

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. 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 how the pattern matches paths are the same as in .gitignore files; see [gitignore\(5\)](#). Unlike .gitignore, negative patterns are forbidden.

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\(1\)](#)). 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 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.

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 normalization. If Git decides that the content is text, its line endings are normalized to LF on checkin.

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 normalization without any content checks, effectively setting the text attribute.

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.

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.

```
*.txt text
*.vcproj eol=crlf
```

```
*.sh eol=lf
*.jpg -text
```

Other source code management systems normalize all text files in their repositories, and there are two ways to enable similar automatic normalization in Git.

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 changing any attributes.

```
[core]
autocrlf = true
```

This does not force normalization of all 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 interoperate with a source code management system that enforces end-of-line normalization, or you simply want all text files in your repository to be normalized, you should instead set the text attribute to `auto` for *all* files.

```
* text=auto
```

This ensures that all files that Git considers to be text will have normalized (LF) line endings in the repository. The `core.eol` configuration variable controls which line endings Git will use for normalized files in your working directory; the default is to use the native line ending for your platform, or CRLF if `core.autocrlf` is set.

Note

When `text=auto` normalization is enabled in an existing repository, any text files containing CRLFs should be normalized. If they are not they will be normalized the next time someone tries to change them, causing unfortunate misattribution. From a clean working directory:

```
$ echo * text=auto >>.gitattributes
$ rm .git/index # Remove the index to force Git to
$ git reset # re-scan the working directory
$ git status # Show files that will be normalized
$ git add -u
$ git add .gitattributes
$ 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.

indent

When the attribute `indent` 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.

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`.

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
```

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, 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\(1\)](#) 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.
- fortran suitable for source code in the Fortran language.
- html suitable for HTML/XHTML documents.
- java suitable for source code in the Java language.
- matlab suitable for source code in the MATLAB language.
- 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.
- 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
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.

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\(1\)](#)). 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\(1\)](#) 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\(1\)](#), 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\(1\)](#) and [git-gui\(1\)](#)) to display the contents of the relevant file. Note that due to performance considerations [gitk\(1\)](#) 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\(1\)](#)).

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
```

EXAMPLE

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\(1\)](#).

GIT

Part of the [git\(1\)](#) suite