

**NAME**

git-log – Show commit logs

**SYNOPSIS**

```
git log [<options>] [<since>..

```

**DESCRIPTION**

Shows the commit logs.

The command takes options applicable to the *git rev-list* command to control what is shown and how, and options applicable to the *git diff*-\* commands to control how the changes each commit introduces are shown.

**OPTIONS**

--<n>

Limits the number of commits to show. Note that this is a commit limiting option, see below.

<since>..

Show only commits between the named two commits. When either <since> or <until> is omitted, it defaults to HEAD, i.e. the tip of the current branch. For a more complete list of ways to spell <since> and <until>, see [gitrevisions\(7\)](#).

--follow

Continue listing the history of a file beyond renames (works only for a single file).

--no-decorate, --decorate[=short|full|no]

Print out the ref names of any commits that are shown. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. The default option is *short*.

--source

Print out the ref name given on the command line by which each commit was reached.

--full-diff

Without this flag, "git log -p <path>..." shows commits that touch the specified paths, and diffs about the same specified paths. With this, the full diff is shown for commits that touch the specified paths; this means that "<path>..." limits only commits, and doesn't limit diff for those commits.

Note that this affects all diff-based output types, e.g. those produced by --stat etc.

--log-size

Before the log message print out its size in bytes. Intended mainly for porcelain tools consumption. If git is unable to produce a valid value size is set to zero. Note that only message is considered, if also a diff is shown its size is not included.

[--] <path>...

Show only commits that are enough to explain how the files that match the specified paths came to be. See "History Simplification" below for details and other simplification modes.

To prevent confusion with options and branch names, paths may need to be prefixed with "-- " to separate them from options or renames.

**Commit Limiting**

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied. Note that they are applied before commit ordering and formatting options, such as --reverse.

-n *number*, --max-count=<number>

Limit the number of commits to output.

- skip=<number>  
Skip *number* commits before starting to show the commit output.
- since=<date>, --after=<date>  
Show commits more recent than a specific date.
- until=<date>, --before=<date>  
Show commits older than a specific date.
- author=<pattern>, --committer=<pattern>  
Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression).
- grep=<pattern>  
Limit the commits output to ones with log message that matches the specified pattern (regular expression).
- all-match  
Limit the commits output to ones that match all given --grep, --author and --committer instead of ones that match at least one.
- i, --regexp-ignore-case  
Match the regexp limiting patterns without regard to letters case.
- E, --extended-regexp  
Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.
- F, --fixed-strings  
Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).
- remove-empty  
Stop when a given path disappears from the tree.
- merges  
Print only merge commits. This is exactly the same as --min-parents=2.
- no-merges  
Do not print commits with more than one parent. This is exactly the same as --max-parents=1.
- min-parents=<number>, --max-parents=<number>, --no-min-parents,  
--no-max-parents  
Show only commits which have at least (or at most) that many commits. In particular, --max-parents=1 is the same as --no-merges, --min-parents=2 is the same as --merges. --max-parents=0 gives all root commits and --min-parents=3 all octopus merges.  
  
--no-min-parents and --no-max-parents reset these limits (to no limit) again. Equivalent forms are --min-parents=0 (any commit has 0 or more parents) and --max-parents=-1 (negative numbers denote no upper limit).
- first-parent  
Follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.
- not  
Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next --not.

- `--all`  
Pretend as if all the refs in refs/ are listed on the command line as `<commit>`.
- `--branches[=<pattern>]`  
Pretend as if all the refs in refs/heads are listed on the command line as `<commit>`. If `<pattern>` is given, limit branches to ones matching given shell glob. If pattern lacks `?`, `,` or `[`, `/` at the end is implied.
- `--tags[=<pattern>]`  
Pretend as if all the refs in refs/tags are listed on the command line as `<commit>`. If `<pattern>` is given, limit tags to ones matching given shell glob. If pattern lacks `?`, `,` or `[`, `/` at the end is implied.
- `--remotes[=<pattern>]`  
Pretend as if all the refs in refs/remotes are listed on the command line as `<commit>`. If `<pattern>` is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks `?`, `,` or `[`, `/` at the end is implied.
- `--glob=<glob-pattern>`  
Pretend as if all the refs matching shell glob `<glob-pattern>` are listed on the command line as `<commit>`. Leading `refs/`, is automatically prepended if missing. If pattern lacks `?`, `,` or `[`, `/` at the end is implied.
- `--ignore-missing`  
Upon seeing an invalid object name in the input, pretend as if the bad input was not given.
- `--bisect`  
Pretend as if the bad bisection ref refs/bisect/bad was listed and as if it was followed by `--not` and the good bisection refs refs/bisect/good-\* on the command line.
- `--stdin`  
In addition to the `<commit>` listed on the command line, read them from the standard input. If a `--` separator is seen, stop reading commits and start reading paths to limit the result.
- `--cherry-mark`  
Like `--cherry-pick` (see below) but mark equivalent commits with `=` rather than omitting them, and inequivalent ones with `+`.
- `--cherry-pick`  
Omit any commit that introduces the same change as another commit on the "other side" when the set of commits are limited with symmetric difference.

For example, if you have two branches, A and B, a usual way to list all commits on only one side of them is with `--left-right` (see the example below in the description of the `--left-right` option). It however shows the commits that were cherry-picked from the other branch (for example, "3rd on b" may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

- `--left-only`, `--right-only`  
List only commits on the respective side of a symmetric range, i.e. only those which would be marked `<` resp. `>` by `--left-right`.

For example, `--cherry-pick --right-only A...B` omits those commits from B which are in A or are patch-equivalent to a commit in A. In other words, this lists the `+` commits from `git cherry A B`. More precisely, `--cherry-pick --right-only --no-merges` gives the exact list.

- `--cherry`  
A synonym for `--right-only --cherry-mark --no-merges`; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with `git log --cherry upstream...mybranch`, similar to `git cherry upstream mybranch`.

`-g, --walk-reflogs`  
 Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, `^commit`, `commit1..commit2`, nor `commit1...commit2` notations cannot be used).

With `--pretty` format other than oneline (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, `commit@{Nth}` notation is used in the output. When the starting commit is specified as `commit@{now}`, output also uses `commit@{timestamp}` notation instead. Under `--pretty=oneline`, the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse`. See also **git-reflog(1)**.

`--merge`  
 After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

`--boundary`  
 Output uninteresting commits at the boundary, which are usually not shown.

### History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular `<path>`. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

`<paths>`  
 Commits modifying the given `<paths>` are selected.

`--simplify-by-decoration`  
 Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode  
 Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

`--full-history`  
 Same as the default mode, but does not prune some history.

`--dense`  
 Only the selected commits are shown, plus some to have a meaningful history.

`--sparse`  
 All commits in the simplified history are shown.

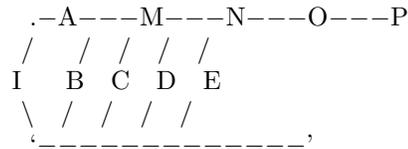
`--simplify-merges`  
 Additional option to `--full-history` to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

`--ancestry-path`  
 When given a range of commits to display (e.g. `commit1..commit2` or `commit2 ^commit1`), only display commits that exist directly on the ancestry chain between the `commit1` and `commit2`, i.e. commits that are both descendants of `commit1`, and ancestors of `commit2`.

A more detailed explanation follows.

Suppose you specified `foo` as the `<paths>`. We shall call commits that modify `foo` `!TREESAME`, and the rest `TREESAME`. (In a diff filtered for `foo`, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file `foo` in this commit graph:



The horizontal line of history `A---P` is taken to be the first parent of each merge. The commits are:

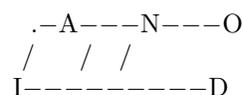
- I is the initial commit, in which `foo` exists with contents "asdf", and a file `quux` exists with contents "quux". Initial commits are compared to an empty tree, so I is `!TREESAME`.
- In A, `foo` contains just "foo".
- B contains the same change as A. Its merge M is trivial and hence `TREESAME` to all parents.
- C does not change `foo`, but its merge N changes it to "foobar", so it is not `TREESAME` to any parent.
- D sets `foo` to "baz". Its merge O combines the strings from N and D to "foobarbaz"; i.e., it is not `TREESAME` to any parent.
- E changes `quux` to "xyzy", and its merge P combines the strings to "quux xyzy". Despite appearing interesting, P is `TREESAME` to all parents.

`rev-list` walks backwards through history, including or excluding commits based on whether `--full-history` and/or parent rewriting (via `--parents` or `--children`) are used. The following settings are available.

#### Default mode

Commits are included if they are not `TREESAME` to any parent (though this can be changed, see `--sparse` below). If the commit was a merge, and it was `TREESAME` to one parent, follow only that parent. (Even if there are several `TREESAME` parents, follow only one of them.) Otherwise, follow all parents.

This results in:



Note how the rule to only follow the TREESAME parent, if one is available, removed B from consideration entirely. C was considered via N, but is TREESAME. Root commits are compared to an empty tree, so I is !TREESAME.

Parent/child relations are only visible with `--parents`, but that does not affect the commits selected in default mode, so we have shown the parent lines.

`--full-history` without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I A B N D O
```

P and M were excluded because they are TREESAME to a parent. E, C and B were all walked, but only B was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

`--full-history` with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see `--sparse` below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```

.-A---M---N---O---P
 /   /   /   /   /
I    B   /   D   /
 \   /   /   /   /
  '-----'

```

Compare to `--full-history` without rewriting above. Note that E was pruned away because it is TREESAME, but the parent list of P was rewritten to contain E's parent I. The same happened for C and N. Note also that P was included despite being TREESAME.

In addition to the above settings, you can change whether TREESAME affects inclusion:

`--dense`

Commits that are walked are included if they are not TREESAME to any parent.

`--sparse`

All commits that are walked are included.

Note that without `--full-history`, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

`--simplify-merges`

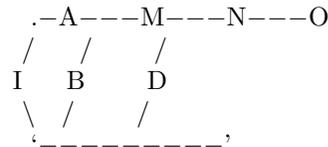
First, build a history graph in the same way that `--full-history` with parent rewriting does (see above).

Then simplify each commit 'C' to its replacement C' in the final history according to the following rules:

- Set 'C' to C.
- Replace each parent 'P' of C' with its simplification 'P'. In the process, drop parents that are ancestors of other parents, and remove duplicates.

- If after this parent rewriting, 'C' is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to `--full-history` with parent rewriting. The example turns into:



Note the major differences in N and P over `--full-history`:

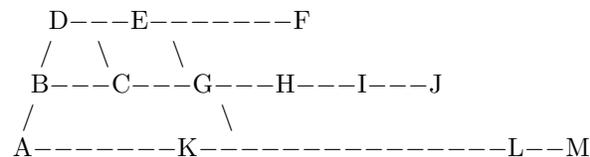
- N's parent list had I removed, because it is an ancestor of the other parent M. Still, N remained because it is !TREESAME.
- P's parent list similarly had I removed. P was then removed completely, because it had one parent and is TREESAME.

Finally, there is a fifth simplification mode available:

`--ancestry-path`

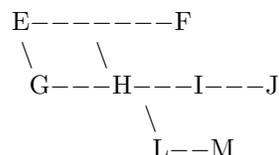
Limit the displayed commits to those directly on the ancestry chain between the "from" and "to" commits in the given commit range. I.e. only display commits that are ancestor of the "to" commit, and descendants of the "from" commit.

As an example use case, consider the following commit history:



A regular `D..M` computes the set of commits that are ancestors of M, but excludes the ones that are ancestors of D. This is useful to see what happened to the history leading to M since D, in the sense that "what does M have that did not exist in D". The result in this example would be all the commits, except A and B (and D itself, of course).

When we want to find out what commits in M are contaminated with the bug introduced by D and need fixing, however, we might want to view only the subset of `D..M` that are actually descendants of D, i.e. excluding C and K. This is exactly what the `--ancestry-path` option does. Applied to the `D..M` range, it results in:



The *--simplify-by-decoration* option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

### Commit Ordering

By default, the commits are shown in reverse chronological order.

*--topo-order*

This option makes them appear in topological order (i.e. descendant commits are shown before their parents).

*--date-order*

This option is similar to *--topo-order* in the sense that no parent comes before all of its children, but otherwise things are still ordered in the commit timestamp order.

*--reverse*

Output the commits in reverse order. Cannot be combined with *--walk-reflogs*.

### Object Traversal

These options are mostly targeted for packing of git repositories.

*--objects*

Print the object IDs of any object referenced by the listed commits. *--objects foo ^bar* thus means "send me all object IDs which I need to download if I have the commit object *bar*, but not *foo*".

*--objects-edge*

Similar to *--objects*, but also print the IDs of excluded commits prefixed with a "-" character. This is used by **git-pack-objects**(1) to build "thin" pack, which records objects in deltified form based on objects contained in these excluded commits to reduce network traffic.

*--unpacked*

Only useful with *--objects*; print the object IDs that are not in packs.

*--no-walk*

Only show the given revs, but do not traverse their ancestors.

*--do-walk*

Overrides a previous *--no-walk*.

### Commit Formatting

*--pretty[=<format>]*, *--format=<format>*

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *email*, *raw* and *format:<string>*. See the "PRETTY FORMATS" section for some additional details for each format. When omitted, the format defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see **git-config**(1)).

*--abbrev-commit*

Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with *--abbrev=<n>* (which also modifies diff output, if it is displayed).

This should make *--pretty=oneline* a whole lot more readable for people using 80-column terminals.

*--no-abbrev-commit*

Show the full 40-byte hexadecimal commit object name. This negates *--abbrev-commit*

and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding[=<encoding>]

The commit objects record the encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8.

--notes[=<ref>]

Show the notes (see **git-notes**(1)) that annotate the commit, when showing the commit log message. This is the default for git log, git show and git whatchanged commands when there is no --pretty, --format nor --oneline option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See **git-config**(1) for more details.

With an optional <ref> argument, show this notes ref instead of the default notes ref(s). The ref is taken to be in refs/notes/ if it is not qualified.

Multiple --notes options can be combined to control which notes are being displayed.

Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above --notes option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes[=<ref>], --[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--relative-date

Synonym for --date=relative.

--date=(relative|local|default|iso|rfc|short|raw)

Only takes effect for dates shown in human-readable format, such as when using "--pretty". log.date config variable sets a default value for log command's --date option.

--date=relative shows dates relative to the current time, e.g. "2 hours ago".

--date=local shows timestamps in user's local timezone.

--date=iso (or --date=iso8601) shows timestamps in ISO 8601 format.

--date=rfc (or --date=rfc2822) shows timestamps in RFC 2822 format, often found in E-mail messages.

--date=short shows only date but not time, in YYYY-MM-DD format.

--date=raw shows the date in the internal raw git format %s %z format.

--date=default shows timestamps in the original timezone (either committer's or author's).

--parents

Print also the parents of the commit (in the form "commit parent..."). Also enables parent rewriting, see *History Simplification* below.

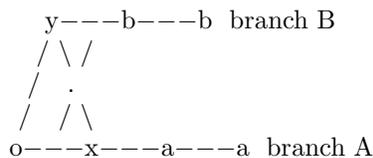
--children

Print also the children of the commit (in the form "commit child..."). Also enables parent rewriting, see *History Simplification* below.

--left-right

Mark which side of a symmetric diff a commit is reachable from. Commits from the left side are prefixed with < and those from the right with >. If combined with --boundary, those commits are prefixed with -.

For example, if you have this topology:



you would get an output like this:

```

$ git rev-list --left-right --boundary --pretty=oneline A...B

>bbbbbbb... 3rd on b
>bbbbbbb... 2nd on b
<aaaaaaa... 3rd on a
<aaaaaaa... 2nd on a
-yyy-yyyy... 1st on b
-xxxxxxx... 1st on a

```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly.

This enables parent rewriting, see *History Simplification* below.

This implies the --topo-order option by default, but the --date-order option may also be specified.

### Diff Formatting

Below are listed options that control the formatting of diff output. Some of them are specific to **git-rev-list(1)**, however other diff options may be given. See **git-diff-files(1)** for more options.

-c

With this option, diff output for a merge commit shows the differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time. Furthermore, it lists only files which were modified from all parents.

--cc

This flag implies the -c options and further compresses the patch output by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification.

-m

This flag makes the merge commits show the full diff like regular commits; for each merge

parent, a separate log entry and diff is generated. An exception is that only diff against the first parent is shown when *--first-parent* option is given; in that case, the output represents the changes the merge brought *into* the then-current branch.

- r Show recursive diffs.
- t Show the tree objects in the diff output. This implies *-r*.
- s Suppress diff output.

## PRETTY FORMATS

If the commit is a merge, and if the *pretty-format* is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the shas of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a *pretty.<name>* config option to either another format name, or a *format:* string, as described below (see **git-config**(1)). Here are the details of the built-in formats:

- 

*oneline*

<sha1> <title line>

This is designed to be as compact as possible.

- 

*short*

commit <sha1>  
Author: <author>

<title line>

- 

*medium*

commit <sha1>  
Author: <author>  
Date: <author date>

<title line>

<full commit message>

- 

*full*

commit <sha1>  
Author: <author>

Commit: <committer>

<title line>

<full commit message>

- 

*fuller*

commit <sha1>

Author: <author>

AuthorDate: <author date>

Commit: <committer>

CommitDate: <committer date>

<title line>

<full commit message>

- 

*email*

From <sha1> <date>

From: <author>

Date: <author date>

Subject: [PATCH] <title line>

<full commit message>

- 

*raw*

The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA1s are displayed in full, regardless of whether `--abbrev` or `--no-abbrev` are used, and *parents* information show the true parent commits, without taking grafts nor history simplification into account.

- 

*format:<string>*

The *format:<string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with `%n` instead of `\n`.

E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
```

```
The title was >>t4119: test autocomputing -p<n> for traditional diff input.<<
```

The placeholders are:

-

*%H*: commit hash

•

*%h*: abbreviated commit hash

•

*%T*: tree hash

•

*%t*: abbreviated tree hash

•

*%P*: parent hashes

•

*%p*: abbreviated parent hashes

•

*%an*: author name

•

*%aN*: author name (respecting `.mailmap`, see **git-shortlog(1)** or **git-blame(1)**)

•

*%ae*: author email

•

*%aE*: author email (respecting `.mailmap`, see **git-shortlog(1)** or **git-blame(1)**)

•

*%ad*: author date (format respects `--date=` option)

•

*%aD*: author date, RFC2822 style

•

*%ar*: author date, relative

•

*%at*: author date, UNIX timestamp

•

*%ai*: author date, ISO 8601 format

•

*%cn*: committer name

•

*%cN*: committer name (respecting `.mailmap`, see **git-shortlog(1)** or **git-blame(1)**)

•

*%ce*: committer email

•

*%cE*: committer email (respecting `.mailmap`, see **git-shortlog(1)** or **git-blame(1)**)

•

*%cd*: committer date

•

*%cD*: committer date, RFC2822 style

•

*%cr*: committer date, relative

•

*%ct*: committer date, UNIX timestamp

•

*%ci*: committer date, ISO 8601 format

•

*%d*: ref names, like the `--decorate` option of **git-log(1)**

•

*%e*: encoding

•

*%s*: subject

•

*%f*: sanitized subject line, suitable for a filename

•

*%b*: body

•

*%B*: raw body (unwrapped subject and body)

•

*%N*: commit notes

•

*%gD*: relog selector, e.g., `refs/stash@{1}`

•

*%gd*: shortened reflog selector, e.g., stash@{1}

- 

*%gs*: reflog subject

- 

*%Cred*: switch color to red

- 

*%Cgreen*: switch color to green

- 

*%Cblue*: switch color to blue

- 

*%Creset*: reset color

- 

*%C(...)*: color specification, as described in color.branch.\* config option

- 

*%m*: left, right or boundary mark

- 

*%n*: newline

- 

*%%*: a raw %

- 

*%x00*: print a byte from a hex code

- 

*%w([<w>[,<i1>[,<i2>]])*: switch line wrapping, like the `-w` option of **git-shortlog**(1).

### Note

Some placeholders may depend on other options given to the revision traversal engine. For example, the *%g\** reflog options will insert an empty string unless we are traversing reflog entries (e.g., by `git log -g`). The *%d* placeholder will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a + (plus sign) after % of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, line-feeds that immediately precede the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ' ' (space) after % of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

•

*tformat*:

The *tformat*: format works exactly like *format*:, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

```
$ git log -2 --pretty=format:%h 4da45bef \
  | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
4da45be
7134973 -- NO NEWLINE
```

```
$ git log -2 --pretty=tformat:%h 4da45bef \
  | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
4da45be
7134973
```

In addition, any unrecognized string that has a % in it is interpreted as if it has *tformat*: in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

## COMMON DIFF OPTIONS

- p, -u, --patch  
Generate patch (see section on generating patches).
- U<n>, --unified=<n>  
Generate diffs with <n> lines of context instead of the usual three. Implies -p.
- raw  
Generate the raw format.
- patch-with-raw  
Synonym for -p --raw.
- patience  
Generate a diff using the "patience diff" algorithm.
- stat[=<width>[,<name-width>]]  
Generate a diffstat. You can override the default output width for 80-column terminal by --stat=<width>. The width of the filename part can be controlled by giving another width to it separated by a comma.
- numstat  
Similar to --stat, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two - instead of saying 0 0.
- shortstat  
Output only the last line of the --stat format containing total number of modified files, as well as number of added and deleted lines.
- dirstat[=<param1,param2,...>]  
Output the distribution of relative amount of changes for each sub-directory. The behavior of --dirstat can be customized by passing it a comma separated list of parameters. The

defaults are controlled by the `diff.dirstat` configuration variable (see `git-config(1)`). The following parameters are available:

#### changes

Compute the `dirstat` numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

#### lines

Compute the `dirstat` numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

#### files

Compute the `dirstat` numbers by counting the number of files changed. Each changed file counts equally in the `dirstat` analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

#### cumulative

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

#### <limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

#### --summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

#### --patch-with-stat

Synonym for `-p --stat`.

#### -z

Separate the commits with NULs instead of with new newlines.

Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

#### --name-only

Show only names of changed files.

#### --name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

#### --submodule[=<format>]

Choose the output format for submodule differences. `<format>` can be one of *short* and *log*.

*short* just shows pairs of commit names, this format is used when this option is not given. *log* is the default value for this option and lists the commits in that commit range like the *summary* option of **git-submodule**(1) does.

`--color[=<when>]`

Show colored diff. The value must be always (the default for <when>), never, or auto. The default value is never.

`--no-color`

Turn off colored diff. It is the same as `--color=never`.

`--word-diff[=<mode>]`

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

*color*

Highlight changed words using only colors. Implies `--color`.

*plain*

Show words as `[-removed-]` and `{added}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

*porcelain*

Use a special line-based format intended for script consumption.

Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+/-/‘` character at the beginning of the line and extending to the end of the line. Newlines in the input are represented by a tilde `~` on a line of its own.

*none*

Disable word diff again.

Note that despite the name of the first mode, *color* is used to highlight the changed parts in all modes if enabled.

`--word-diff-regex=<regex>`

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `[[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

The regex can also be set via a diff driver or configuration option, see **gitattributes**(1) or **git-config**(1). Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

`--color-words[=<regex>]`

Equivalent to `--word-diff=color` plus (if a regex was specified)

`--word-diff-regex=<regex>`.

`--no-renames`

Turn off rename detection, even when the configuration file gives the default to do so.

`--check`

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with

non-zero status if problems are found. Not compatible with `--exit-code`.

`--full-index`

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

`--binary`

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

`--abbrev[=<n>]`

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

`-B[<n>][/ <m>], --break-rewrites[=[<n>][/ <m>]]`

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number *m* controls this aspect of the `-B` option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with `-M`, a totally-rewritten file is also considered as the source of a rename (usually `-M` only considers a file that disappeared as the source of a rename), and the number *n* controls this aspect of the `-B` option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

`-M[<n>], --find-renames[=<n>]`

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see `--follow`. If *n* is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed.

`-C[<n>], --find-copies[=<n>]`

Detect copies as well as renames. See also `--find-copies-harder`. If *n* is specified, it has the same meaning as for `-M<n>`.

`--find-copies-harder`

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

`-D, --irreversible-delete`

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` nor `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

`-l<num>`

The `-M` and `-C` options require  $O(n^2)$  processing time where  $n$  is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

- `--diff-filter=[(A|C|D|M|R|T|U|X|B)...[*]]`  
Select only files that are Added (A), Copied (C), Deleted (D), Modified (M), Renamed (R), have their type (i.e. regular file, symlink, submodule, ...) changed (T), are Unmerged (U), are Unknown (X), or have had their pairing Broken (B). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.
- `-S<string>`  
Look for differences that introduce or remove an instance of `<string>`. Note that this is different than the string simply appearing in diff output; see the *pickaxe* entry in **gitdiffcore(7)** for more details.
- `-G<regex>`  
Look for differences whose added or removed line matches the given `<regex>`.
- `--pickaxe-all`  
When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in `<string>`.
- `--pickaxe-regex`  
Make the `<string>` not a plain string but an extended POSIX regex to match.
- `-O<orderfile>`  
Output the patch in the order specified in the `<orderfile>`, which has one shell glob pattern per line.
- `-R`  
Swap two inputs; that is, show differences from index or on-disk file to tree contents.
- `--relative[=<path>]`  
When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a `<path>` as an argument.
- `-a, --text`  
Treat all files as text.
- `--ignore-space-at-eol`  
Ignore changes in whitespace at EOL.
- `-b, --ignore-space-change`  
Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.
- `-w, --ignore-all-space`  
Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.
- `--inter-hunk-context=<lines>`  
Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.
- `--exit-code`  
Make the program exit with codes similar to `diff(1)`. That is, it exits with 1 if there were differences and 0 means no differences.
- `--quiet`  
Disable all output of the program. Implies `--exit-code`.

- `--ext-diff`  
Allow an external diff helper to be executed. If you set an external diff driver with **gitattributes**(5), you need to use this option with **git-log**(1) and friends.
- `--no-ext-diff`  
Disallow external diff drivers.
- `--textconv`, `--no-textconv`  
Allow (or disallow) external text conversion filters to be run when comparing binary files. See **gitattributes**(5) for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for **git-diff**(1) and **git-log**(1), but not for **git-format-patch**(1) or diff plumbing commands.
- `--ignore-submodules[=<when>]`  
Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in **git-config**(1) or **gitmodules**(5). When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.
- `--src-prefix=<prefix>`  
Show the given source prefix instead of "a/".
- `--dst-prefix=<prefix>`  
Show the given destination prefix instead of "b/".
- `--no-prefix`  
Do not show any source or destination prefix.

For more detailed explanation on these common options, see also **gitdiffcore**(7).

## GENERATING PATCHES WITH -P

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a `-p` option, "git diff" without the `--raw` option, or "git log" with the `-p` option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the `GIT_EXTERNAL_DIFF` and the `GIT_DIFF_OPTS` environment variables.

What the `-p` option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

```
diff --git a/file1 b/file2
```

The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode <mode>
new mode <mode>
deleted file mode <mode>
new file mode <mode>
```

```

copy from <path>
copy to <path>
rename from <path>
rename to <path>
similarity index <number>
dissimilarity index <number>
index <hash>..<hash> <mode>

```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the a/ and b/ prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.
4. All the file1 files in the output refer to files before the commit, and all the file2 files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```

diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a

```

## COMBINED DIFF FORMAT

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with `git-diff(1)` or `git-show(1)`. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```

diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
}

- static void describe(char *arg)
- static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)

```

```

{
+   unsigned char sha1[20];
+   struct commit *cmit;
struct commit_list *list;
static int initialized = 0;
struct commit_name *n;

+   if (get_sha1(arg, sha1) < 0)
+       usage(describe_usage);
+   cmit = lookup_commit_reference(sha1);
+   if (!cmit)
+       usage(describe_usage);
+
if (!initialized) {
    initialized = 1;
    for_each_ref(get_name);
}

```

1. It is preceded with a "git diff" header, that looks like this (when `-c` option is used):

```
diff --combined file
```

or like this (when `--cc` option is used):

```
diff --cc file
```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

```
index <hash>,<hash>..<hash>
mode <mode>,<mode>..<mode>
new file mode <mode>
deleted file mode <mode>,<mode>
```

The mode `<mode>,<mode>..<mode>` line appears only if at least one of the `<mode>` is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two `<tree-ish>` and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

```
--- a/file
+++ b/file
```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

```
@@@ <from-file-range> <from-file-range> <to-file-range> @@@
```

There are (number of parents + 1) @ characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has - (minus — appears in A but removed in B), + (plus — missing in A but added to B), or " " (space — unchanged) prefix, this format compares two or more files file1, file2,... with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A - character in the column N means that the line appears in fileN but it does not appear in the result. A + character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two - removals from both file1 and file2, plus ++ to mean one line that was added does not appear in either file1 nor file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with +).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

## EXAMPLES

`git log --no-merges`

Show the whole commit history, but skip any merges

`git log v2.6.12.. include/scsi drivers/scsi`

Show all commits since version *v2.6.12* that changed any file in the include/scsi or drivers/scsi subdirectories

`git log --since="2 weeks ago" -- gitk`

Show the changes during the last two weeks to the file *gitk*. The "--" is necessary to avoid confusion with the **branch** named *gitk*

`git log --name-status release..test`

Show the commits that are in the "test" branch but not yet in the "release" branch, along with the list of paths each commit modifies.

`git log --follow builtin-rev-list.c`

Shows the commits that changed builtin-rev-list.c, including those commits that occurred before the file was given its present name.

`git log --branches --not --remotes=origin`

Shows all commits that are in any of local branches but not in any of remote-tracking branches for *origin* (what you have that origin doesn't).

`git log master --not --remotes=*/master`

Shows all commits that are in local master but not in any remote repository master branches.

`git log -p -m --first-parent`

Shows the history including change diffs, but only from the "main branch" perspective, skipping commits that come from merged branches, and showing full diffs of changes introduced by the merges. This makes sense only when following a strict policy of merging all topic branches when staying on a single integration branch.

## DISCUSSION

At the core level, git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What `readdir(2)` returns are what are recorded and compared with the data git keeps track of, which in turn are expected to be what `lstat(2)` and `creat(2)` accepts. There is no such thing as pathname encoding translation.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

Although we encourage that the commit log messages are encoded in UTF-8, both the core and git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, git does not forbid it. However, there are a few things to keep in mind.

1.

*git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have `i18n.commitencoding` in `.git/config` file, like this:

```
[i18n]
    commitencoding = ISO-8859-1
```

Commit objects created with the above setting record the value of `i18n.commitencoding` in its encoding header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2.

*git log*, *git show*, *git blame* and friends look at the encoding header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

```
[i18n]
    logoutputencoding = ISO-8859-1
```

If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

## CONFIGURATION

See **git-config(1)** for core variables and **git-diff(1)** for settings related to diff generation.

`format.pretty`

Default for the `--format` option. (See "PRETTY FORMATS" above.) Defaults to "medium".

`i18n.logOutputEncoding`

Encoding to use when displaying logs. (See "Discussion", above.) Defaults to the value of `i18n.commitEncoding` if set, UTF-8 otherwise.

`log.date`

Default format for human-readable dates. (Compare the `--date` option.) Defaults to "default", which means to write dates like Sat May 8 19:35:34 2010 -0500.

`log.showroot`

If false, *git log* and related commands will not treat the initial commit as a big creation event. Any root commits in `git log -p` output would be shown without a diff attached. The default

is true.

mailmap.file

See **git-shortlog(1)**.

notes.displayRef

Which refs, in addition to the default set by `core.notesRef` or `GIT_NOTES_REF`, to read notes from when showing commit messages with the *log* family of commands. See **git-notes(1)**.

May be an unabbreviated ref name or a glob and may be specified multiple times. A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the `--no-notes` option, overridden by the `GIT_NOTES_DISPLAY_REF` environment variable, and overridden by the `--notes=<ref>` option.

## GIT

Part of the **git(1)** suite