

Especially nice is the --oneline modifier to keep the commit list concise.