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.

# 1 Setting up a repository

[git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config)

This tutorial provides an overview of how to set up a repository (repo) under Git version control. This resource will walk you through initializing a Git repository for a new or existing project. Included below are workflow examples of repositories both created locally and cloned from remote repositories. This guide assumes a basic familiarity with a command-line interface.

The high level points this guide will cover are:

* Initializing a new Git repo
* Cloning an existing Git repo
* Committing a modified version of a file to the repo
* Configuring a Git repo for remote collaboration
* Common Git version control commands

By the end of this module, you should be able to create a Git repo, use common Git commands, commit a modified file, view your project’s history and configure a connection to a Git hosting service (Bitbucket).

## Initializing a new repository: git init

To create a new repo, you'll use the git init command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new master branch.

### Versioning an existing project with a new git repository

This example assumes you already have an existing project folder that you would like to create a repo within. You'll first cd to the root project folder and then execute the git init command.

cd /path/to/your/existing/code

git init

Pointing git init to an existing project directory will execute the same initialization setup as mentioned above, but scoped to that project directory.

git init <project directory>

Visit the [git init](http://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) page for a more detailed resource on git init.

## Cloning an existing repository: git clone

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all [version control](http://bitbucket-marketing.atlassian.com/product/version-control-software) operations are managed through their local repository.

git clone <repo url>

## Saving changes to the repository: git add and git commit

cd /path/to/project

echo "test content for git tutorial" >> CommitTest.txt

git add CommitTest.txt

git commit -m "added CommitTest.txt to the repo"

After executing this example, your repo will now have CommitTest.txt added to the history and will track future updates to the file.

This example introduced two additional git commands: add and commit.

## Repo-to-repo collaboration: git push

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out source code from an SVN repository. Unlike SVN, Git makes no distinction between the working copies and the central repository—they're all full-fledged [Git repositories](http://bitbucket-marketing.atlassian.com/product/code-repository).

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you push or pull commits from one repository to another.

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a centralized workflow using Git. This is accomplished through conventions rather than being hardwired into the VCS itself.

### Bare vs. cloned repositories

If you used git clone in the previous "Initializing a new Repository" section to set up your local repository, your repository is already configured for remote collaboration. git clone will automatically configure your repo with a remote pointed to the Git URL you cloned it from. This means that once you make changes to a file and commit them, you can git push those changes to the remote repository.

If you used git init to make a fresh repo, you'll have no remote repo to push changes to. A common pattern when initializing a new repo is to go to a hosted Git service like Bitbucket and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and git push to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping. We discuss this process in the Configuration & Set Up guide below.

If you prefer to host your own remote repo, you'll need to set up a "Bare Repository." Both git init and git clone accept a --bare argument. The most common use case for bare repo is to create a remote central Git repository

## Configuration & set up: git config

Once you have a remote repo setup, you will need to add a remote repo url to your local git config, and set an upstream branch for your local branches. The git remote command offers such utility.

git remote add <remote\_name> <remote\_repo\_url>

This command will map remote repository at <remote\_repo\_url> to a ref in your local repo under <remote\_name>. Once you have mapped the remote repo you can push local branches to it.

git push -u <remote\_name> <local\_branch\_name>

This command will push the local repo branch under <local\_branc\_name> to the remote repo at <remote\_name>.

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository. Several common configuration options are listed below.

Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

* Local: <repo>/.git/config – Repository-specific settings.
* Global: /.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* System: $(prefix)/etc/gitconfig – System-wide settings.

Define the author name to be used for all commits in the current repository. Typically, you’ll want to use the --global flag to set configuration options for the current user.

git config --global user.name <name>

Adding the --local option or not passing a config level option at all, will set the user.name for the current local repository.

git config --local user.email <email>

Define the author email to be used for all commits by the current user.

git config --global alias.<alias-name> <git-command>

Create a shortcut for a Git command. This is a powerful utility to create custom shortcuts for commonly used git commands. A simplistic example would be:

git config --global alias.ci commit

This creates a ci command that you can execute as a shortcut to git commit. To learn more about git aliases visit the [git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

git config --system core.editor <editor>

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor (e.g., vi). This example introduces the --system option. The --system option will set the configuration for the entire system, meaning all users and repos on a machine

git config --global --edit

Open the global configuration file in a text editor for manual editing.

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you’ll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag. One important exception is to override the author email address. You may wish to set your personal email address for personal and open source repositories, and your professional email address for work-related repositories.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

* <repo>/.git/config – Repository-specific settings.
* ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* $(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you’ll see something like the following:

The first thing you’ll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

Tell Git who you are git config

git --global user.name "John Smith" git config --global user.email john@example.com

Select your favorite text editor

git config --global core.editor vim

This will produce the ~ /.gitconfig file from the previous section

# 2 git init

This page will explore the git init command in depth. By the end of this page you will be informed on the core functionality and extended feature set of git init. This exploration includes:

* git init options and usage
* .git directory overview
* custom git init directory environment values
* git init vs. git clone
* git init bare repositories
* git init templates

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository. Most other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

Aside from the .git directory, in the root directory of the project, an existing project remains unaltered (unlike SVN, Git doesn't require a .git subdirectory in every subdirectory).

By default, git init will initialize the Git configuration to the .git subdirectory path.

## Usage

Compared to SVN, the git init command is an incredibly easy way to create new version-controlled projects. Git doesn’t require you to create a repository, import files, and check out a working copy. Additionally, Git does not require any pre-existing server or admin privileges. All you have to do is cd into your project subdirectory and run git init, and you'll have a fully functional Git repository.

git init

Transform the current directory into a Git repository. This adds a .git subdirectory to the current directory and makes it possible to start recording revisions of the project.

git init <directory>

Create an empty Git repository in the specified directory. Running this command will create a new subdirectory called containing nothing but the .git subdirectory.

If you've already run git init on a project directory and it contains a .git subdirectory, you can safely run git init again on the same project directory. It will not override an existing .git configuration.

### git init vs. git clone

A quick note: git init and git clone can be easily confused. At a high level, they can both be used to "initialize a new git repository." However, git clone is dependent on git init. git clone is used to create a copy of an existing repository. Internally, git clone first calls git init to create a new repository. It then copies the data from the existing repository, and checks out a new set of working files. Learn more on the [git clone page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone).

## Bare repositories --- git init --bare

git init --bare <directory>

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag (see discussion below). Conventionally, repositories initialized with the --bare flag end in .git. For example, the bare version of a repository called my-project should be stored in a directory called my-project.git.

The --bare flag creates a repository that doesn’t have a working directory, making it impossible to edit files and commit changes in that repository. You would create a bare repository to git push and git pull from, but never directly commit to it. Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite changes. Think of --bare as a way to mark a repository as a storage facility, as opposed to a development environment. This means that for virtually all Git workflows, the central repository is bare, and developers local repositories are non-bare.

## Configuration

All configurations of git init <directory> take a <directory> argument. If you provide the <directory>, the command is run inside it. If this directory does not exist, it will be created. In addition to the options and configuration already discussed, Git init has a few other command line options. A full list of them follows:

# 3 git clone

Here we'll examine the git clone command in depth. git clone is a Git command line utility which is used to target an existing repository and create a clone, or copy of the target repository. In this page we'll discuss extended configuration options and common use cases of git clone. Some points we'll cover here are:

* Cloning a local or remote repository
* Cloning a bare repository
* Using shallow options to partially clone repositories
* Git URL syntax and supported protocols

On the [setting up a repository guide](https://www.atlassian.com/git/tutorials/setting-up-a-repository), we covered a basic use case of git clone. This page will explore more complex cloning and configuration scenarios.

## Purpose: repo-to-repo collaboration development copy

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository

### Repo-to-repo collaboration

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they're all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you [push](https://www.atlassian.com/git/tutorials/syncing/git-push) or [pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commits from one repository to another

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a [centralized workflow](https://www.atlassian.com/git/tutorials/comparing-workflows) using Git. The point is, this is accomplished through conventions rather than being hardwired into the VCS itself.

## Usage

git clone is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory, at another location. The original repository can be located on the local filesystem or on remote machine accessible supported protocols. The git clone command copies an existing Git repository. This is sort of like SVN checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository. This makes it very easy to interact with a central repository. This automatic connection is established by creating Git refs to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.

git clone ssh://john@example.com/path/to/my-project.git

cd my-project

# Start working on the project

The first command initializes a new Git repository in the my-project folder on your local machine and populates it with the contents of the central repository. Then, you can cd into the project and start editing files, committing snapshots, and interacting with other repositories. Also note that the .git extension is omitted from the cloned repository. This reflects the non-bare status of the local copy.

### Cloning to a specific folder

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called ~<directory>! on the local machine.

### Cloning a specific tag

git clone -branch <tag> <repo>

Clone the repository located at <repo> and only clone the ref for <tag>.

### Shallow clone

git clone -depth=1 <repo>

Clone the repository located at <repo> and only clone the   
history of commits specified by the option depth=1. In this example a clone of <repo> is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

## Configuration options

### git clone -branch

The -branch argument lets you specify a specific a branch to clone instead of the branch the remote HEAD is pointing to, usually the master branch. In addition you can pass a tag instead of branch for the same effect.

git clone -branch new\_feature git://remoterepository.git

This above example would clone only the new\_feature branch from the remote Git repository. This is purely a convince utility to save you time from downloading the HEAD ref of the repository and then having to additionally fetch the ref you need.

### git clone -mirror vs. git clone -bare

#### git clone --bare

Similar to git init --bare, when the -bare argument is passed to git clone, a copy of the remote repository will be made with an omitted working directory. This means that a repository will be set up with the history of the project that can be pushed and pulled from, but cannot be edited directly. In addition, no remote branches for the repo will be configured with the -bare repository. Like git init --bare, this is used to create a hosted repository that developers will not edit directly.

#### git clone --mirror

Passing the --mirror argument implicitly passes the --bare argument as well. This means the behavior of --bare is inherited by --mirror. Resulting in a bare repo with no editable working files. In addition, --mirror will clone all the extended refs of the remote repository, and maintain remote branch tracking configuration. You can then run git remote update on the mirror and it will overwrite all refs from the origin repo. Giving you exact 'mirrored' functionality.

# 4 git config

In this document, we'll take an in-depth look at the git config command. We briefly discussed git config usage on our [Setting up a Repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) page. The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to .gitconfig text files. Executing git config will modify a configuration text file. We'll be covering common configuration settings like email, username, and editor. We'll discuss Git aliases, which allow you to create shortcuts for frequently used Git operations. Becoming familiar with git config and the various Git configuration settings will help you create a powerful, customized Git workflow.

## Usage

The most basic use case for git config is to invoke it with a configuration name, which will display the set value at that name. Configuration names are dot delimited strings composed of a 'section' and a 'key' based on their hierarchy. For example: user.email

git config user.email

In this example, email is a child property of the user configuration block. This will return the configured email address, if any, that Git will associate with locally created commits.

### git config levels and files

Before we further discuss git config usage, let's take a moment to cover configuration levels. The git config command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

* **--local**

By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: .git/config

* **--global**

Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory. ~ /.gitconfig on unix systems and C:\Users\<username>\.gitconfig on windows

* **--system**

System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. The system level configuration file lives in a gitconfig file off the system root path. $(prefix)/etc/gitconfig on unix systems. On windows this file can be found at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer.

Thus the order of priority for configuration levels is: local, global, system. This means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

### Writing a value

Expanding on what we already know about git config, let's look at an example in which we write a value:

git config --global user.email "your\_email@example.com"

This example writes the value your\_email@example.com to the configuration name user.email. It uses the --global flag so this value is set for the current operating system user.

### Writing a value

Expanding on what we already know about git config, let's look at an example in which we write a value:

git config --global user.email "your\_email@example.com"

This example writes the value your\_email@example.com to the configuration name user.email. It uses the --global flag so this value is set for the current operating system user.

## git config editor - core.editor

Many Git commands will launch a text editor to prompt for further input. One of the most common use cases for git config is configuring which editor Git should use. Listed below is a table of popular editors and matching git config commands:

## Merge tools

In the event of a merge conflict, Git will launch a "merge tool." By default, Git uses an internal implementation of the common Unix diff program. The internal Git diff is a minimal merge conflict viewer. There are many external third party merge conflict resolutions that can be used instead. For an overview of various merge tools and configuration, see our guide on [tips and tools to resolve conflits with Git](https://developer.atlassian.com/blog/2015/12/tips-tools-to-solve-git-conflicts/).

git config --global merge.tool kdiff3

## Aliases

You may be familiar with the concept of aliases from your operating system command-line; if not, they're custom shortcuts that define which command will expand to longer or combined commands. Aliases save you the time and energy cost of typing frequently used commands. Git provides its own alias system. A common use case for Git aliases is shortening the commit command. Git aliases are stored in Git configuration files. This means you can use the git config command to configure aliases.

git config --global alias.ci commit

This example creates a ci alias for the git commit command. You can then invoke git commit by executing git ci. Aliases can also reference other aliases to create powerful combos.

git config --global alias.amend ci --amend

This example creates an alias amend which composes the ci alias into a new alias that uses --amend flag.

**Formatting & whitespace**

Git has several "whitespace" features that can be configured to highlight whitespace issues when using git diff. The whitespace issues will be highlighted using the configured color color.diff.whitespace

The following features are enabled by default:

* blank-at-eol highlights orphan whitespaces at the line endings
* space-before-tab highlights a space character that appears before a tab character when indenting a line
* blank-at-eof highlights blank lines inserted at the end of a file

The following features are disabled by default

* indent-with-non-tab highlights a line that is indented with spaces instead of tabs
* tab-in-indent highlights an initial tab indent as an error
* trailing-space is shorthand for both blank-at-eol and blank-at-eof
* cr-at-eol highlights a carriage-return at the line endings
* tabwidth=<n> defines how many character positions a tab occupies. The default value is 8. Allowed values are 1-63

## Summary

In this article, we covered the use of the git config command. We discussed how the command is a convince method for editing raw git config files on the filesystem. We looked at basic read and write operations for configuration options. We took a look at common config patterns:

* How to configure the Git editor
* How to override configuration levels
* How to reset configuration defaults
* How to customize git colors

Overall, git config is a helper tool that provides a shortcut to editing raw git config files on disk. We covered in depth personal customization options. Basic knowledge of git configuration options is a prerequisite for [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository). See our guide there for a demonstration of the basics.

# 5 Git Alias

This section will focus on Git aliases. To better understand the value of Git aliases we must first discuss what an alias is. The term alias is synonymous with a shortcut. Alias creation is a common pattern found in other popular utilities like `bash` shell. Aliases are used to create shorter commands that map to longer commands. Aliases enable more efficient workflows by requiring fewer keystrokes to execute a command. For a brief example, consider the git checkout command. The checkout command is a frequently used git command, which adds up in cumulative keystrokes over time. An alias can be created that maps git co to git checkout, which saves precious human fingertip power by allowing the shorter keystroke form: git co to be typed instead.

## Git Alias Overview

It is important to note that there is no direct git alias command. Aliases are created through the use of the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) command and the Git configuration files. As with other configuration values, aliases can be created in a local or global scope.  
  
To better understand Git aliases let us create some examples.

$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status

The previous code example creates globally stored shortcuts for common git commands. Creating the aliases will not modify the source commands. So git checkout will still be available even though we now have the git co alias. These aliases were created with the --global flag which means they will be stored in Git's global operating system level configuration file. On linux systems, the global config file is located in the User home directory at /.gitconfig.

    [alias]  
        co = checkout  
            br = branch  
            ci = commit  
            st = status

This demonstrates that the aliases are now equivalent to the source commands.

## Usage

Git aliasing is enabled through the use of git config, For command-line option and usage examples please review the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) documentation.

## Examples

### Using aliases to create new Git commands

A common Git pattern is to remove recently added files from the staging area. This is achieved by leveraging options to the git reset command. A new alias can be created to encapsulate this behavior and create a new alias-command-keyword which is easy to remember:

git config --global alias.unstage 'reset HEAD --'

The preceding code example creates a new alias unstage. This now enables the invocation of git unstage. git unstage which will perform a reset on the staging area. This makes the following two commands equivalent.

git unstage fileA  
$ git reset HEAD -- fileA

## Discussion

### How do I create Git Aliases?

Aliases can be created through two primary methods:

#### Directly editing Git config files

The global or local config files can be manually edited and saved to create aliases. The global config file lives at $HOME/.gitconfig file path. The local path lives within an active git repository at /.git/config  
  
The config files will respect an [alias] section that looks like:

[alias]  
 co = checkout

This means that co is a shortcut for checkout

#### Using the git config to create aliases

As previously demonstrated the git config command is a convenient utility to quickly create aliases. The git config command is actually a helper utility for writing to the global and local Git config files.

git config --global alias.co checkout

Invoking this command will update the underlying global config file just as it had been edited in our previous example.

## Git Alias Summary

Git aliases are a powerful workflow tool that create shortcuts to frequently used Git commands. Using Git aliases will make you a faster and more efficient developer. Aliases can be used to wrap a sequence of Git commands into new faux Git command. Git aliases are created through the use of the git config command which essentially modifies local or global Git config files. Learn more on the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) page.

# 6 Saving changes

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

When working in Git, or other version control systems, the concept of "saving" is a more nuanced process than saving in a word processor or other traditional file editing applications. The traditional software expression of "saving" is synonymous with the Git term "committing". A commit is the Git equivalent of a "save". Traditional saving should be thought of as a file system operation that is used to overwrite an existing file or write a new file. Alternatively, Git committing is an operation that acts upon a collection of files and directories.

Saving changes in Git vs SVN is also a different process. SVN Commits or 'check-ins' are operations that make a remote push to a centralized server. This means an SVN commit needs Internet access in order to fully 'save' project changes. Git commits can be captured and built up locally, then pushed to a remote server as needed using the git push -u origin master command. The difference between the two methods is a fundamental difference between architecture designs. Git is a distributed application model whereas SVN is a centralized model. Distributed applications are generally more robust as they do not have a single point of failure like a centralized server.

The commands: git add, [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), and [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) are all used in combination to save a snapshot of a Git project's current state.

Git has an additional saving mechanism called 'the stash'. The stash is an ephemeral storage area for changes that are not ready to be committed. The stash operates on the working directory, the first of [the three trees](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) and has extensive usage options. To learn more visit the [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) page.

A Git repository can be configured to ignore specific files or directories. This will prevent Git from saving changes to any ignored content. Git has multiple methods of configuration that manage the ignore list. Git ignore configure is discussed in further detail on the [git ignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore) page.

## git add

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit).

In conjunction with these commands, you'll also need [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) to view the state of the working directory and the staging area.

## How it works

The git add and [git commit](https://www.atlassian.com/git/tutorials/saving-changes) commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team’s collaboration model. They are the means to record versions of a project into the repository’s history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you’re ready to save a copy of the current state of the project, you stage changes with git add. After you’re happy with the staged snapshot, you commit it to the project history with git commit. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command is used to undo a commit or staged snapshot.

In addition to git add and git commit, a third command [git push](https://www.atlassian.com/git/tutorials/syncing) is essential for a complete collaborative Git workflow. git push is utilized to send the committed changes to remote repositories for collaboration. This enables other team members to access a set of saved changes.

The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

## The staging area

The primary function of the git add command, is to promote pending changes in the working directory, to the git staging area. The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you’re coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history. The staging area is considered one of the ["three trees" of Git](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), along with, the working directory, and the commit history.

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it’s important to create atomic commits so that it’s easy to track down bugs and revert changes with minimal impact on the rest of the project.

## Common options

git add <file>

Stage all changes in <file> for the next commit.

git add <directory>

Stage all changes in <directory> for the next commit.

git add -p

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

## Examples

When you’re starting a new project, git add serves the same function as svn import. To create an initial commit of the current directory, use the following two commands:

git add .

git commit

Once you’ve got your project up-and-running, new files can be added by passing the path to git add:

git add hello.py

git commit

The above commands can also be used to record changes to existing files. Again, Git doesn’t differentiate between staging changes in new files vs. changes in files that have already been added to the repository.

## Summary

In review, git add is the first command in a chain of operations that directs Git to "save" a snapshot of the current project state, into the commit history. When used on its own, git add will promote pending changes from the working directory to the staging area. The [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command is used to examine the current state of the repository and can be used to confirm a git add promotion. The [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) command is used to undo a git add. The [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) command is then used to Commit a snapshot of the staging directory to the repositories commit history.

# Git diff

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

## Comparing changes with git diff

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more. This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

## Reading diffs: outputs

### Raw output format

The following examples will be executed in a simple repo. The repo is created with the commands below:

$:> mkdir diff\_test\_repo

$:> cd diff\_test\_repo

$:> touch diff\_test.txt

$:> echo "this is a git diff test example" > diff\_test.txt

$:> git init .

Initialized empty Git repository in /Users/kev/code/test/.git/

$:> git add diff\_test.txt

$:> git commit -am"add diff test file"

[master (root-commit) 6f77fc3] add diff test file

1 file changed, 1 insertion(+)

create mode 100644 diff\_test.txt

If we execute git diff at this point, there will be no output. This is expected behavior as there are no changes in the repo to diff. Once the repo is created and we've added the diff\_test.txt file, we can change the contents of the file to start experimenting with diff output.

$:> echo "this is a diff example" > diff\_test.txt

Executing this command will change the content of the diff\_test.txt file. Once modified, we can view a diff and analyze the output. Now executing git diff will produce the following output:

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Let us now examine a more detailed breakdown of the diff output.

### 1. Comparison input

diff --git a/diff\_test.txt b/diff\_test.txt

This line displays the input sources of the diff. We can see that a/diff\_test.txt and b/diff\_test.txt have been passed to the diff.

### 2. Meta data

index 6b0c6cf..b37e70a 100644

This line displays some internal Git metadata. You will most likely not need this information. The numbers in this output correspond to Git object version hash identifiers.

### 3. Markers for changes

--- a/diff\_test.txt

+++ b/diff\_test.txt

These lines are a legend that assigns symbols to each diff input source. In this case, changes from a/diff\_test.txt are marked with a --- and the changes from b/diff\_test.txt are marked with the +++ symbol.

### 4. Diff chunks

The remaining diff output is a list of diff 'chunks'. A diff only displays the sections of the file that have changes. In our current example, we only have one chunk as we are working with a simple scenario. Chunks have their own granular output semantics.

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

The first line is the chunk header. Each chunk is prepended by a header inclosed within @@ symbols. The content of the header is a summary of changes made to the file. In our simplified example, we have -1 +1 meaning line one had changes. In a more realistic diff, you would see a header like:

@@ -34,6 +34,8 @@

In this header example, 6 lines have been extracted starting from line number 34. Additionally, 8 lines have been added starting at line number 34.

The remaining content of the diff chunk displays the recent changes. Each changed line is prepended with a + or - symbol indicating which version of the diff input the changes come from. As we previously discussed, - indicates changes from the a/diff\_test.txt and + indicates changes from b/diff\_test.txt.

## Highlighting changes

### 1. git diff --color-words

git diff also has a special mode for highlighting changes with much better granularity: ‐‐color-words. This mode tokenizes added and removed lines by whitespace and then diffs those.

$:> git diff --color-words

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

this is agit difftest example

Now the output displays only the color-coded words that have changed.

### 2. git diff-highlight

If you clone the git source, you’ll find a sub-directory called contrib. It contains a bunch of git-related tools and other interesting bits and pieces that haven’t yet been promoted to git core. One of these is a Perl script called diff-highlight. Diff-highlight pairs up matching lines of diff output and highlights sub-word fragments that have changed.

$:> git diff | /your/local/path/to/git-core/contrib/diff-highlight/diff-highlight

diff --git a/diff\_test.txt b/diff\_test.txt

index 6b0c6cf..b37e70a 100644

--- a/diff\_test.txt

+++ b/diff\_test.txt

@@ -1 +1 @@

-this is a git diff test example

+this is a diff example

Now we’ve pared down our diff to the smallest possible change.

## Diffing binary files

In addition to the text file utilities we have thus far demonstrated, git diff can be run on binary files. Unfortunately, the default output is not very helpful.

$:> git diff

Binary files a/script.pdf and b/script.pdf differ

Git does have a feature that allows you to specify a shell command to transform the content of your binary files into text prior to performing the diff. It does require a little set up though. First, you need to specify a textconv filter describing how to convert a certain type of binary to text. We're using a simple utility called pdftohtml (available via homebrew) to convert my PDFs into human readable HTML. You can set this up for a single repository by editing your .git/config file, or globally by editing ~ /.gitconfig

[diff "pdfconv"]

textconv=pdftohtml -stdout

Then all you need to do is associate one or more file patterns with our pdfconv filter. You can do this by creating a .gitattributes file in the root of your repository.

\*.pdf diff=pdfconv

Once configured, git diff will first run the binary file through the configured converter script and diff the converter output. The same technique can be applied to get useful diffs from all sorts of binary files, for example: zips, jars and other archives: using unzip -l (or similar) in place of pdf2html will show you paths that have been added or removed between commits images: exiv2 can be used to show metadata changes such as image dimensions documents: conversion tools exist for transforming .odf, .doc and other document formats to plain text. In a pinch, strings will often work for binary files where no formal converter exists.

## Comparing files: git diff file

The git diff command can be passed an explicit file path option. When a file path is passed to git diff the diff operation will be scoped to the specified file. The below examples demonstrate this usage.

git diff HEAD ./path/to/file

This example is scoped to ./path/to/file when invoked, it will compare the specific changes in the working directory, against the index, showing the changes that are not staged yet. By default git diff will execute the comparison against HEAD. Omitting HEAD in the example above git diff ./path/to/file has the same effect.

git diff --cached ./path/to/file

When git diff is invoked with the --cached option the diff will compare the staged changes with the local repository. The --cached option is synonymous with --staged.

## Comparing all changes

Invoking git diff without a file path will compare changes across the entire repository. The above, file specific examples, can be invoked without the ./path/to/file argument and have the same output results across all files in the local repo.

## Changes since last commit

By default git diff will show you any uncommitted changes since the last commit.

git diff

## Comparing files between two different commits

git diff can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff.

$:> git log --prety=oneline

957fbc92b123030c389bf8b4b874522bdf2db72c add feature

ce489262a1ee34340440e55a0b99ea6918e19e7a rename some classes

6b539f280d8b0ec4874671bae9c6bed80b788006 refactor some code for feature

646e7863348a427e1ed9163a9a96fa759112f102 add some copy to body

$:> git diff 957fbc92b123030c389bf8b4b874522bdf2db72c ce489262a1ee34340440e55a0b99ea6918e19e7a

## Comparing branches

### Comparing two branches

Branches are compared like all other ref inputs to git diff

git diff branch1..other-feature-branch

This example introduces the dot operator. The two dots in this example indicate the diff input is the tips of both branches. The same effect happens if the dots are omitted and a space is used between the branches. Additionally, there is a three dot operator:

git diff branch1...other-feature-branch

The three dot operator initiates the diff by changing the first input parameter branch1. It changes branch1 into a ref of the shared common ancestor commit between the two diff inputs, the shared ancestor of branch1 and other-feature-branch. The last parameter input parameter remains unchanged as the tip of other-feature-branch.

## Comparing files from two branches

To compare a specific file across branches, pass in the path of the file as the third argument to git diff

git diff master new\_branch ./diff\_test.txt

## Summary

This page disscused the Git diffing process and the git diff command. We discussed how to read git diff output and the various data included in the output. Examples were provided on how to alter the git diff output with highlighting and colors. We discussed different diffing strategies such as how to diff files in branches and specific commits. In addition to the git diff command, we also used git log and git checkout.

# Git stash

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

* Git Stash
  + [Stashing your work](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#stashing-your-work)
  + [Re-applying your stashed changes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#re-applying-your-stashed-changes)
  + [Stashing untracked or ignored files](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#stashing-untracked-or-ignored)
  + [Managing multiple stashes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#managing-multiple-stashes)
  + [Viewing stash diffs](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#viewing-stash-diffs)
  + [Partial stashes](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#partial-stashes)
  + [Creating a branch from your stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#creating-a-branch-from-your-stash)
  + [Cleaning up your stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#cleaning-up-your-stash)
  + [How git stash works](https://www.atlassian.com/git/tutorials/saving-changes/git-stash#how-git-stash-works)

## Stashing your work

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master

nothing to commit, working tree clean

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

## Re-applying your stashed changes

You can reapply previously stashed changes with git stash pop:

$ git status

On branch master

nothing to commit, working tree clean

$ git stash pop

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:

$ git stash apply

On branch master

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

This is useful if you want to apply the same stashed changes to multiple branches.

Now that you know the basics of stashing, there is one caveat with git stash you need to be aware of: by default Git won't stash changes made to untracked or ignored files.

## Stashing untracked or ignored files

By default, running git stash will stash:

* changes that have been added to your index (staged changes)
* changes made to files that are currently tracked by Git (unstaged changes)

But it will **not** stash:

* new files in your working copy that have not yet been staged
* files that have been [ignored](https://www.atlassian.com/git/tutorials/gitignore)

So if we add a third file to our example above, but don't stage it (i.e. we don't run git add), git stash won't stash it.

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files:

You can include changes to [ignored](https://www.atlassian.com/git/tutorials/gitignore) files as well by passing the -a option (or --all) when running git stash.



## Managing multiple stashes

You aren't limited to a single stash. You can run git stash several times to create multiple stashes, and then use git stash list to view them. By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from. After a while it can be difficult to remember what each stash contains:

$ git stash list

stash@{0}: WIP on master: 5002d47 our new homepage

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

To provide a bit more context, it's good practice to annotate your stashes with a description, using git stash save "message":

$ git stash save "add style to our site"

Saved working directory and index state On master: add style to our site

HEAD is now at 5002d47 our new homepage

$ git stash list

stash@{0}: On master: add style to our site

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage

By default, git stash pop will re-apply the most recently created stash: stash@{0}

You can choose which stash to re-apply by passing its identifier as the last argument, for example:

$ git stash pop stash@{2}

## Viewing stash diffs

You can view a summary of a stash with git stash show:

$ git stash show

index.html | 1 +

style.css | 3 +++

2 files changed, 4 insertions(+)

Or pass the -p option (or --patch) to view the full diff of a stash:

$ git stash show -p

diff --git a/style.css b/style.css

new file mode 100644

index 0000000..d92368b

--- /dev/null

+++ b/style.css

@@ -0,0 +1,3 @@

+\* {

+ text-decoration: blink;

+}

diff --git a/index.html b/index.html

index 9daeafb..ebdcbd2 100644

--- a/index.html

+++ b/index.html

@@ -1 +1,2 @@

+<link rel="stylesheet" href="style.css"/>

## Partial stashes

You can also choose to stash just a single file, a collection of files, or individual changes from within files. If you pass the -p option (or --patch) to git stash, it will iterate through each changed "hunk" in your working copy and ask whether you wish to stash it:

$ git stash -p

diff --git a/style.css b/style.css

new file mode 100644

index 0000000..d92368b

--- /dev/null

+++ b/style.css

@@ -0,0 +1,3 @@

+\* {

+ text-decoration: blink;

+}

Stash this hunk [y,n,q,a,d,/,e,?]? y

diff --git a/index.html b/index.html

index 9daeafb..ebdcbd2 100644

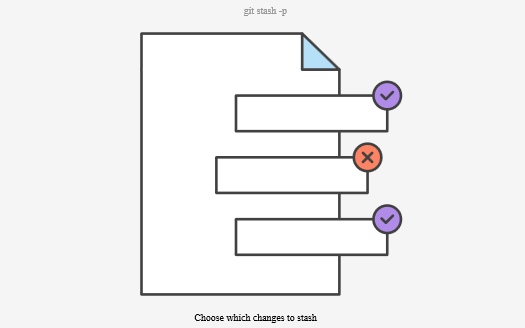
--- a/index.html

+++ b/index.html

@@ -1 +1,2 @@

+<link rel="stylesheet" href="style.css"/>

Stash this hunk [y,n,q,a,d,/,e,?]? n



## Creating a branch from your stash

If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to:

$ git stash branch add-stylesheet stash@{1}

Switched to a new branch 'add-stylesheet'

On branch add-stylesheet

Changes to be committed:

new file: style.css

Changes not staged for commit:

modified: index.html

Dropped refs/stash@{1} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)

This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it.

## Cleaning up your stash

If you decide you no longer need a particular stash, you can delete it with git stash drop:

$ git stash drop stash@{1}

Dropped stash@{1} (17e2697fd8251df6163117cb3d58c1f62a5e7cdb)

Or you can delete all of your stashes with:

$ git stash clear

## How git stash works

If you just wanted to know how to use git stash, you can stop reading here. But if you're curious about how Git (and git stash) works under the hood, read on!

Stashes are actually encoded in your repository as commit objects. The special ref at .git/refs/stash points to your most recently created stash, and previously created stashes are referenced by the stash ref's reflog. This is why you refer to stashes by stash@{n}: you're actually referring to the nth reflog entry for the stash ref. Since a stash is just a commit, you can inspect it with git log:

$ git log --oneline --graph stash@{0}

\*-. 953ddde WIP on master: 5002d47 our new homepage

|\ \

| | \* 24b35a1 untracked files on master: 5002d47 our new homepage

| \* 7023dd4 index on master: 5002d47 our new homepage

|/

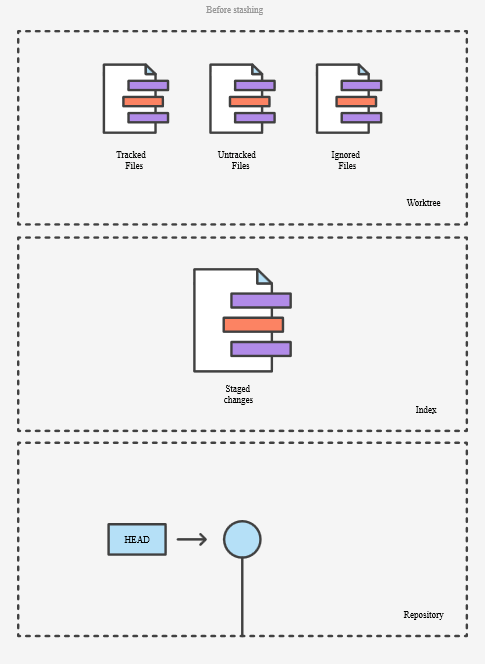
\* 5002d47 our new homepage

Depending on what you stashed, a single git stash operation creates either two or three new commits. The commits in the diagram above are:

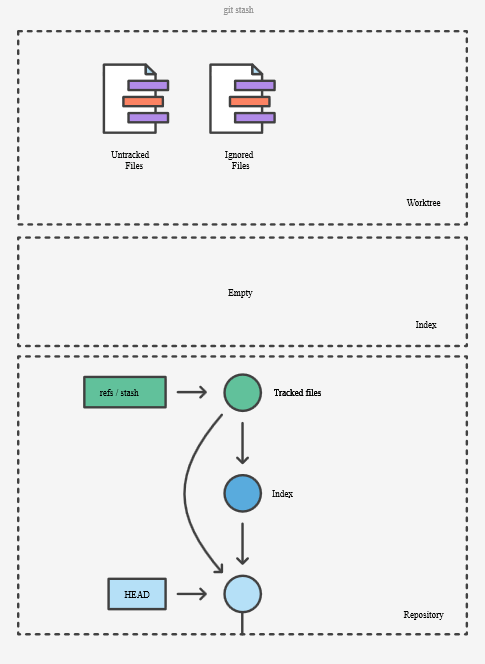
* stash@{0}, a new commit to store the tracked files that were in your working copy when you ran git stash
* stash@{0}'s first parent, the pre-existing commit that was at HEAD when you ran git stash
* stash@{0}'s second parent, a new commit representing the index when you ran git stash
* stash@{0}'s third parent, a new commit representing untracked files that were in your working copy when you ran git stash. This third parent only created if:
  + your working copy actually contained untracked files; and
  + you specified the --include-untracked or --all option when invoked git stash.

How git stash encodes your worktree and index as commits:

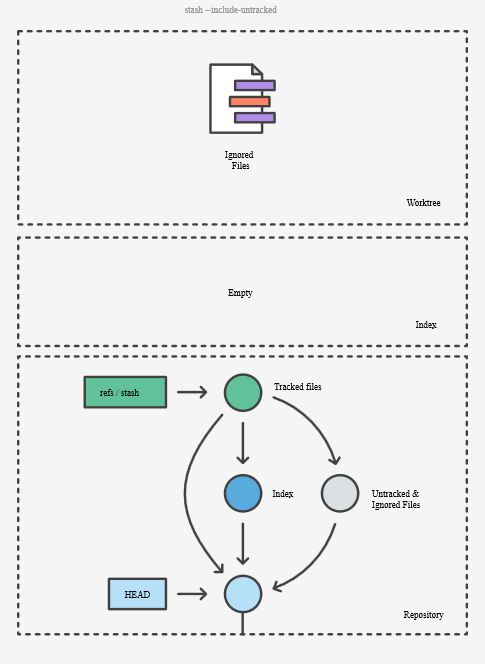
* Before stashing, your worktree may contain changes to tracked files, untracked files, and ignored files. Some of these changes may also be staged in the index.



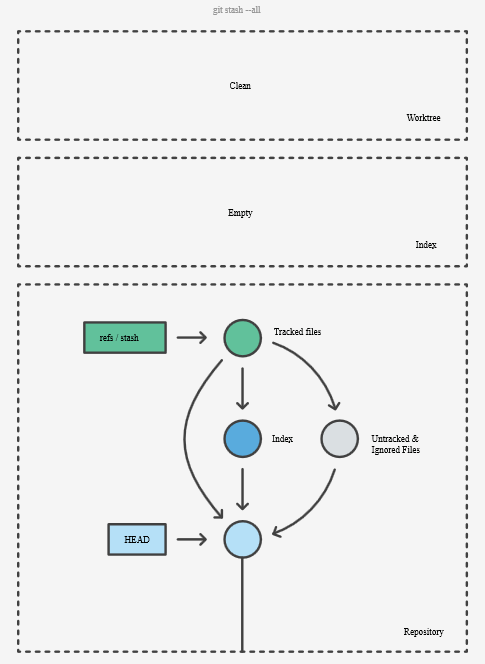
Invoking git stash encodes any changes to tracked files as two new commits in your DAG: one for unstaged changes, and one for changes staged in the index. The special refs/stash ref is updated to point to them.



Using the --include-untracked option also encodes any changes to untracked files as an additional commit.



Using the --all option includes changes to any ignored files alongside changes to untracked files in the same commit.



When you run git stash pop, the changes from the commits above are used to update your working copy and index, and the stash reflog is shuffled to remove the popped commit. Note that the popped commits aren't immediately deleted, but do become candidates for future garbage collection.

# .gitignore

[git add](https://www.atlassian.com/git/tutorials/saving-changes) [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) [git diff](https://www.atlassian.com/git/tutorials/saving-changes/git-diff) [git stash](https://www.atlassian.com/git/tutorials/saving-changes/git-stash) [.gitignore](https://www.atlassian.com/git/tutorials/saving-changes/gitignore)

Git sees every file in your working copy as one of three things:

1. tracked - a file which has been previously staged or committed;
2. untracked - a file which has not been staged or committed; or
3. ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

* dependency caches, such as the contents of /node\_modules or /packages
* compiled code, such as .o, .pyc, and .class files
* build output directories, such as /bin, /out, or /target
* files generated at runtime, such as .log, .lock, or .tmp
* hidden system files, such as .DS\_Store or Thumbs.db
* personal IDE config files, such as .idea/workspace.xml

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. .gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

* Ignoring files in Git
  + [Git ignore patterns](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#git-ignore-patterns)
  + [Shared .gitignore files in your repository](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#shared)
  + [Personal Git ignore rules](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#personal-git-ignore-rules)
  + [Global Git ignore rules](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#global-git-ignore-rules)
  + [Ignoring a previously committed file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#ignoring-a-previously-committed)
  + [Committing an ignored file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#committing-an-ignored-file)
  + [Stashing an ignored file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#stashing-an-ignored-file)
  + [Debugging .gitignore files](https://www.atlassian.com/git/tutorials/saving-changes/gitignore#debugging)

## Git ignore patterns

.gitignore uses [globbing patterns](http://linux.die.net/man/7/glob) to match against file names. You can construct your patterns using various symbols:

| **Pattern** | **Example matches** | **Explanation\*** |
| --- | --- | --- |
| \*\*/logs | logs/debug.log logs/monday/foo.bar build/logs/debug.log | You can prepend a pattern with a double asterisk to match directories anywhere in the repository. |
| \*\*/logs/debug.log | logs/debug.log build/logs/debug.log but not logs/build/debug.log | You can also use a double asterisk to match files based on their name and the name of their parent directory. |
| \*.log | debug.log foo.log .log logs/debug.log | An asterisk is a wildcard that matches zero or more characters. |
| \*.log  !important.log | debug.log trace.log but not important.log logs/important.log | Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but also matches a negating pattern defined later in the file, it will not be ignored. |
| \*.log  !important/\*.log trace.\* | debug.log important/trace.log but not important/debug.log | Patterns defined after a negating pattern will re-ignore any previously negated files. |
| /debug.log | debug.log but not logs/debug.log | Prepending a slash matches files only in the repository root. |
| debug.log | debug.log logs/debug.log | By default, patterns match files in any directory |
| debug?.log | debug0.log debugg.log but not debug10.log | A question mark matches exactly one character. |
| debug[0-9].log | debug0.log debug1.log but not debug10.log | Square brackets can also be used to match a single character from a specified range. |
| debug[01].log | debug0.log debug1.log but not  debug2.log debug01.log | Square brackets match a single character form the specified set. |
| debug[!01].log | debug2.log but not debug0.log debug1.log debug01.log | An exclamation mark can be used to match any character except one from the specified set. |
| debug[a-z].log | debuga.log debugb.log but not debug1.log | Ranges can be numeric or alphabetic. |
| logs | logs logs/debug.log logs/latest/foo.bar build/logs build/logs/debug.log | If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named logs are ignored |
| logs/ | logs/debug.log logs/latest/foo.bar build/logs/foo.bar build/logs/latest/debug.log | Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and subdirectories – will be ignored |
| logs/  !logs/important.log | logs/debug.log logs/important.log | Wait a minute! Shouldn't logs/important.log be negated in the example on the left  Nope! Due to a performance-related quirk in Git, you can not negate a file that is ignored due to a pattern matching a directory |
| logs/\*\*/debug.log | logs/debug.log logs/monday/debug.log logs/monday/pm/debug.log | A double asterisk matches zero or more directories. |
| logs/\*day/debug.log | logs/monday/debug.log logs/tuesday/debug.log but not logs/latest/debug.log | Wildcards can be used in directory names as well. |
| logs/debug.log | logs/debug.log but not debug.log build/logs/debug.log | Patterns specifying a file in a particular directory are relative to the repository root. (You can prepend a slash if you like, but it doesn't do anything special.) |

\*\* these explanations assume your .gitignore file is in the top level directory of your repository, as is the convention. If your repository has multiple .gitignore files, simply mentally replace "repository root" with "directory containing the .gitignore file" (and consider unifying them, for the sanity of your team).\*

In addition to these characters, you can use # to include comments in your .gitignore file:

# ignore all logs

\*.log

You can use \ to escape .gitignore pattern characters if you have files or directories containing them:

# ignore the file literally named foo[01].txt

foo\[01\].txt

## Shared .gitignore files in your repository

Git ignore rules are usually defined in a .gitignore file at the root of your repository. However, you can choose to define multiple .gitignore files in different directories in your repository. Each pattern in a particular .gitignore file is tested relative to the directory containing that file. However the convention, and simplest approach, is to define a single .gitignore file in the root. As your .gitignore file is checked in, it is versioned like any other file in your repository and shared with your teammates when you push. Typically you should only include patterns in .gitignore that will benefit other users of the repository.

## Personal Git ignore rules

You can also define personal ignore patterns for a particular repository in a special file at .git/info/exclude. These are not versioned, and not distributed with your repository, so it's an appropriate place to include patterns that will likely only benefit you. For example if you have a custom logging setup, or special development tools that produce files in your repository's working directory, you could consider adding them to .git/info/exclude to prevent them from being accidentally committed to your repository.

## Global Git ignore rules

In addition, you can define global Git ignore patterns for all repositories on your local system by setting the Git core.excludesFile property. You'll have to create this file yourself. If you're unsure where to put your global .gitignore file, your home directory isn't a bad choice (and makes it easy to find later). Once you've created the file, you'll need to configure its location with git config:

$ touch ~/.gitignore

$ git config --global core.excludesFile ~/.gitignore

You should be careful what patterns you choose to globally ignore, as different file types are relevant for different projects. Special operating system files (e.g. .DS\_Store and thumbs.db) or temporary files created by some developer tools are typical candidates for ignoring globally.

## Ignoring a previously committed file

If you want to ignore a file that you've committed in the past, you'll need to delete the file from your repository and then add a .gitignore rule for it. Using the --cached option with git rm means that the file will be deleted from your repository, but will remain in your working directory as an ignored file.

$ echo debug.log >> .gitignore

$ git rm --cached debug.log

rm 'debug.log'

$ git commit -m "Start ignoring debug.log"

You can omit the --cached option if you want to delete the file from both the repository and your local file system.

## Committing an ignored file

It is possible to force an ignored file to be committed to the repository using the -f (or --force) option with git add:

$ cat .gitignore

\*.log

$ git add -f debug.log

$ git commit -m "Force adding debug.log"

You might consider doing this if you have a general pattern (like \*.log) defined, but you want to commit a specific file. However a better solution is to define an exception to the general rule:

$ echo !debug.log >> .gitignore

$ cat .gitignore

\*.log

!debug.log

$ git add debug.log

$ git commit -m "Adding debug.log"

This approach is more obvious, and less confusing, for your teammates.

## Stashing an ignored file

[git stash](https://www.atlassian.com/git/tutorials/git-stash) is a powerful Git feature for temporarily shelving and reverting local changes, allowing you to re-apply them later on. As you'd expect, by default git stash ignores ignored files and only stashes changes to files that are tracked by Git. However, you can invoke [git stash with the --all option](https://www.atlassian.com/git/tutorials/git-stash/#stashing-untracked-or-ignored) to stash changes to ignored and untracked files as well.

## Debugging .gitignore files

If you have complicated .gitignore patterns, or patterns spread over multiple .gitignore files, it can be difficult to track down why a particular file is being ignored. You can use the git check-ignore command with the -v (or --verbose) option to determine which pattern is causing a particular file to be ignored:

$ git check-ignore -v debug.log

.gitignore:3:\*.log debug.log

The output shows:

<file containing the pattern> : <line number of the pattern> : <pattern> <file name>

You can pass multiple file names to git check-ignore if you like, and the names themselves don't even have to correspond to files that exist in your repository.

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.

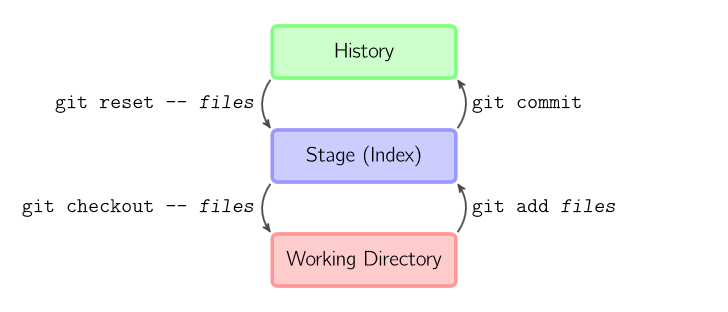
### git revert

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>

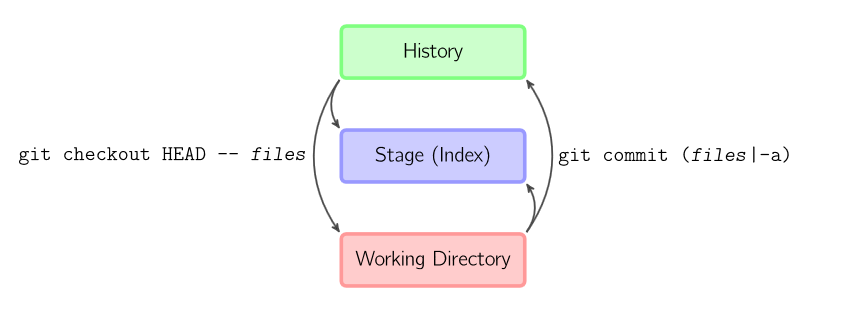


The four commands above copy files between the working directory, the stage (also called the index), and the history (in the form of commits).

* 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.

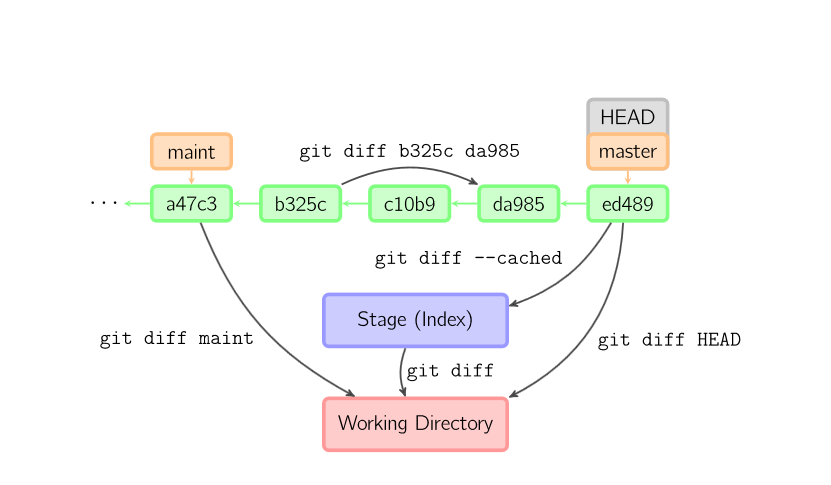


* git commit -a is equivalent to running git add on all filenames that existed in the latest commit, and then running git commit.
* git commit *files* creates a new commit containing the contents of the latest commit, plus a snapshot of *files* taken from the working directory. Additionally, *files*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

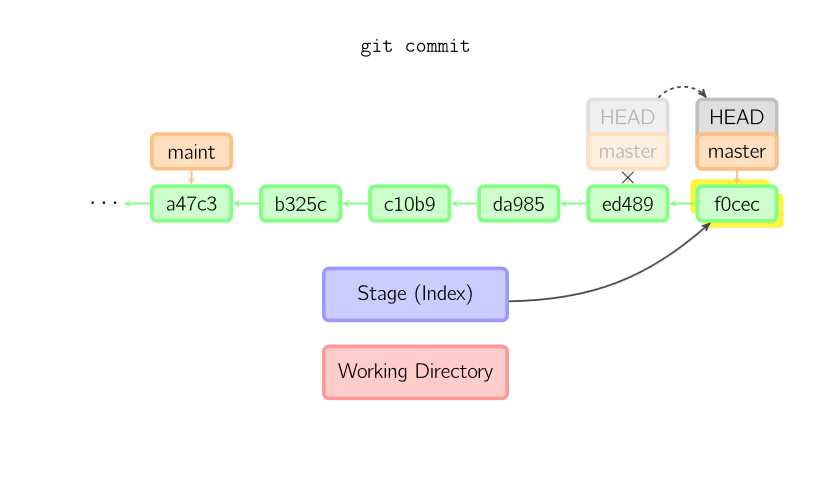
### Diff

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

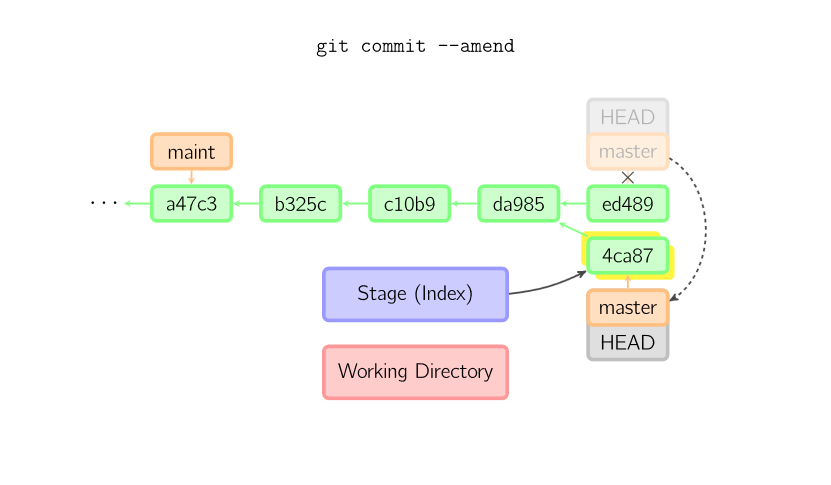


### Commit

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



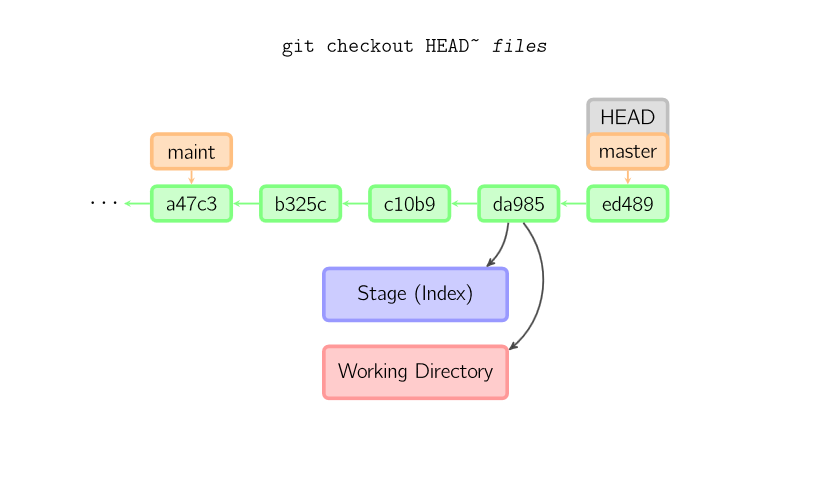
Sometimes a mistake is made in a commit, but this is easy to correct with git commit --amend. When you use this command, git creates a new commit with the same parent as the current commit. (The old commit will be discarded if nothing else references it.)



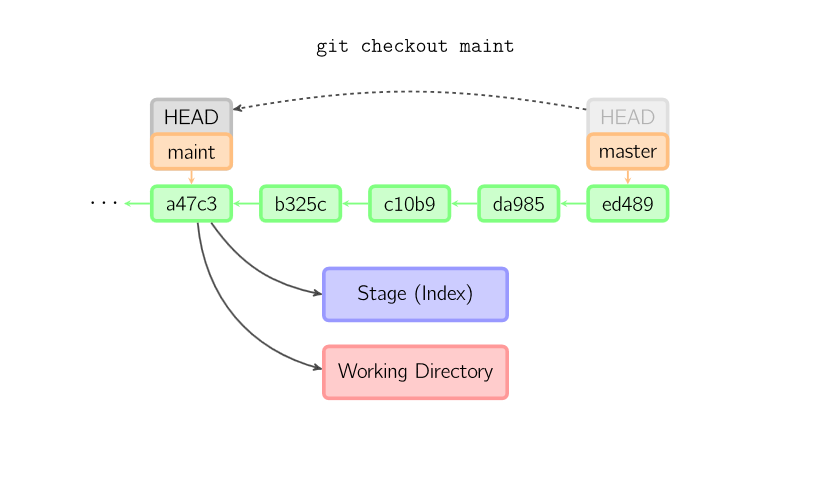
### Checkout

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

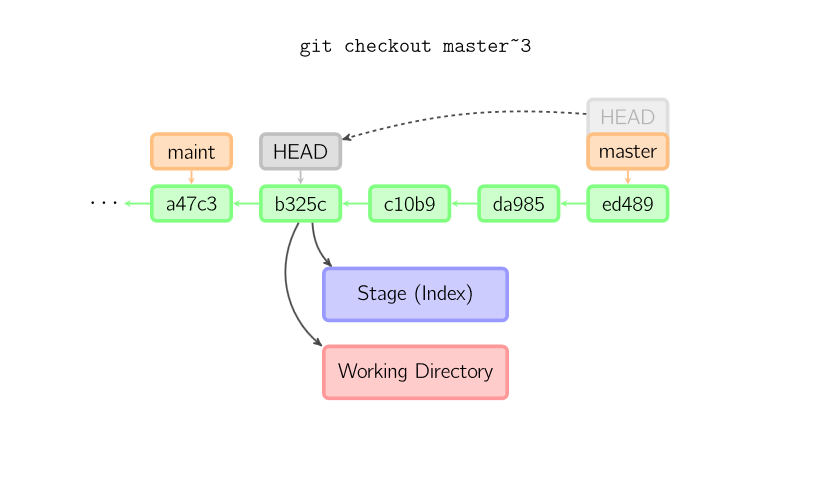
When a filename (and/or -p) is given, git copies those files from the given commit to the stage and the working directory. For example, git checkout HEAD~ foo.c copies the file foo.c from the commit called HEAD~ (the parent of the current commit) to the working directory, and also stages it. (If no commit 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.

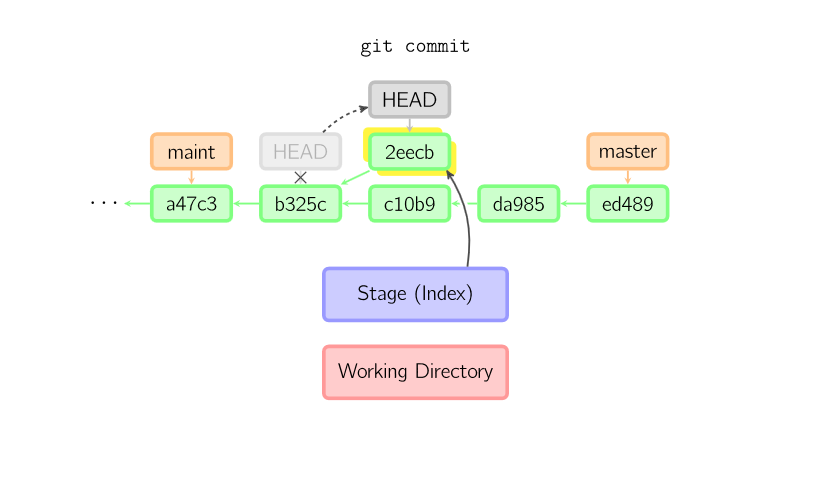


When a filename is not given and the reference is not a (local) branch — say, it is a tag, a remote branch, a SHA-1 ID, or something like master~3 — we get an anonymous branch, called a detached HEAD. This is useful for jumping around the history. Say you want to compile version 1.6.6.1 of git. You can 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).

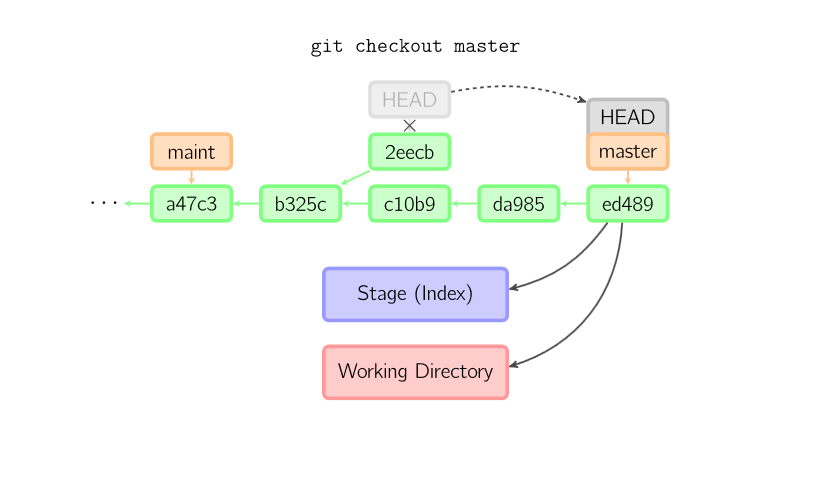


### Committing with a Detached HEAD

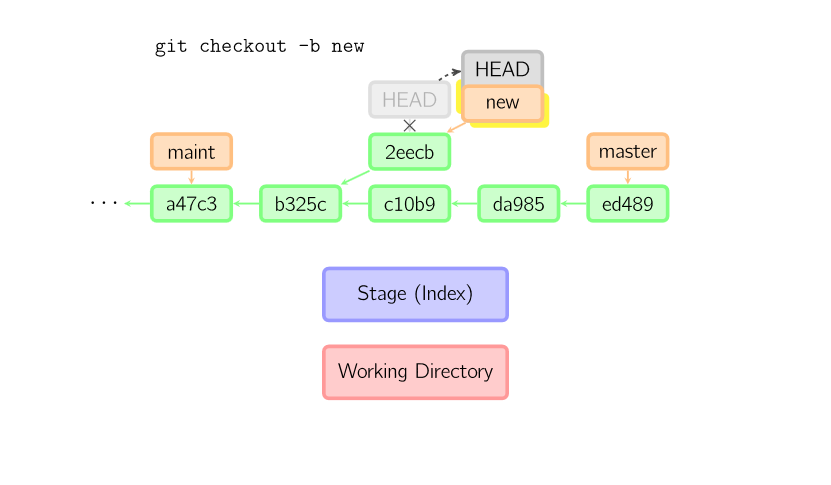
When HEAD is detached, commits work like normal, except no named branch gets updated. (You can think of this as an anonymous branch.)



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.



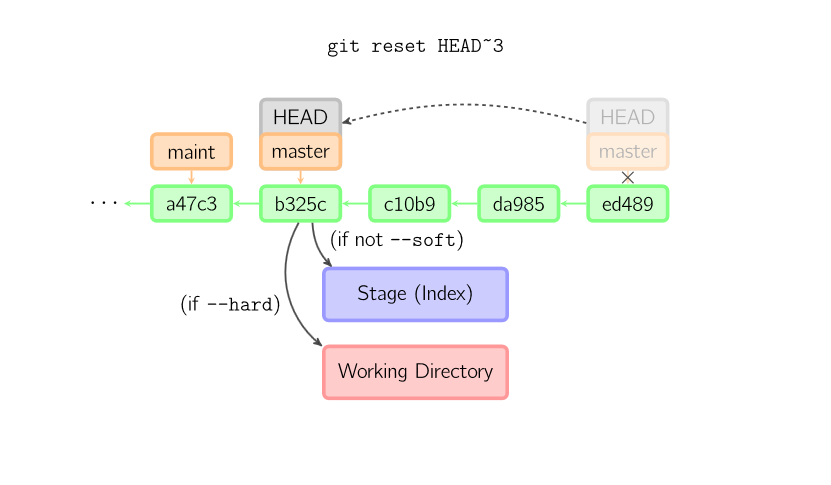
If, on the other hand, you want to save this state, you can create a new named branch using git checkout -b name



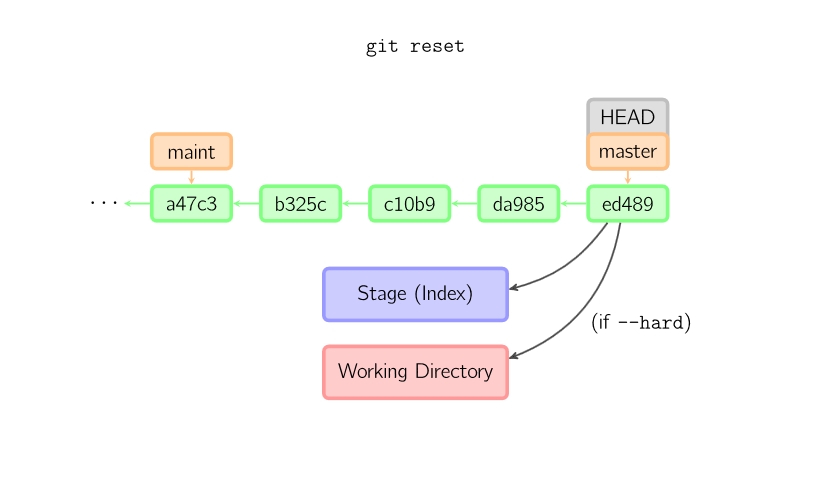
### Reset

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.

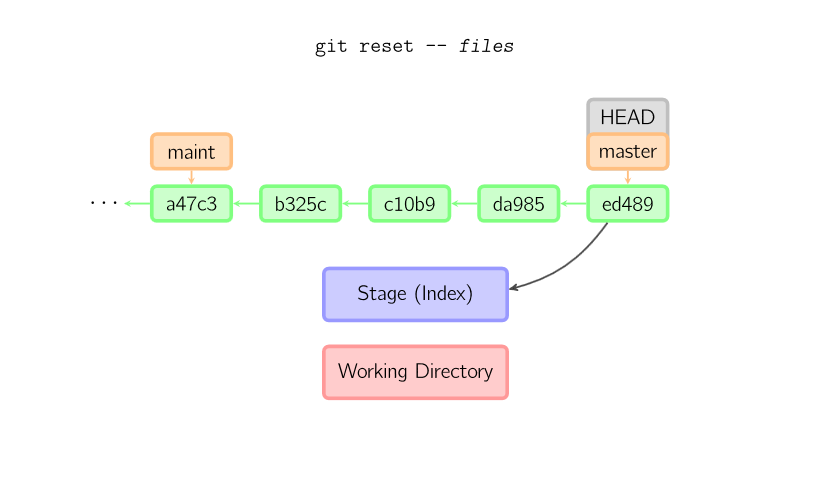
If a commit is given with no filenames, the current branch is moved to that commit, and then the stage is updated to match this commit. If --hard is given, the working directory is also updated. If --soft is given, neither is updated.



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

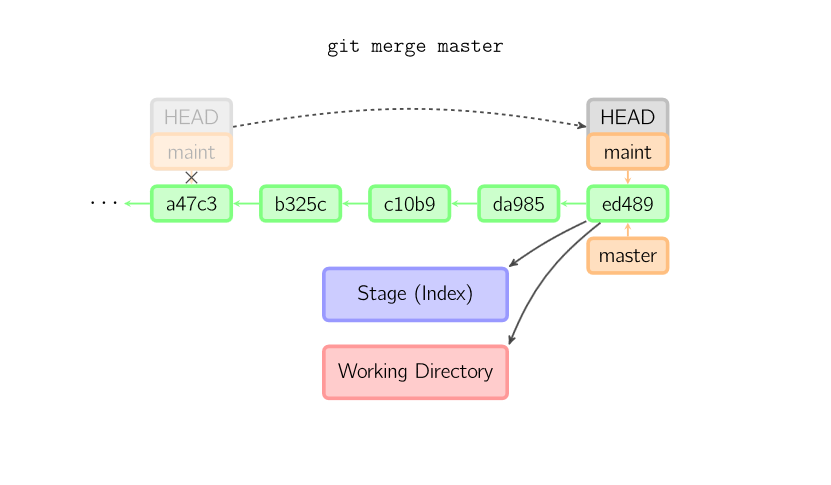


If a filename (and/or -p) is given, then the command works similarly to [checkout](http://marklodato.github.io/visual-git-guide/index-en.html#checkout) with a filename, except only the stage (and not the working directory) is updated. (You may also specify the commit from which to take files, rather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

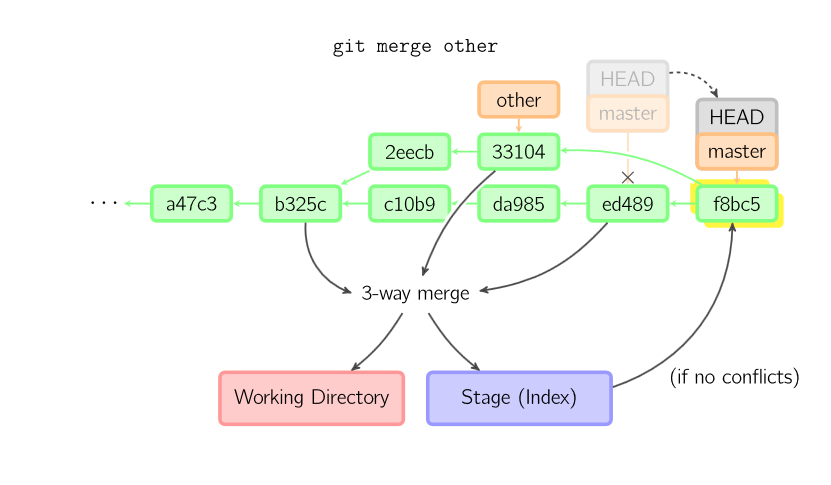


### Merge

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.

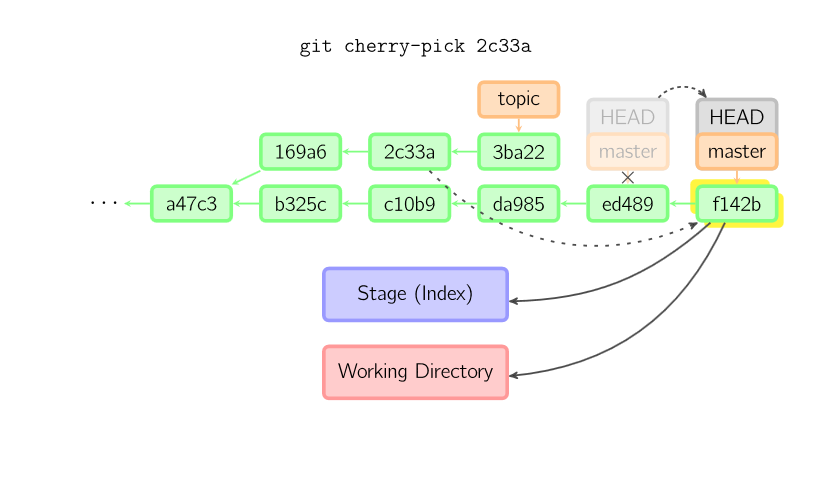


Otherwise, a "real" merge must occur. You can choose other strategies, but the default is to perform a "recursive" merge, which basically takes the current commit (ed489 below), the other commit (33104), and their common ancestor (b325c), and performs a [three-way merge](http://en.wikipedia.org/wiki/Three-way_merge). 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.



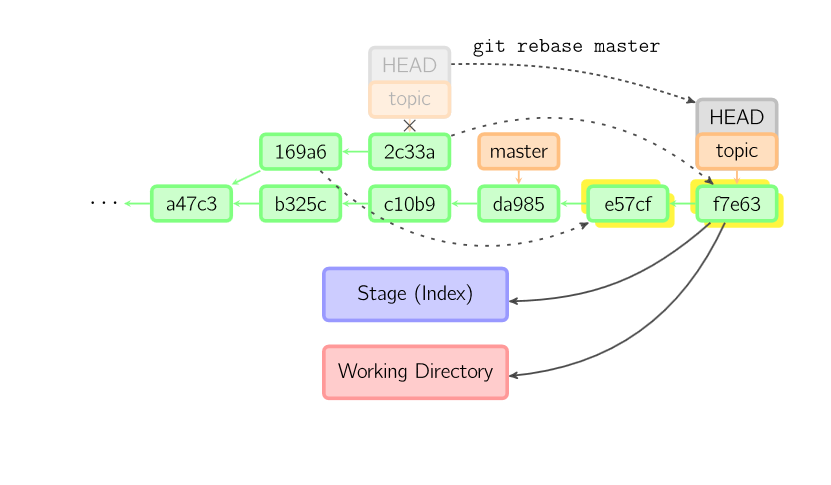
### Cherry Pick

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



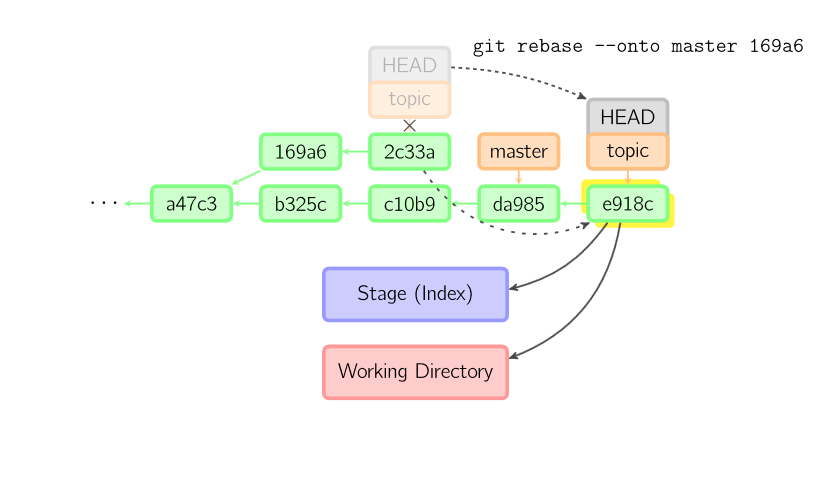
### Rebase

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.