[](https://git-scm.com/)--fast-version-control

Top of Form



Bottom of Form

* [**About**](https://git-scm.com/about)
* [**Documentation**](https://git-scm.com/doc)
  + [**Reference**](https://git-scm.com/docs)
  + [Book](https://git-scm.com/book)
  + [Videos](https://git-scm.com/videos)
  + [External Links](https://git-scm.com/doc/ext)
* [**Downloads**](https://git-scm.com/downloads)
* [**Community**](https://git-scm.com/community)

[**English ▾Topics ▾Version 2.29.0 ▾**](https://git-scm.com/docs/gitglossary)gitglossary last updated in 2.29.0

## NAME

gitglossary - A Git Glossary

## SYNOPSIS

\*

## DESCRIPTION

**alternate object database**

Via the alternates mechanism, a [repository](https://git-scm.com/docs/gitglossary#def_repository) can inherit part of its [object database](https://git-scm.com/docs/gitglossary#def_object_database) from another object database, which is called an "alternate".

**bare repository**

A bare repository is normally an appropriately named [directory](https://git-scm.com/docs/gitglossary#def_directory) with a .git suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the Git administrative and control files that would normally be present in the hidden .git sub-directory are directly present in the repository.git directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.

**blob object**

Untyped [object](https://git-scm.com/docs/gitglossary#def_object), e.g. the contents of a file.

**branch**

A "branch" is an active line of development. The most recent [commit](https://git-scm.com/docs/gitglossary#def_commit) on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch [head](https://git-scm.com/docs/gitglossary#def_head), which moves forward as additional development is done on the branch. A single Git [repository](https://git-scm.com/docs/gitglossary#def_repository) can track an arbitrary number of branches, but your [working tree](https://git-scm.com/docs/gitglossary#def_working_tree) is associated with just one of them (the "current" or "checked out" branch), and [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) points to that branch.

**cache**

Obsolete for: [index](https://git-scm.com/docs/gitglossary#def_index).

**chain**

A list of objects, where each [object](https://git-scm.com/docs/gitglossary#def_object) in the list contains a reference to its successor (for example, the successor of a [commit](https://git-scm.com/docs/gitglossary#def_commit) could be one of its [parents](https://git-scm.com/docs/gitglossary#def_parent)).

**changeset**

BitKeeper/cvsps speak for "[commit](https://git-scm.com/docs/gitglossary#def_commit)". Since Git does not store changes, but states, it really does not make sense to use the term "changesets" with Git.

**checkout**

The action of updating all or part of the [working tree](https://git-scm.com/docs/gitglossary#def_working_tree) with a [tree object](https://git-scm.com/docs/gitglossary#def_tree_object) or [blob](https://git-scm.com/docs/gitglossary#def_blob_object) from the [object database](https://git-scm.com/docs/gitglossary#def_object_database), and updating the [index](https://git-scm.com/docs/gitglossary#def_index) and [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) if the whole working tree has been pointed at a new [branch](https://git-scm.com/docs/gitglossary#def_branch).

**cherry-picking**

In [SCM](https://git-scm.com/docs/gitglossary#def_SCM) jargon, "cherry pick" means to choose a subset of changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In Git, this is performed by the "git cherry-pick" command to extract the change introduced by an existing [commit](https://git-scm.com/docs/gitglossary#def_commit) and to record it based on the tip of the current [branch](https://git-scm.com/docs/gitglossary#def_branch) as a new commit.

**clean**

A [working tree](https://git-scm.com/docs/gitglossary#def_working_tree) is clean, if it corresponds to the [revision](https://git-scm.com/docs/gitglossary#def_revision) referenced by the current [head](https://git-scm.com/docs/gitglossary#def_head). Also see "[dirty](https://git-scm.com/docs/gitglossary#def_dirty)".

**commit**

As a noun: A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for [commit object](https://git-scm.com/docs/gitglossary#def_commit_object).

As a verb: The action of storing a new snapshot of the project’s state in the Git history, by creating a new commit representing the current state of the [index](https://git-scm.com/docs/gitglossary#def_index) and advancing [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) to point at the new commit.

**commit object**

An [object](https://git-scm.com/docs/gitglossary#def_object) which contains the information about a particular [revision](https://git-scm.com/docs/gitglossary#def_revision), such as [parents](https://git-scm.com/docs/gitglossary#def_parent), committer, author, date and the [tree object](https://git-scm.com/docs/gitglossary#def_tree_object) which corresponds to the top [directory](https://git-scm.com/docs/gitglossary#def_directory) of the stored revision.

**commit-ish (also committish)**

A [commit object](https://git-scm.com/docs/gitglossary#def_commit_object) or an [object](https://git-scm.com/docs/gitglossary#def_object) that can be recursively dereferenced to a commit object. The following are all commit-ishes: a commit object, a [tag object](https://git-scm.com/docs/gitglossary#def_tag_object) that points to a commit object, a tag object that points to a tag object that points to a commit object, etc.

**core Git**

Fundamental data structures and utilities of Git. Exposes only limited source code management tools.

**DAG**

Directed acyclic graph. The [commit objects](https://git-scm.com/docs/gitglossary#def_commit_object) form a directed acyclic graph, because they have parents (directed), and the graph of commit objects is acyclic (there is no [chain](https://git-scm.com/docs/gitglossary#def_chain) which begins and ends with the same [object](https://git-scm.com/docs/gitglossary#def_object)).

**dangling object**

An [unreachable object](https://git-scm.com/docs/gitglossary#def_unreachable_object) which is not [reachable](https://git-scm.com/docs/gitglossary#def_reachable) even from other unreachable objects; a dangling object has no references to it from any reference or [object](https://git-scm.com/docs/gitglossary#def_object) in the [repository](https://git-scm.com/docs/gitglossary#def_repository).

**detached HEAD**

Normally the [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) stores the name of a [branch](https://git-scm.com/docs/gitglossary#def_branch), and commands that operate on the history HEAD represents operate on the history leading to the tip of the branch the HEAD points at. However, Git also allows you to [check out](https://git-scm.com/docs/gitglossary#def_checkout) an arbitrary [commit](https://git-scm.com/docs/gitglossary#def_commit) that isn’t necessarily the tip of any particular branch. The HEAD in such a state is called "detached".

Note that commands that operate on the history of the current branch (e.g. git commit to build a new history on top of it) still work while the HEAD is detached. They update the HEAD to point at the tip of the updated history without affecting any branch. Commands that update or inquire information **about** the current branch (e.g. git branch --set-upstream-to that sets what remote-tracking branch the current branch integrates with) obviously do not work, as there is no (real) current branch to ask about in this state.

**directory**

The list you get with "ls" :-)

**dirty**

A [working tree](https://git-scm.com/docs/gitglossary#def_working_tree) is said to be "dirty" if it contains modifications which have not been [committed](https://git-scm.com/docs/gitglossary#def_commit) to the current [branch](https://git-scm.com/docs/gitglossary#def_branch).

**evil merge**

An evil merge is a [merge](https://git-scm.com/docs/gitglossary#def_merge) that introduces changes that do not appear in any [parent](https://git-scm.com/docs/gitglossary#def_parent).

**fast-forward**

A fast-forward is a special type of [merge](https://git-scm.com/docs/gitglossary#def_merge) where you have a [revision](https://git-scm.com/docs/gitglossary#def_revision) and you are "merging" another [branch](https://git-scm.com/docs/gitglossary#def_branch)'s changes that happen to be a descendant of what you have. In such a case, you do not make a new [merge](https://git-scm.com/docs/gitglossary#def_merge) [commit](https://git-scm.com/docs/gitglossary#def_commit) but instead just update to his revision. This will happen frequently on a [remote-tracking branch](https://git-scm.com/docs/gitglossary#def_remote_tracking_branch) of a remote [repository](https://git-scm.com/docs/gitglossary#def_repository).

**fetch**

Fetching a [branch](https://git-scm.com/docs/gitglossary#def_branch) means to get the branch’s [head ref](https://git-scm.com/docs/gitglossary#def_head_ref) from a remote [repository](https://git-scm.com/docs/gitglossary#def_repository), to find out which objects are missing from the local [object database](https://git-scm.com/docs/gitglossary#def_object_database), and to get them, too. See also [git-fetch[1]](https://git-scm.com/docs/git-fetch).

**file system**

Linus Torvalds originally designed Git to be a user space file system, i.e. the infrastructure to hold files and directories. That ensured the efficiency and speed of Git.

**Git archive**

Synonym for [repository](https://git-scm.com/docs/gitglossary#def_repository) (for arch people).

**gitfile**

A plain file .git at the root of a working tree that points at the directory that is the real repository.

**grafts**

Grafts enables two otherwise different lines of development to be joined together by recording fake ancestry information for commits. This way you can make Git pretend the set of [parents](https://git-scm.com/docs/gitglossary#def_parent) a [commit](https://git-scm.com/docs/gitglossary#def_commit) has is different from what was recorded when the commit was created. Configured via the .git/info/grafts file.

Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see [git-replace[1]](https://git-scm.com/docs/git-replace) for a more flexible and robust system to do the same thing.

**hash**

In Git’s context, synonym for [object name](https://git-scm.com/docs/gitglossary#def_object_name).

**head**

A [named reference](https://git-scm.com/docs/gitglossary#def_ref) to the [commit](https://git-scm.com/docs/gitglossary#def_commit) at the tip of a [branch](https://git-scm.com/docs/gitglossary#def_branch). Heads are stored in a file in $GIT\_DIR/refs/heads/ directory, except when using packed refs. (See [git-pack-refs[1]](https://git-scm.com/docs/git-pack-refs).)

**HEAD**

The current [branch](https://git-scm.com/docs/gitglossary#def_branch). In more detail: Your [working tree](https://git-scm.com/docs/gitglossary#def_working_tree) is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the [heads](https://git-scm.com/docs/gitglossary#def_head) in your repository, except when using a [detached HEAD](https://git-scm.com/docs/gitglossary#def_detached_HEAD), in which case it directly references an arbitrary commit.

**head ref**

A synonym for [head](https://git-scm.com/docs/gitglossary#def_head).

**hook**

During the normal execution of several Git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the $GIT\_DIR/hooks/ directory, and are enabled by simply removing the .sample suffix from the filename. In earlier versions of Git you had to make them executable.

**index**

A collection of files with stat information, whose contents are stored as objects. The index is a stored version of your [working tree](https://git-scm.com/docs/gitglossary#def_working_tree). Truth be told, it can also contain a second, and even a third version of a working tree, which are used when [merging](https://git-scm.com/docs/gitglossary#def_merge).

**index entry**

The information regarding a particular file, stored in the [index](https://git-scm.com/docs/gitglossary#def_index). An index entry can be unmerged, if a [merge](https://git-scm.com/docs/gitglossary#def_merge) was started, but not yet finished (i.e. if the index contains multiple versions of that file).

**master**

The default development [branch](https://git-scm.com/docs/gitglossary#def_branch). Whenever you create a Git [repository](https://git-scm.com/docs/gitglossary#def_repository), a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.

**merge**

As a verb: To bring the contents of another [branch](https://git-scm.com/docs/gitglossary#def_branch) (possibly from an external [repository](https://git-scm.com/docs/gitglossary#def_repository)) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first [fetching](https://git-scm.com/docs/gitglossary#def_fetch) the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a [pull](https://git-scm.com/docs/gitglossary#def_pull). Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.

As a noun: unless it is a [fast-forward](https://git-scm.com/docs/gitglossary#def_fast_forward), a successful merge results in the creation of a new [commit](https://git-scm.com/docs/gitglossary#def_commit) representing the result of the merge, and having as [parents](https://git-scm.com/docs/gitglossary#def_parent) the tips of the merged [branches](https://git-scm.com/docs/gitglossary#def_branch). This commit is referred to as a "merge commit", or sometimes just a "merge".

**object**

The unit of storage in Git. It is uniquely identified by the [SHA-1](https://git-scm.com/docs/gitglossary#def_SHA1) of its contents. Consequently, an object cannot be changed.

**object database**

Stores a set of "objects", and an individual [object](https://git-scm.com/docs/gitglossary#def_object) is identified by its [object name](https://git-scm.com/docs/gitglossary#def_object_name). The objects usually live in $GIT\_DIR/objects/.

**object identifier**

Synonym for [object name](https://git-scm.com/docs/gitglossary#def_object_name).

**object name**

The unique identifier of an [object](https://git-scm.com/docs/gitglossary#def_object). The object name is usually represented by a 40 character hexadecimal string. Also colloquially called [SHA-1](https://git-scm.com/docs/gitglossary#def_SHA1).

**object type**

One of the identifiers "[commit](https://git-scm.com/docs/gitglossary#def_commit_object)", "[tree](https://git-scm.com/docs/gitglossary#def_tree_object)", "[tag](https://git-scm.com/docs/gitglossary#def_tag_object)" or "[blob](https://git-scm.com/docs/gitglossary#def_blob_object)" describing the type of an [object](https://git-scm.com/docs/gitglossary#def_object).

**octopus**

To [merge](https://git-scm.com/docs/gitglossary#def_merge) more than two [branches](https://git-scm.com/docs/gitglossary#def_branch).

**origin**

The default upstream [repository](https://git-scm.com/docs/gitglossary#def_repository). Most projects have at least one upstream project which they track. By default **origin** is used for that purpose. New upstream updates will be fetched into [remote-tracking branches](https://git-scm.com/docs/gitglossary#def_remote_tracking_branch) named origin/name-of-upstream-branch, which you can see using git branch -r.

**overlay**

Only update and add files to the working directory, but don’t delete them, similar to how **cp -R** would update the contents in the destination directory. This is the default mode in a [checkout](https://git-scm.com/docs/gitglossary#def_checkout) when checking out files from the [index](https://git-scm.com/docs/gitglossary#def_index) or a [tree-ish](https://git-scm.com/docs/gitglossary#def_tree-ish). In contrast, no-overlay mode also deletes tracked files not present in the source, similar to **rsync --delete**.

**pack**

A set of objects which have been compressed into one file (to save space or to transmit them efficiently).

**pack index**

The list of identifiers, and other information, of the objects in a [pack](https://git-scm.com/docs/gitglossary#def_pack), to assist in efficiently accessing the contents of a pack.

**pathspec**

Pattern used to limit paths in Git commands.

Pathspecs are used on the command line of "git ls-files", "git ls-tree", "git add", "git grep", "git diff", "git checkout", and many other commands to limit the scope of operations to some subset of the tree or worktree. See the documentation of each command for whether paths are relative to the current directory or toplevel. The pathspec syntax is as follows:

* any path matches itself
* the pathspec up to the last slash represents a directory prefix. The scope of that pathspec is limited to that subtree.
* the rest of the pathspec is a pattern for the remainder of the pathname. Paths relative to the directory prefix will be matched against that pattern using fnmatch(3); in particular, **\*** and **?** **can** match directory separators.

For example, Documentation/\*.jpg will match all .jpg files in the Documentation subtree, including Documentation/chapter\_1/figure\_1.jpg.

A pathspec that begins with a colon : has special meaning. In the short form, the leading colon : is followed by zero or more "magic signature" letters (which optionally is terminated by another colon :), and the remainder is the pattern to match against the path. The "magic signature" consists of ASCII symbols that are neither alphanumeric, glob, regex special characters nor colon. The optional colon that terminates the "magic signature" can be omitted if the pattern begins with a character that does not belong to "magic signature" symbol set and is not a colon.

In the long form, the leading colon : is followed by an open parenthesis (, a comma-separated list of zero or more "magic words", and a close parentheses ), and the remainder is the pattern to match against the path.

A pathspec with only a colon means "there is no pathspec". This form should not be combined with other pathspec.

**top**

The magic word top (magic signature: /) makes the pattern match from the root of the working tree, even when you are running the command from inside a subdirectory.

**literal**

Wildcards in the pattern such as \* or ? are treated as literal characters.

**icase**

Case insensitive match.

**glob**

Git treats the pattern as a shell glob suitable for consumption by fnmatch(3) with the FNM\_PATHNAME flag: wildcards in the pattern will not match a / in the pathname. For example, "Documentation/\*.html" matches "Documentation/git.html" but not "Documentation/ppc/ppc.html" or "tools/perf/Documentation/perf.html".

Two consecutive asterisks ("\*\*") in patterns matched against full pathname may have special meaning:

* A leading "\*\*" followed by a slash means match in all directories. For example, "\*\*/foo" matches file or directory "foo" anywhere, the same as pattern "foo". "\*\*/foo/bar" matches file or directory "bar" anywhere that is directly under directory "foo".
* A trailing "/\*\*" matches everything inside. For example, "abc/\*\*" matches all files inside directory "abc", relative to the location of the .gitignore file, with infinite depth.
* A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "a/\*\*/b" matches "a/b", "a/x/b", "a/x/y/b" and so on.
* Other consecutive asterisks are considered invalid.

Glob magic is incompatible with literal magic.

**attr**

After attr: comes a space separated list of "attribute requirements", all of which must be met in order for the path to be considered a match; this is in addition to the usual non-magic pathspec pattern matching. See [gitattributes[5]](https://git-scm.com/docs/gitattributes).

Each of the attribute requirements for the path takes one of these forms:

* "ATTR" requires that the attribute ATTR be set.
* "-ATTR" requires that the attribute ATTR be unset.
* "ATTR=VALUE" requires that the attribute ATTR be set to the string VALUE.
* "!ATTR" requires that the attribute ATTR be unspecified.

Note that when matching against a tree object, attributes are still obtained from working tree, not from the given tree object.

**exclude**

After a path matches any non-exclude pathspec, it will be run through all exclude pathspecs (magic signature: ! or its synonym ^). If it matches, the path is ignored. When there is no non-exclude pathspec, the exclusion is applied to the result set as if invoked without any pathspec.

**parent**

A [commit object](https://git-scm.com/docs/gitglossary#def_commit_object) contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.

**pickaxe**

The term [pickaxe](https://git-scm.com/docs/gitglossary#def_pickaxe) refers to an option to the diffcore routines that help select changes that add or delete a given text string. With the --pickaxe-all option, it can be used to view the full [changeset](https://git-scm.com/docs/gitglossary#def_changeset) that introduced or removed, say, a particular line of text. See [git-diff[1]](https://git-scm.com/docs/git-diff).

**plumbing**

Cute name for [core Git](https://git-scm.com/docs/gitglossary#def_core_git).

**porcelain**

Cute name for programs and program suites depending on [core Git](https://git-scm.com/docs/gitglossary#def_core_git), presenting a high level access to core Git. Porcelains expose more of a [SCM](https://git-scm.com/docs/gitglossary#def_SCM) interface than the [plumbing](https://git-scm.com/docs/gitglossary#def_plumbing).

**per-worktree ref**

Refs that are per-[worktree](https://git-scm.com/docs/gitglossary#def_working_tree), rather than global. This is presently only [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) and any refs that start with refs/bisect/, but might later include other unusual refs.

**pseudoref**

Pseudorefs are a class of files under $GIT\_DIR which behave like refs for the purposes of rev-parse, but which are treated specially by git. Pseudorefs both have names that are all-caps, and always start with a line consisting of a [SHA-1](https://git-scm.com/docs/gitglossary#def_SHA1) followed by whitespace. So, HEAD is not a pseudoref, because it is sometimes a symbolic ref. They might optionally contain some additional data. MERGE\_HEAD and CHERRY\_PICK\_HEAD are examples. Unlike [per-worktree refs](https://git-scm.com/docs/gitglossary#def_per_worktree_ref), these files cannot be symbolic refs, and never have reflogs. They also cannot be updated through the normal ref update machinery. Instead, they are updated by directly writing to the files. However, they can be read as if they were refs, so git rev-parse MERGE\_HEAD will work.

**pull**

Pulling a [branch](https://git-scm.com/docs/gitglossary#def_branch) means to [fetch](https://git-scm.com/docs/gitglossary#def_fetch) it and [merge](https://git-scm.com/docs/gitglossary#def_merge) it. See also [git-pull[1]](https://git-scm.com/docs/git-pull).

**push**

Pushing a [branch](https://git-scm.com/docs/gitglossary#def_branch) means to get the branch’s [head ref](https://git-scm.com/docs/gitglossary#def_head_ref) from a remote [repository](https://git-scm.com/docs/gitglossary#def_repository), find out if it is an ancestor to the branch’s local head ref, and in that case, putting all objects, which are [reachable](https://git-scm.com/docs/gitglossary#def_reachable) from the local head ref, and which are missing from the remote repository, into the remote [object database](https://git-scm.com/docs/gitglossary#def_object_database), and updating the remote head ref. If the remote [head](https://git-scm.com/docs/gitglossary#def_head) is not an ancestor to the local head, the push fails.

**reachable**

All of the ancestors of a given [commit](https://git-scm.com/docs/gitglossary#def_commit) are said to be "reachable" from that commit. More generally, one [object](https://git-scm.com/docs/gitglossary#def_object) is reachable from another if we can reach the one from the other by a [chain](https://git-scm.com/docs/gitglossary#def_chain) that follows [tags](https://git-scm.com/docs/gitglossary#def_tag) to whatever they tag, [commits](https://git-scm.com/docs/gitglossary#def_commit_object) to their parents or trees, and [trees](https://git-scm.com/docs/gitglossary#def_tree_object) to the trees or [blobs](https://git-scm.com/docs/gitglossary#def_blob_object) that they contain.

**rebase**

To reapply a series of changes from a [branch](https://git-scm.com/docs/gitglossary#def_branch) to a different base, and reset the [head](https://git-scm.com/docs/gitglossary#def_head) of that branch to the result.

**ref**

A name that begins with refs/ (e.g. refs/heads/master) that points to an [object name](https://git-scm.com/docs/gitglossary#def_object_name) or another ref (the latter is called a [symbolic ref](https://git-scm.com/docs/gitglossary#def_symref)). For convenience, a ref can sometimes be abbreviated when used as an argument to a Git command; see [gitrevisions[7]](https://git-scm.com/docs/gitrevisions) for details. Refs are stored in the [repository](https://git-scm.com/docs/gitglossary#def_repository).

The ref namespace is hierarchical. Different subhierarchies are used for different purposes (e.g. the refs/heads/ hierarchy is used to represent local branches).

There are a few special-purpose refs that do not begin with refs/. The most notable example is HEAD.

**reflog**

A reflog shows the local "history" of a ref. In other words, it can tell you what the 3rd last revision in **this** repository was, and what was the current state in **this** repository, yesterday 9:14pm. See [git-reflog[1]](https://git-scm.com/docs/git-reflog) for details.

**refspec**

A "refspec" is used by [fetch](https://git-scm.com/docs/gitglossary#def_fetch) and [push](https://git-scm.com/docs/gitglossary#def_push) to describe the mapping between remote [ref](https://git-scm.com/docs/gitglossary#def_ref) and local ref.

**remote repository**

A [repository](https://git-scm.com/docs/gitglossary#def_repository) which is used to track the same project but resides somewhere else. To communicate with remotes, see [fetch](https://git-scm.com/docs/gitglossary#def_fetch) or [push](https://git-scm.com/docs/gitglossary#def_push).

**remote-tracking branch**

A [ref](https://git-scm.com/docs/gitglossary#def_ref) that is used to follow changes from another [repository](https://git-scm.com/docs/gitglossary#def_repository). It typically looks like **refs/remotes/foo/bar** (indicating that it tracks a branch named **bar** in a remote named **foo**), and matches the right-hand-side of a configured fetch [refspec](https://git-scm.com/docs/gitglossary#def_refspec). A remote-tracking branch should not contain direct modifications or have local commits made to it.

**repository**

A collection of [refs](https://git-scm.com/docs/gitglossary#def_ref) together with an [object database](https://git-scm.com/docs/gitglossary#def_object_database) containing all objects which are [reachable](https://git-scm.com/docs/gitglossary#def_reachable) from the refs, possibly accompanied by meta data from one or more [porcelains](https://git-scm.com/docs/gitglossary#def_porcelain). A repository can share an object database with other repositories via [alternates mechanism](https://git-scm.com/docs/gitglossary#def_alternate_object_database).

**resolve**

The action of fixing up manually what a failed automatic [merge](https://git-scm.com/docs/gitglossary#def_merge) left behind.

**revision**

Synonym for [commit](https://git-scm.com/docs/gitglossary#def_commit) (the noun).

**rewind**

To throw away part of the development, i.e. to assign the [head](https://git-scm.com/docs/gitglossary#def_head) to an earlier [revision](https://git-scm.com/docs/gitglossary#def_revision).

**SCM**

Source code management (tool).

**SHA-1**

"Secure Hash Algorithm 1"; a cryptographic hash function. In the context of Git used as a synonym for [object name](https://git-scm.com/docs/gitglossary#def_object_name).

**shallow clone**

Mostly a synonym to [shallow repository](https://git-scm.com/docs/gitglossary#def_shallow_repository) but the phrase makes it more explicit that it was created by running git clone --depth=... command.

**shallow repository**

A shallow [repository](https://git-scm.com/docs/gitglossary#def_repository) has an incomplete history some of whose [commits](https://git-scm.com/docs/gitglossary#def_commit) have [parents](https://git-scm.com/docs/gitglossary#def_parent) cauterized away (in other words, Git is told to pretend that these commits do not have the parents, even though they are recorded in the [commit object](https://git-scm.com/docs/gitglossary#def_commit_object)). This is sometimes useful when you are interested only in the recent history of a project even though the real history recorded in the upstream is much larger. A shallow repository is created by giving the --depth option to [git-clone[1]](https://git-scm.com/docs/git-clone), and its history can be later deepened with [git-fetch[1]](https://git-scm.com/docs/git-fetch).

**stash entry**

An [object](https://git-scm.com/docs/gitglossary#def_object) used to temporarily store the contents of a [dirty](https://git-scm.com/docs/gitglossary#def_dirty) working directory and the index for future reuse.

**submodule**

A [repository](https://git-scm.com/docs/gitglossary#def_repository) that holds the history of a separate project inside another repository (the latter of which is called [superproject](https://git-scm.com/docs/gitglossary#def_superproject)).

**superproject**

A [repository](https://git-scm.com/docs/gitglossary#def_repository) that references repositories of other projects in its working tree as [submodules](https://git-scm.com/docs/gitglossary#def_submodule). The superproject knows about the names of (but does not hold copies of) commit objects of the contained submodules.

**symref**

Symbolic reference: instead of containing the [SHA-1](https://git-scm.com/docs/gitglossary#def_SHA1) id itself, it is of the format **ref: refs/some/thing** and when referenced, it recursively dereferences to this reference. [**HEAD**](https://git-scm.com/docs/gitglossary#def_HEAD) is a prime example of a symref. Symbolic references are manipulated with the [git-symbolic-ref[1]](https://git-scm.com/docs/git-symbolic-ref) command.

**tag**

A [ref](https://git-scm.com/docs/gitglossary#def_ref) under refs/tags/ namespace that points to an object of an arbitrary type (typically a tag points to either a [tag](https://git-scm.com/docs/gitglossary#def_tag_object) or a [commit object](https://git-scm.com/docs/gitglossary#def_commit_object)). In contrast to a [head](https://git-scm.com/docs/gitglossary#def_head), a tag is not updated by the commit command. A Git tag has nothing to do with a Lisp tag (which would be called an [object type](https://git-scm.com/docs/gitglossary#def_object_type) in Git’s context). A tag is most typically used to mark a particular point in the commit ancestry [chain](https://git-scm.com/docs/gitglossary#def_chain).

**tag object**

An [object](https://git-scm.com/docs/gitglossary#def_object) containing a [ref](https://git-scm.com/docs/gitglossary#def_ref) pointing to another object, which can contain a message just like a [commit object](https://git-scm.com/docs/gitglossary#def_commit_object). It can also contain a (PGP) signature, in which case it is called a "signed tag object".

**topic branch**

A regular Git [branch](https://git-scm.com/docs/gitglossary#def_branch) that is used by a developer to identify a conceptual line of development. Since branches are very easy and inexpensive, it is often desirable to have several small branches that each contain very well defined concepts or small incremental yet related changes.

**tree**

Either a [working tree](https://git-scm.com/docs/gitglossary#def_working_tree), or a [tree object](https://git-scm.com/docs/gitglossary#def_tree_object) together with the dependent [blob](https://git-scm.com/docs/gitglossary#def_blob_object) and tree objects (i.e. a stored representation of a working tree).

**tree object**

An [object](https://git-scm.com/docs/gitglossary#def_object) containing a list of file names and modes along with refs to the associated blob and/or tree objects. A [tree](https://git-scm.com/docs/gitglossary#def_tree) is equivalent to a [directory](https://git-scm.com/docs/gitglossary#def_directory).

**tree-ish (also treeish)**

A [tree object](https://git-scm.com/docs/gitglossary#def_tree_object) or an [object](https://git-scm.com/docs/gitglossary#def_object) that can be recursively dereferenced to a tree object. Dereferencing a [commit object](https://git-scm.com/docs/gitglossary#def_commit_object) yields the tree object corresponding to the [revision](https://git-scm.com/docs/gitglossary#def_revision)'s top [directory](https://git-scm.com/docs/gitglossary#def_directory). The following are all tree-ishes: a [commit-ish](https://git-scm.com/docs/gitglossary#def_commit-ish), a tree object, a [tag object](https://git-scm.com/docs/gitglossary#def_tag_object) that points to a tree object, a tag object that points to a tag object that points to a tree object, etc.

**unmerged index**

An [index](https://git-scm.com/docs/gitglossary#def_index) which contains unmerged [index entries](https://git-scm.com/docs/gitglossary#def_index_entry).

**unreachable object**

An [object](https://git-scm.com/docs/gitglossary#def_object) which is not [reachable](https://git-scm.com/docs/gitglossary#def_reachable) from a [branch](https://git-scm.com/docs/gitglossary#def_branch), [tag](https://git-scm.com/docs/gitglossary#def_tag), or any other reference.

**upstream branch**

The default [branch](https://git-scm.com/docs/gitglossary#def_branch) that is merged into the branch in question (or the branch in question is rebased onto). It is configured via branch.<name>.remote and branch.<name>.merge. If the upstream branch of **A** is **origin/B** sometimes we say "**A** is tracking **origin/B**".

**working tree**

The tree of actual checked out files. The working tree normally contains the contents of the [HEAD](https://git-scm.com/docs/gitglossary#def_HEAD) commit’s tree, plus any local changes that you have made but not yet committed.

## SEE ALSO

[gittutorial[7]](https://git-scm.com/docs/gittutorial), [gittutorial-2[7]](https://git-scm.com/docs/gittutorial-2), [gitcvs-migration[7]](https://git-scm.com/docs/gitcvs-migration), [giteveryday[7]](https://git-scm.com/docs/giteveryday), [The Git User’s Manual](https://git-scm.com/docs/user-manual)

## GIT

Part of the [git[1]](https://git-scm.com/docs/git) suite

[About this site](https://git-scm.com/site)  
Patches, suggestions, and comments are welcome.

Git is a member of [Software Freedom Conservancy](https://git-scm.com/sfc)

[](https://git-scm.com/)--everything-is-local

Top of Form



Bottom of Form

* [**About**](https://git-scm.com/about)
* [**Documentation**](https://git-scm.com/doc)
  + [**Reference**](https://git-scm.com/docs)
  + [Book](https://git-scm.com/book)
  + [Videos](https://git-scm.com/videos)
  + [External Links](https://git-scm.com/doc/ext)
* [**Downloads**](https://git-scm.com/downloads)
* [**Community**](https://git-scm.com/community)

[**English ▾Topics ▾Version 2.29.2 ▾**](https://git-scm.com/docs/gitattributes)gitattributes last updated in 2.29.2

**NAME**

gitattributes - Defining attributes per path

**SYNOPSIS**

$GIT\_DIR/info/attributes, .gitattributes

**DESCRIPTION**

A gitattributes file is a simple text file that gives attributes to pathnames.

Each line in gitattributes file is of form:

pattern attr1 attr2 ...

That is, a pattern followed by an attributes list, separated by whitespaces. Leading and trailing whitespaces are ignored. Lines that begin with **#** are ignored. Patterns that begin with a double quote are quoted in C style. When the pattern matches the path in question, the attributes listed on the line are given to the path.

Each attribute can be in one of these states for a given path:

**Set**

The path has the attribute with special value "true"; this is specified by listing only the name of the attribute in the attribute list.

**Unset**

The path has the attribute with special value "false"; this is specified by listing the name of the attribute prefixed with a dash - in the attribute list.

**Set to a value**

The path has the attribute with specified string value; this is specified by listing the name of the attribute followed by an equal sign = and its value in the attribute list.

**Unspecified**

No pattern matches the path, and nothing says if the path has or does not have the attribute, the attribute for the path is said to be Unspecified.

When more than one pattern matches the path, a later line overrides an earlier line. This overriding is done per attribute.

The rules by which the pattern matches paths are the same as in .gitignore files (see [gitignore[5]](https://git-scm.com/docs/gitignore)), with a few exceptions:

* negative patterns are forbidden
* patterns that match a directory do not recursively match paths inside that directory (so using the trailing-slash path/ syntax is pointless in an attributes file; use path/\*\* instead)

When deciding what attributes are assigned to a path, Git consults $GIT\_DIR/info/attributes file (which has the highest precedence), .gitattributes file in the same directory as the path in question, and its parent directories up to the toplevel of the work tree (the further the directory that contains .gitattributes is from the path in question, the lower its precedence). Finally global and system-wide files are considered (they have the lowest precedence).

When the .gitattributes file is missing from the work tree, the path in the index is used as a fall-back. During checkout process, .gitattributes in the index is used and then the file in the working tree is used as a fall-back.

If you wish to affect only a single repository (i.e., to assign attributes to files that are particular to one user’s workflow for that repository), then attributes should be placed in the $GIT\_DIR/info/attributes file. Attributes which should be version-controlled and distributed to other repositories (i.e., attributes of interest to all users) should go into .gitattributes files. Attributes that should affect all repositories for a single user should be placed in a file specified by the core.attributesFile configuration option (see [git-config[1]](https://git-scm.com/docs/git-config)). Its default value is $XDG\_CONFIG\_HOME/git/attributes. If $XDG\_CONFIG\_HOME is either not set or empty, $HOME/.config/git/attributes is used instead. Attributes for all users on a system should be placed in the $(prefix)/etc/gitattributes file.

Sometimes you would need to override a setting of an attribute for a path to Unspecified state. This can be done by listing the name of the attribute prefixed with an exclamation point !.

**EFFECTS**

Certain operations by Git can be influenced by assigning particular attributes to a path. Currently, the following operations are attributes-aware.

**Checking-out and checking-in**

These attributes affect how the contents stored in the repository are copied to the working tree files when commands such as **git switch**, **git checkout** and **git merge** run. They also affect how Git stores the contents you prepare in the working tree in the repository upon **git add** and **git commit**.

**text**

This attribute enables and controls end-of-line normalization. When a text file is normalized, its line endings are converted to LF in the repository. To control what line ending style is used in the working directory, use the eol attribute for a single file and the core.eol configuration variable for all text files. Note that setting core.autocrlf to true or input overrides core.eol (see the definitions of those options in [git-config[1]](https://git-scm.com/docs/git-config)).

**Set**

Setting the text attribute on a path enables end-of-line normalization and marks the path as a text file. End-of-line conversion takes place without guessing the content type.

**Unset**

Unsetting the text attribute on a path tells Git not to attempt any end-of-line conversion upon checkin or checkout.

**Set to string value "auto"**

When text is set to "auto", the path is marked for automatic end-of-line conversion. If Git decides that the content is text, its line endings are converted to LF on checkin. When the file has been committed with CRLF, no conversion is done.

**Unspecified**

If the text attribute is unspecified, Git uses the core.autocrlf configuration variable to determine if the file should be converted.

Any other value causes Git to act as if text has been left unspecified.

**eol**

This attribute sets a specific line-ending style to be used in the working directory. It enables end-of-line conversion without any content checks, effectively setting the text attribute. Note that setting this attribute on paths which are in the index with CRLF line endings may make the paths to be considered dirty. Adding the path to the index again will normalize the line endings in the index.

**Set to string value "crlf"**

This setting forces Git to normalize line endings for this file on checkin and convert them to CRLF when the file is checked out.

**Set to string value "lf"**

This setting forces Git to normalize line endings to LF on checkin and prevents conversion to CRLF when the file is checked out.

**Backwards compatibility with crlf attribute**

For backwards compatibility, the crlf attribute is interpreted as follows:

crlf text

-crlf -text

crlf=input eol=lf

**End-of-line conversion**

While Git normally leaves file contents alone, it can be configured to normalize line endings to LF in the repository and, optionally, to convert them to CRLF when files are checked out.

If you simply want to have CRLF line endings in your working directory regardless of the repository you are working with, you can set the config variable "core.autocrlf" without using any attributes.

[core]

autocrlf = true

This does not force normalization of text files, but does ensure that text files that you introduce to the repository have their line endings normalized to LF when they are added, and that files that are already normalized in the repository stay normalized.

If you want to ensure that text files that any contributor introduces to the repository have their line endings normalized, you can set the text attribute to "auto" for **all** files.

\* text=auto

The attributes allow a fine-grained control, how the line endings are converted. Here is an example that will make Git normalize .txt, .vcproj and .sh files, ensure that .vcproj files have CRLF and .sh files have LF in the working directory, and prevent .jpg files from being normalized regardless of their content.

\* text=auto

\*.txt text

\*.vcproj text eol=crlf

\*.sh text eol=lf

\*.jpg -text

|  |  |
| --- | --- |
| **Note** | When text=auto conversion is enabled in a cross-platform project using push and pull to a central repository the text files containing CRLFs should be normalized. |

From a clean working directory:

$ echo "\* text=auto" >.gitattributes

$ git add --renormalize .

$ git status # Show files that will be normalized

$ git commit -m "Introduce end-of-line normalization"

If any files that should not be normalized show up in **git status**, unset their text attribute before running **git add -u**.

manual.pdf -text

Conversely, text files that Git does not detect can have normalization enabled manually.

weirdchars.txt text

If core.safecrlf is set to "true" or "warn", Git verifies if the conversion is reversible for the current setting of core.autocrlf. For "true", Git rejects irreversible conversions; for "warn", Git only prints a warning but accepts an irreversible conversion. The safety triggers to prevent such a conversion done to the files in the work tree, but there are a few exceptions. Even though…​

* **git add** itself does not touch the files in the work tree, the next checkout would, so the safety triggers;
* **git apply** to update a text file with a patch does touch the files in the work tree, but the operation is about text files and CRLF conversion is about fixing the line ending inconsistencies, so the safety does not trigger;
* **git diff** itself does not touch the files in the work tree, it is often run to inspect the changes you intend to next **git add**. To catch potential problems early, safety triggers.

**working-tree-encoding**

Git recognizes files encoded in ASCII or one of its supersets (e.g. UTF-8, ISO-8859-1, …​) as text files. Files encoded in certain other encodings (e.g. UTF-16) are interpreted as binary and consequently built-in Git text processing tools (e.g. **git diff**) as well as most Git web front ends do not visualize the contents of these files by default.

In these cases you can tell Git the encoding of a file in the working directory with the working-tree-encoding attribute. If a file with this attribute is added to Git, then Git re-encodes the content from the specified encoding to UTF-8. Finally, Git stores the UTF-8 encoded content in its internal data structure (called "the index"). On checkout the content is re-encoded back to the specified encoding.

Please note that using the working-tree-encoding attribute may have a number of pitfalls:

* Alternative Git implementations (e.g. JGit or libgit2) and older Git versions (as of March 2018) do not support the working-tree-encoding attribute. If you decide to use the working-tree-encoding attribute in your repository, then it is strongly recommended to ensure that all clients working with the repository support it.

For example, Microsoft Visual Studio resources files (\*.rc) or PowerShell script files (\*.ps1) are sometimes encoded in UTF-16. If you declare \*.ps1 as files as UTF-16 and you add foo.ps1 with a working-tree-encoding enabled Git client, then foo.ps1 will be stored as UTF-8 internally. A client without working-tree-encoding support will checkout foo.ps1 as UTF-8 encoded file. This will typically cause trouble for the users of this file.

If a Git client that does not support the working-tree-encoding attribute adds a new file bar.ps1, then bar.ps1 will be stored "as-is" internally (in this example probably as UTF-16). A client with working-tree-encoding support will interpret the internal contents as UTF-8 and try to convert it to UTF-16 on checkout. That operation will fail and cause an error.

* Reencoding content to non-UTF encodings can cause errors as the conversion might not be UTF-8 round trip safe. If you suspect your encoding to not be round trip safe, then add it to core.checkRoundtripEncoding to make Git check the round trip encoding (see [git-config[1]](https://git-scm.com/docs/git-config)). SHIFT-JIS (Japanese character set) is known to have round trip issues with UTF-8 and is checked by default.
* Reencoding content requires resources that might slow down certain Git operations (e.g **git checkout** or **git add**).

Use the working-tree-encoding attribute only if you cannot store a file in UTF-8 encoding and if you want Git to be able to process the content as text.

As an example, use the following attributes if your **\*.ps1** files are UTF-16 encoded with byte order mark (BOM) and you want Git to perform automatic line ending conversion based on your platform.

\*.ps1 text working-tree-encoding=UTF-16

Use the following attributes if your **\*.ps1** files are UTF-16 little endian encoded without BOM and you want Git to use Windows line endings in the working directory (use UTF-16LE-BOM instead of UTF-16LE if you want UTF-16 little endian with BOM). Please note, it is highly recommended to explicitly define the line endings with eol if the working-tree-encoding attribute is used to avoid ambiguity.

\*.ps1 text working-tree-encoding=UTF-16LE eol=CRLF

You can get a list of all available encodings on your platform with the following command:

iconv --list

If you do not know the encoding of a file, then you can use the file command to guess the encoding:

file foo.ps1

**ident**

When the attribute ident is set for a path, Git replaces $Id$ in the blob object with $Id:, followed by the 40-character hexadecimal blob object name, followed by a dollar sign $ upon checkout. Any byte sequence that begins with $Id: and ends with $ in the worktree file is replaced with $Id$ upon check-in.

**filter**

A filter attribute can be set to a string value that names a filter driver specified in the configuration.

A filter driver consists of a clean command and a smudge command, either of which can be left unspecified. Upon checkout, when the smudge command is specified, the command is fed the blob object from its standard input, and its standard output is used to update the worktree file. Similarly, the clean command is used to convert the contents of worktree file upon checkin. By default these commands process only a single blob and terminate. If a long running process filter is used in place of clean and/or smudge filters, then Git can process all blobs with a single filter command invocation for the entire life of a single Git command, for example git add --all. If a long running process filter is configured then it always takes precedence over a configured single blob filter. See section below for the description of the protocol used to communicate with a process filter.

One use of the content filtering is to massage the content into a shape that is more convenient for the platform, filesystem, and the user to use. For this mode of operation, the key phrase here is "more convenient" and not "turning something unusable into usable". In other words, the intent is that if someone unsets the filter driver definition, or does not have the appropriate filter program, the project should still be usable.

Another use of the content filtering is to store the content that cannot be directly used in the repository (e.g. a UUID that refers to the true content stored outside Git, or an encrypted content) and turn it into a usable form upon checkout (e.g. download the external content, or decrypt the encrypted content).

These two filters behave differently, and by default, a filter is taken as the former, massaging the contents into more convenient shape. A missing filter driver definition in the config, or a filter driver that exits with a non-zero status, is not an error but makes the filter a no-op passthru.

You can declare that a filter turns a content that by itself is unusable into a usable content by setting the filter.<driver>.required configuration variable to true.

Note: Whenever the clean filter is changed, the repo should be renormalized: $ git add --renormalize .

For example, in .gitattributes, you would assign the filter attribute for paths.

\*.c filter=indent

Then you would define a "filter.indent.clean" and "filter.indent.smudge" configuration in your .git/config to specify a pair of commands to modify the contents of C programs when the source files are checked in ("clean" is run) and checked out (no change is made because the command is "cat").

[filter "indent"]

clean = indent

smudge = cat

For best results, clean should not alter its output further if it is run twice ("clean→clean" should be equivalent to "clean"), and multiple smudge commands should not alter clean's output ("smudge→smudge→clean" should be equivalent to "clean"). See the section on merging below.

The "indent" filter is well-behaved in this regard: it will not modify input that is already correctly indented. In this case, the lack of a smudge filter means that the clean filter **must** accept its own output without modifying it.

If a filter **must** succeed in order to make the stored contents usable, you can declare that the filter is required, in the configuration:

[filter "crypt"]

clean = openssl enc ...

smudge = openssl enc -d ...

required

Sequence "%f" on the filter command line is replaced with the name of the file the filter is working on. A filter might use this in keyword substitution. For example:

[filter "p4"]

clean = git-p4-filter --clean %f

smudge = git-p4-filter --smudge %f

Note that "%f" is the name of the path that is being worked on. Depending on the version that is being filtered, the corresponding file on disk may not exist, or may have different contents. So, smudge and clean commands should not try to access the file on disk, but only act as filters on the content provided to them on standard input.

**Long Running Filter Process**

If the filter command (a string value) is defined via filter.<driver>.process then Git can process all blobs with a single filter invocation for the entire life of a single Git command. This is achieved by using the long-running process protocol (described in technical/long-running-process-protocol.txt).

When Git encounters the first file that needs to be cleaned or smudged, it starts the filter and performs the handshake. In the handshake, the welcome message sent by Git is "git-filter-client", only version 2 is supported, and the supported capabilities are "clean", "smudge", and "delay".

Afterwards Git sends a list of "key=value" pairs terminated with a flush packet. The list will contain at least the filter command (based on the supported capabilities) and the pathname of the file to filter relative to the repository root. Right after the flush packet Git sends the content split in zero or more pkt-line packets and a flush packet to terminate content. Please note, that the filter must not send any response before it received the content and the final flush packet. Also note that the "value" of a "key=value" pair can contain the "=" character whereas the key would never contain that character.

packet: git> command=smudge

packet: git> pathname=path/testfile.dat

packet: git> 0000

packet: git> CONTENT

packet: git> 0000

The filter is expected to respond with a list of "key=value" pairs terminated with a flush packet. If the filter does not experience problems then the list must contain a "success" status. Right after these packets the filter is expected to send the content in zero or more pkt-line packets and a flush packet at the end. Finally, a second list of "key=value" pairs terminated with a flush packet is expected. The filter can change the status in the second list or keep the status as is with an empty list. Please note that the empty list must be terminated with a flush packet regardless.

packet: git< status=success

packet: git< 0000

packet: git< SMUDGED\_CONTENT

packet: git< 0000

packet: git< 0000 # empty list, keep "status=success" unchanged!

If the result content is empty then the filter is expected to respond with a "success" status and a flush packet to signal the empty content.

packet: git< status=success

packet: git< 0000

packet: git< 0000 # empty content!

packet: git< 0000 # empty list, keep "status=success" unchanged!

In case the filter cannot or does not want to process the content, it is expected to respond with an "error" status.

packet: git< status=error

packet: git< 0000

If the filter experiences an error during processing, then it can send the status "error" after the content was (partially or completely) sent.

packet: git< status=success

packet: git< 0000

packet: git< HALF\_WRITTEN\_ERRONEOUS\_CONTENT

packet: git< 0000

packet: git< status=error

packet: git< 0000

In case the filter cannot or does not want to process the content as well as any future content for the lifetime of the Git process, then it is expected to respond with an "abort" status at any point in the protocol.

packet: git< status=abort

packet: git< 0000

Git neither stops nor restarts the filter process in case the "error"/"abort" status is set. However, Git sets its exit code according to the filter.<driver>.required flag, mimicking the behavior of the filter.<driver>.clean / filter.<driver>.smudge mechanism.

If the filter dies during the communication or does not adhere to the protocol then Git will stop the filter process and restart it with the next file that needs to be processed. Depending on the filter.<driver>.required flag Git will interpret that as error.

**Delay**

If the filter supports the "delay" capability, then Git can send the flag "can-delay" after the filter command and pathname. This flag denotes that the filter can delay filtering the current blob (e.g. to compensate network latencies) by responding with no content but with the status "delayed" and a flush packet.

packet: git> command=smudge

packet: git> pathname=path/testfile.dat

packet: git> can-delay=1

packet: git> 0000

packet: git> CONTENT

packet: git> 0000

packet: git< status=delayed

packet: git< 0000

If the filter supports the "delay" capability then it must support the "list\_available\_blobs" command. If Git sends this command, then the filter is expected to return a list of pathnames representing blobs that have been delayed earlier and are now available. The list must be terminated with a flush packet followed by a "success" status that is also terminated with a flush packet. If no blobs for the delayed paths are available, yet, then the filter is expected to block the response until at least one blob becomes available. The filter can tell Git that it has no more delayed blobs by sending an empty list. As soon as the filter responds with an empty list, Git stops asking. All blobs that Git has not received at this point are considered missing and will result in an error.

packet: git> command=list\_available\_blobs

packet: git> 0000

packet: git< pathname=path/testfile.dat

packet: git< pathname=path/otherfile.dat

packet: git< 0000

packet: git< status=success

packet: git< 0000

After Git received the pathnames, it will request the corresponding blobs again. These requests contain a pathname and an empty content section. The filter is expected to respond with the smudged content in the usual way as explained above.

packet: git> command=smudge

packet: git> pathname=path/testfile.dat

packet: git> 0000

packet: git> 0000 # empty content!

packet: git< status=success

packet: git< 0000

packet: git< SMUDGED\_CONTENT

packet: git< 0000

packet: git< 0000 # empty list, keep "status=success" unchanged!

**Example**

A long running filter demo implementation can be found in contrib/long-running-filter/example.pl located in the Git core repository. If you develop your own long running filter process then the GIT\_TRACE\_PACKET environment variables can be very helpful for debugging (see [git[1]](https://git-scm.com/docs/git)).

Please note that you cannot use an existing filter.<driver>.clean or filter.<driver>.smudge command with filter.<driver>.process because the former two use a different inter process communication protocol than the latter one.

**Interaction between checkin/checkout attributes**

In the check-in codepath, the worktree file is first converted with filter driver (if specified and corresponding driver defined), then the result is processed with ident (if specified), and then finally with text (again, if specified and applicable).

In the check-out codepath, the blob content is first converted with text, and then ident and fed to filter.

**Merging branches with differing checkin/checkout attributes**

If you have added attributes to a file that cause the canonical repository format for that file to change, such as adding a clean/smudge filter or text/eol/ident attributes, merging anything where the attribute is not in place would normally cause merge conflicts.

To prevent these unnecessary merge conflicts, Git can be told to run a virtual check-out and check-in of all three stages of a file when resolving a three-way merge by setting the merge.renormalize configuration variable. This prevents changes caused by check-in conversion from causing spurious merge conflicts when a converted file is merged with an unconverted file.

As long as a "smudge→clean" results in the same output as a "clean" even on files that are already smudged, this strategy will automatically resolve all filter-related conflicts. Filters that do not act in this way may cause additional merge conflicts that must be resolved manually.

**Generating diff text**

**diff**

The attribute diff affects how Git generates diffs for particular files. It can tell Git whether to generate a textual patch for the path or to treat the path as a binary file. It can also affect what line is shown on the hunk header @@ -k,l +n,m @@ line, tell Git to use an external command to generate the diff, or ask Git to convert binary files to a text format before generating the diff.

**Set**

A path to which the diff attribute is set is treated as text, even when they contain byte values that normally never appear in text files, such as NUL.

**Unset**

A path to which the diff attribute is unset will generate Binary files differ (or a binary patch, if binary patches are enabled).

**Unspecified**

A path to which the diff attribute is unspecified first gets its contents inspected, and if it looks like text and is smaller than core.bigFileThreshold, it is treated as text. Otherwise it would generate Binary files differ.

**String**

Diff is shown using the specified diff driver. Each driver may specify one or more options, as described in the following section. The options for the diff driver "foo" are defined by the configuration variables in the "diff.foo" section of the Git config file.

**Defining an external diff driver**

The definition of a diff driver is done in gitconfig, not gitattributes file, so strictly speaking this manual page is a wrong place to talk about it. However…​

To define an external diff driver jcdiff, add a section to your $GIT\_DIR/config file (or $HOME/.gitconfig file) like this:

[diff "jcdiff"]

command = j-c-diff

When Git needs to show you a diff for the path with diff attribute set to jcdiff, it calls the command you specified with the above configuration, i.e. j-c-diff, with 7 parameters, just like GIT\_EXTERNAL\_DIFF program is called. See [git[1]](https://git-scm.com/docs/git) for details.

**Defining a custom hunk-header**

Each group of changes (called a "hunk") in the textual diff output is prefixed with a line of the form:

@@ -k,l +n,m @@ TEXT

This is called a **hunk header**. The "TEXT" portion is by default a line that begins with an alphabet, an underscore or a dollar sign; this matches what GNU **diff -p** output uses. This default selection however is not suited for some contents, and you can use a customized pattern to make a selection.

First, in .gitattributes, you would assign the diff attribute for paths.

\*.tex diff=tex

Then, you would define a "diff.tex.xfuncname" configuration to specify a regular expression that matches a line that you would want to appear as the hunk header "TEXT". Add a section to your $GIT\_DIR/config file (or $HOME/.gitconfig file) like this:

[diff "tex"]

xfuncname = "^(\\\\(sub)\*section\\{.\*)$"

Note. A single level of backslashes are eaten by the configuration file parser, so you would need to double the backslashes; the pattern above picks a line that begins with a backslash, and zero or more occurrences of sub followed by section followed by open brace, to the end of line.

There are a few built-in patterns to make this easier, and tex is one of them, so you do not have to write the above in your configuration file (you still need to enable this with the attribute mechanism, via .gitattributes). The following built in patterns are available:

* ada suitable for source code in the Ada language.
* bibtex suitable for files with BibTeX coded references.
* cpp suitable for source code in the C and C++ languages.
* csharp suitable for source code in the C# language.
* css suitable for cascading style sheets.
* dts suitable for devicetree (DTS) files.
* elixir suitable for source code in the Elixir language.
* fortran suitable for source code in the Fortran language.
* fountain suitable for Fountain documents.
* golang suitable for source code in the Go language.
* html suitable for HTML/XHTML documents.
* java suitable for source code in the Java language.
* markdown suitable for Markdown documents.
* matlab suitable for source code in the MATLAB and Octave languages.
* objc suitable for source code in the Objective-C language.
* pascal suitable for source code in the Pascal/Delphi language.
* perl suitable for source code in the Perl language.
* php suitable for source code in the PHP language.
* python suitable for source code in the Python language.
* ruby suitable for source code in the Ruby language.
* rust suitable for source code in the Rust language.
* tex suitable for source code for LaTeX documents.

**Customizing word diff**

You can customize the rules that git diff --word-diff uses to split words in a line, by specifying an appropriate regular expression in the "diff.\*.wordRegex" configuration variable. For example, in TeX a backslash followed by a sequence of letters forms a command, but several such commands can be run together without intervening whitespace. To separate them, use a regular expression in your $GIT\_DIR/config file (or $HOME/.gitconfig file) like this:

[diff "tex"]

wordRegex = "\\\\[a-zA-Z]+|[{}]|\\\\.|[^\\{}[:space:]]+"

A built-in pattern is provided for all languages listed in the previous section.

**Performing text diffs of binary files**

Sometimes it is desirable to see the diff of a text-converted version of some binary files. For example, a word processor document can be converted to an ASCII text representation, and the diff of the text shown. Even though this conversion loses some information, the resulting diff is useful for human viewing (but cannot be applied directly).

The textconv config option is used to define a program for performing such a conversion. The program should take a single argument, the name of a file to convert, and produce the resulting text on stdout.

For example, to show the diff of the exif information of a file instead of the binary information (assuming you have the exif tool installed), add the following section to your $GIT\_DIR/config file (or $HOME/.gitconfig file):

[diff "jpg"]

textconv = exif

|  |  |
| --- | --- |
| **Note** | The text conversion is generally a one-way conversion; in this example, we lose the actual image contents and focus just on the text data. This means that diffs generated by textconv are *not* suitable for applying. For this reason, only git diff and the git log family of commands (i.e., log, whatchanged, show) will perform text conversion. git format-patch will never generate this output. If you want to send somebody a text-converted diff of a binary file (e.g., because it quickly conveys the changes you have made), you should generate it separately and send it as a comment *in addition to* the usual binary diff that you might send. |

Because text conversion can be slow, especially when doing a large number of them with git log -p, Git provides a mechanism to cache the output and use it in future diffs. To enable caching, set the "cachetextconv" variable in your diff driver’s config. For example:

[diff "jpg"]

textconv = exif

cachetextconv = true

This will cache the result of running "exif" on each blob indefinitely. If you change the textconv config variable for a diff driver, Git will automatically invalidate the cache entries and re-run the textconv filter. If you want to invalidate the cache manually (e.g., because your version of "exif" was updated and now produces better output), you can remove the cache manually with git update-ref -d refs/notes/textconv/jpg (where "jpg" is the name of the diff driver, as in the example above).

**Choosing textconv versus external diff**

If you want to show differences between binary or specially-formatted blobs in your repository, you can choose to use either an external diff command, or to use textconv to convert them to a diff-able text format. Which method you choose depends on your exact situation.

The advantage of using an external diff command is flexibility. You are not bound to find line-oriented changes, nor is it necessary for the output to resemble unified diff. You are free to locate and report changes in the most appropriate way for your data format.

A textconv, by comparison, is much more limiting. You provide a transformation of the data into a line-oriented text format, and Git uses its regular diff tools to generate the output. There are several advantages to choosing this method:

1. Ease of use. It is often much simpler to write a binary to text transformation than it is to perform your own diff. In many cases, existing programs can be used as textconv filters (e.g., exif, odt2txt).
2. Git diff features. By performing only the transformation step yourself, you can still utilize many of Git’s diff features, including colorization, word-diff, and combined diffs for merges.
3. Caching. Textconv caching can speed up repeated diffs, such as those you might trigger by running git log -p.

**Marking files as binary**

Git usually guesses correctly whether a blob contains text or binary data by examining the beginning of the contents. However, sometimes you may want to override its decision, either because a blob contains binary data later in the file, or because the content, while technically composed of text characters, is opaque to a human reader. For example, many postscript files contain only ASCII characters, but produce noisy and meaningless diffs.

The simplest way to mark a file as binary is to unset the diff attribute in the .gitattributes file:

\*.ps -diff

This will cause Git to generate Binary files differ (or a binary patch, if binary patches are enabled) instead of a regular diff.

However, one may also want to specify other diff driver attributes. For example, you might want to use textconv to convert postscript files to an ASCII representation for human viewing, but otherwise treat them as binary files. You cannot specify both -diff and diff=ps attributes. The solution is to use the diff.\*.binary config option:

[diff "ps"]

textconv = ps2ascii

binary = true

**Performing a three-way merge**

**merge**

The attribute merge affects how three versions of a file are merged when a file-level merge is necessary during git merge, and other commands such as git revert and git cherry-pick.

**Set**

Built-in 3-way merge driver is used to merge the contents in a way similar to **merge** command of RCS suite. This is suitable for ordinary text files.

**Unset**

Take the version from the current branch as the tentative merge result, and declare that the merge has conflicts. This is suitable for binary files that do not have a well-defined merge semantics.

**Unspecified**

By default, this uses the same built-in 3-way merge driver as is the case when the merge attribute is set. However, the merge.default configuration variable can name different merge driver to be used with paths for which the merge attribute is unspecified.

**String**

3-way merge is performed using the specified custom merge driver. The built-in 3-way merge driver can be explicitly specified by asking for "text" driver; the built-in "take the current branch" driver can be requested with "binary".

**Built-in merge drivers**

There are a few built-in low-level merge drivers defined that can be asked for via the merge attribute.

**text**

Usual 3-way file level merge for text files. Conflicted regions are marked with conflict markers <<<<<<<, ======= and >>>>>>>. The version from your branch appears before the ======= marker, and the version from the merged branch appears after the ======= marker.

**binary**

Keep the version from your branch in the work tree, but leave the path in the conflicted state for the user to sort out.

**union**

Run 3-way file level merge for text files, but take lines from both versions, instead of leaving conflict markers. This tends to leave the added lines in the resulting file in random order and the user should verify the result. Do not use this if you do not understand the implications.

**Defining a custom merge driver**

The definition of a merge driver is done in the .git/config file, not in the gitattributes file, so strictly speaking this manual page is a wrong place to talk about it. However…​

To define a custom merge driver filfre, add a section to your $GIT\_DIR/config file (or $HOME/.gitconfig file) like this:

[merge "filfre"]

name = feel-free merge driver

driver = filfre %O %A %B %L %P

recursive = binary

The merge.\*.name variable gives the driver a human-readable name.

The merge.\*.driver variable’s value is used to construct a command to run to merge ancestor’s version (%O), current version (%A) and the other branches' version (%B). These three tokens are replaced with the names of temporary files that hold the contents of these versions when the command line is built. Additionally, %L will be replaced with the conflict marker size (see below).

The merge driver is expected to leave the result of the merge in the file named with %A by overwriting it, and exit with zero status if it managed to merge them cleanly, or non-zero if there were conflicts.

The merge.\*.recursive variable specifies what other merge driver to use when the merge driver is called for an internal merge between common ancestors, when there are more than one. When left unspecified, the driver itself is used for both internal merge and the final merge.

The merge driver can learn the pathname in which the merged result will be stored via placeholder %P.

**conflict-marker-size**

This attribute controls the length of conflict markers left in the work tree file during a conflicted merge. Only setting to the value to a positive integer has any meaningful effect.

For example, this line in .gitattributes can be used to tell the merge machinery to leave much longer (instead of the usual 7-character-long) conflict markers when merging the file Documentation/git-merge.txt results in a conflict.

Documentation/git-merge.txt conflict-marker-size=32

**Checking whitespace errors**

**whitespace**

The core.whitespace configuration variable allows you to define what **diff** and **apply** should consider whitespace errors for all paths in the project (See [git-config[1]](https://git-scm.com/docs/git-config)). This attribute gives you finer control per path.

**Set**

Notice all types of potential whitespace errors known to Git. The tab width is taken from the value of the core.whitespace configuration variable.

**Unset**

Do not notice anything as error.

**Unspecified**

Use the value of the core.whitespace configuration variable to decide what to notice as error.

**String**

Specify a comma separate list of common whitespace problems to notice in the same format as the core.whitespace configuration variable.

**Creating an archive**

**export-ignore**

Files and directories with the attribute export-ignore won’t be added to archive files.

**export-subst**

If the attribute export-subst is set for a file then Git will expand several placeholders when adding this file to an archive. The expansion depends on the availability of a commit ID, i.e., if [git-archive[1]](https://git-scm.com/docs/git-archive) has been given a tree instead of a commit or a tag then no replacement will be done. The placeholders are the same as those for the option --pretty=format: of [git-log[1]](https://git-scm.com/docs/git-log), except that they need to be wrapped like this: $Format:PLACEHOLDERS$ in the file. E.g. the string $Format:%H$ will be replaced by the commit hash.

**Packing objects**

**delta**

Delta compression will not be attempted for blobs for paths with the attribute delta set to false.

**Viewing files in GUI tools**

**encoding**

The value of this attribute specifies the character encoding that should be used by GUI tools (e.g. [gitk[1]](https://git-scm.com/docs/gitk) and [git-gui[1]](https://git-scm.com/docs/git-gui)) to display the contents of the relevant file. Note that due to performance considerations [gitk[1]](https://git-scm.com/docs/gitk) does not use this attribute unless you manually enable per-file encodings in its options.

If this attribute is not set or has an invalid value, the value of the gui.encoding configuration variable is used instead (See [git-config[1]](https://git-scm.com/docs/git-config)).

**USING MACRO ATTRIBUTES**

You do not want any end-of-line conversions applied to, nor textual diffs produced for, any binary file you track. You would need to specify e.g.

\*.jpg -text -diff

but that may become cumbersome, when you have many attributes. Using macro attributes, you can define an attribute that, when set, also sets or unsets a number of other attributes at the same time. The system knows a built-in macro attribute, binary:

\*.jpg binary

Setting the "binary" attribute also unsets the "text" and "diff" attributes as above. Note that macro attributes can only be "Set", though setting one might have the effect of setting or unsetting other attributes or even returning other attributes to the "Unspecified" state.

**DEFINING MACRO ATTRIBUTES**

Custom macro attributes can be defined only in top-level gitattributes files ($GIT\_DIR/info/attributes, the .gitattributes file at the top level of the working tree, or the global or system-wide gitattributes files), not in .gitattributes files in working tree subdirectories. The built-in macro attribute "binary" is equivalent to:

[attr]binary -diff -merge -text

**EXAMPLES**

If you have these three gitattributes file:

(in $GIT\_DIR/info/attributes)

a\* foo !bar -baz

(in .gitattributes)

abc foo bar baz

(in t/.gitattributes)

ab\* merge=filfre

abc -foo -bar

\*.c frotz

the attributes given to path t/abc are computed as follows:

1. By examining t/.gitattributes (which is in the same directory as the path in question), Git finds that the first line matches. merge attribute is set. It also finds that the second line matches, and attributes foo and bar are unset.
2. Then it examines .gitattributes (which is in the parent directory), and finds that the first line matches, but t/.gitattributes file already decided how merge, foo and bar attributes should be given to this path, so it leaves foo and bar unset. Attribute baz is set.
3. Finally it examines $GIT\_DIR/info/attributes. This file is used to override the in-tree settings. The first line is a match, and foo is set, bar is reverted to unspecified state, and baz is unset.

As the result, the attributes assignment to t/abc becomes:

foo set to true

bar unspecified

baz set to false

merge set to string value "filfre"

frotz unspecified

**SEE ALSO**

[git-check-attr[1]](https://git-scm.com/docs/git-check-attr).

**GIT**

Part of the [git[1]](https://git-scm.com/docs/git) suite

[About this site](https://git-scm.com/site)  
Patches, suggestions, and comments are welcome.

Git is a member of [Software Freedom Conservancy](https://git-scm.com/sfc)