NAME

git-rev-list - Lists commit objects in reverse chronological order

SYNOPSIS

git rev-list [--max-count=<number>]
[--skip=<number>]
[--max-age=<timestamp>]
[--min-age=<timestamp>]
[--sparse]
[--merges]
[--no-merges]
[--min-parents=<number>]
[--no-min-parents]
[--max-parents=<number>]
[--no-max-parents]
[--first-parent]
[--remove-empty]
[--full-history]
[--not]
[--all]
[--branches[=<pattern>]]
[--tags[=<pattern>]]
[--remotes[=<pattern>]]
[--glob=<glob-pattern>]
[--ignore-missing]
[--stdin]
[--quiet]
[--topo-order]
[--parents]
[--timestamp]
[--left-right]
[--left-only]
[--right-only]
[--cherry-mark]
[--cherry-pick]
[--encoding=<encoding>]
[--(author|committer|grep)=<pattern>]
[--regexp-ignore-case | -i]
[--extended-regexp | -E]
[--fixed-strings | -F]
[--date=(local|relative|default|iso|rfc|short)]
[[--objects | --objects-edge] [--unpacked]]
[--pretty | --header]
[--bisect]
[--bisect-vars]
[--bisect-all]
[--merge]
[--reverse]
[--walk-reflogs]
[--no-walk] [--do-walk]
[--use-bitmap-index]
<commit>... [-- <paths>...]

DESCRIPTION

List commits that are reachable by following the parent links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them. The output is given in reverse chronological order by default.

You can think of this as a set operation. Commits given on the command line form a set of commits that are reachable from any of them, and then commits reachable from any of the ones given with ^ in front are subtracted from that set. The remaining commits are what comes out in the command's output. Various other options and paths parameters can be used to further limit the result.

Thus, the following command:

$ git rev-list foo bar ^baz

means list all the commits which are reachable from *foo* or *bar*, but not from *baz*.

A special notation <*commit1*>..<*commit2*> can be used as a short-hand for ^<commit1> <*commit2*>. For example, either of the following may be used interchangeably:

$ git rev-list origin..HEAD
$ git rev-list HEAD ^origin

Another special notation is <*commit1*>...<*commit2*> which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

$ git rev-list A B --not $(git merge-base --all A B)
$ git rev-list A...B

rev-list is a very essential Git command, since it provides the ability to build and traverse commit ancestry graphs. For this reason, it has a lot of different options that enables it to be used by commands as different as *git bisect* and *git repack*.

OPTIONS

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.

Using more options generally further limits the output (e.g. --since=<date1> limits to commits newer than <date1>, and using it with --grep=<pattern> further limits to commits whose log message has a line that matches <pattern>), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as --reverse.

-<number>, -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.

--max-age=<timestamp>, --min-age=<timestamp>
: Limit the commits output to specified time range.

--author=<pattern>, --committer=<pattern>
: Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one --author=<pattern>, commits whose author matches any of the given patterns are chosen (similarly for multiple

--committer=<pattern>).

--grep-reflog=<pattern>
Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one --grep-reflog, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless --walk-reflogs is in use.

--grep=<pattern>
Limit the commits output to ones with log message that matches the specified pattern (regular expression). With more than one --grep=<pattern>, commits whose message matches any of the given patterns are chosen (but see --all-match).

When --show-notes is in effect, the message from the notes as if it is part of the log message.

--all-match
Limit the commits output to ones that match all given --grep, instead of ones that match at least one.

-i, --regexp-ignore-case
Match the regular expression limiting patterns without regard to letter case.

--basic-regexp
Consider the limiting patterns to be basic regular expressions; this is the default.

-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).

--perl-regexp
Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--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 parent 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.

--exclude=<glob-pattern>
Do not include refs matching <*glob-pattern*> that the next --all, --branches, --tags, --remotes, or --glob would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next --all, --branches, --tags, --remotes, or --glob option (other options or arguments do not clear accumlated patterns).

The patterns given should not begin with refs/heads, refs/tags, or refs/remotes when applied to --branches, --tags, or --remotes, respectively, and they must begin with refs/ when applied to --glob or --all. If a trailing */** is intended, it must be given explicitly.

--ignore-missing
Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--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.

--quiet
Don't print anything to standard output. This form is primarily meant to allow the caller to test the exit status to see if a range of objects is fully connected (or not). It is faster than redirecting stdout to /dev/null as the output does not have to be formatted.

--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). However, it 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*, and *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 excluded boundary commits. Boundary commits are prefixed with -.

--use-bitmap-index
Try to speed up the traversal using the pack bitmap index (if one is available). Note that when traversing with --objects, trees and blobs will not have their associated path printed.

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:

```
.-A---M---N---O---P---Q
/ / / / / /
I B C D E Y
 / / / / /
‘------------ X
```

The horizontal line of history A---Q 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 “xyzzy”, and its merge P combines the strings to “quux xyzzy”. P is TREESAME to O, but not to E.
- X is an independent root commit that added a new file side, and Y modified it. Y is TREESAME to X. Its merge Q added side to P, and Q is TREESAME to P, but not to Y.

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:

```
.-A---N---O
/ / /
I---------D
```

  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 Q

  M was excluded because it is TREESAME to both parents. 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---Q
  / / / / /
  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 Es parent I. The same happened for C and N, and X, Y and Q.

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.

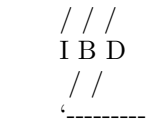
--simplify-merges
: First, build a history graph in the same way that --full-history with parent rewriting does (see above).

  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 or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
  - 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:

  ```
  .-A---M---N---O
  ```



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

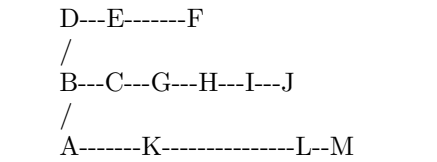
- Ns parent list had I removed, because it is an ancestor of the other parent M. Still, N remained because it is !TREESAME.
- Ps parent list similarly had I removed. P was then removed completely, because it had one parent and is TREESAME.
- Qs parent list had Y simplified to X. X was then removed, because it was a TREESAME root. Q was then removed completely, because it had one parent and is TREESAME.

Finally, there is a fifth simplification mode available:

--ancestry-path

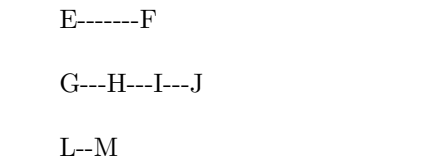
Limit the displayed commits to those directly on the ancestry chain between the “from” and “to” commits in the given commit range. I.e. only display commits that are ancestor of the “to” commit and descendants of the “from” commit.

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).

Bisection Helpers

--bisect

Limit output to the one commit object which is roughly halfway between included and excluded commits. Note that the bad bisection ref refs/bisect/bad is added to the included commits (if it exists) and the good bisection refs refs/bisect/good-* are added to the excluded commits (if they exist). Thus, supposing there are no refs in refs/bisect/, if

$ git rev-list --bisect foo ^bar ^baz

outputs *midpoint*, the output of the two commands

$ git rev-list foo ^midpoint
$ git rev-list midpoint ^bar ^baz

would be of roughly the same length. Finding the change which introduces a regression is thus reduced to a binary search: repeatedly generate and test new midpoint's until the commit chain is of length one.

--bisect-vars
This calculates the same as --bisect, except that refs in refs/bisect/ are not used, and except that this outputs text ready to be eval'ed by the shell. These lines will assign the name of the midpoint revision to the variable bisect_rev, and the expected number of commits to be tested after bisect_rev is tested to bisect_nr, the expected number of commits to be tested if bisect_rev turns out to be good to bisect_good, the expected number of commits to be tested if bisect_rev turns out to be bad to bisect_bad, and the number of commits we are bisecting right now to bisect_all.

--bisect-all
This outputs all the commit objects between the included and excluded commits, ordered by their distance to the included and excluded commits. Refs in refs/bisect/ are not used. The farthest from them is displayed first. (This is the only one displayed by --bisect.)

This is useful because it makes it easy to choose a good commit to test when you want to avoid to test some of them for some reason (they may not compile for example).

This option can be used along with --bisect-vars, in this case, after all the sorted commit objects, there will be the same text as if --bisect-vars had been used alone.

Commit Ordering

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

--date-order
Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order
Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order
Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

---1----2----4----7

3----5----6----8---

where the numbers denote the order of commit timestamps, git rev-list and friends with --date-order show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With --topo-order, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--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[=(sorted|unsorted)]
Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument unsorted is given, the commits are shown in the order they were given on the command line. Otherwise (if sorted or no argument was given), the commits are shown in reverse chronological order by commit time.

--do-walk
Overrides a previous --no-walk.

Commit Formatting

Using these options, **git-rev-list(1)** will act similar to the more specialized family of commit log tools: **git-log(1)**, **git-show(1)**, and **git-whatchanged(1)**

--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, or --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.

--show-signature
Check the validity of a signed commit object by passing the signature to gpg --verify and show the output.

--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 the 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 time zone.

--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 email messages.

--date=short shows only the date, but not the 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 time zone (either committer's or author's).

--header
Print the contents of the commit in raw-format; each record is separated with a NUL character.

--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.

--timestamp
Print the raw commit timestamp.

--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:

y---b---b branch B
/ /

```
/ .
/ /
o---x---a---a branch A
```

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
-yyyyyyy... 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.

--show-linear-break[=<barrier>]
: When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If <barrier> is specified, it is the string that will be shown instead of the default one.

--count
: Print a number stating how many commits would have been listed, and suppress all other output. When used together with --left-right, instead print the counts for left and right commits, separated by a tab. When used together with --cherry-mark, omit patch equivalent commits from these counts and print the count for equivalent commits separated by a tab.

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 sha1s 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 SHA-1s are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or 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
- *%GG*: raw verification message from GPG for a signed commit
- *%G?*: show G for a Good signature, B for a Bad signature, U for a good, untrusted signature and N for no signature
- *%GS*: show the name of the signer for a signed commit
- *%GK*: show the key used to sign a signed commit
- *%gD*: reflog selector, e.g., refs/stash@{1}
- *%gd*: shortened reflog selector, e.g., stash@{1}
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see **git-shortlog(1)** or **git-blame(1)**)
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see **git-shortlog(1)** or **git-blame(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; adding auto, at the beginning will emit color only when colors are enabled for log output (by color.diff, color.ui, or --color, and respecting the auto settings of the former if we are going to a terminal). auto alone (i.e. %C(auto)) will turn on auto coloring on the next placeholders until the color is switched again.
- *%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)**.
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

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 NEWLINEn unless /n/
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef
  | perl -pe $_ .=  -- NO NEWLINEn 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
  ```

GIT

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