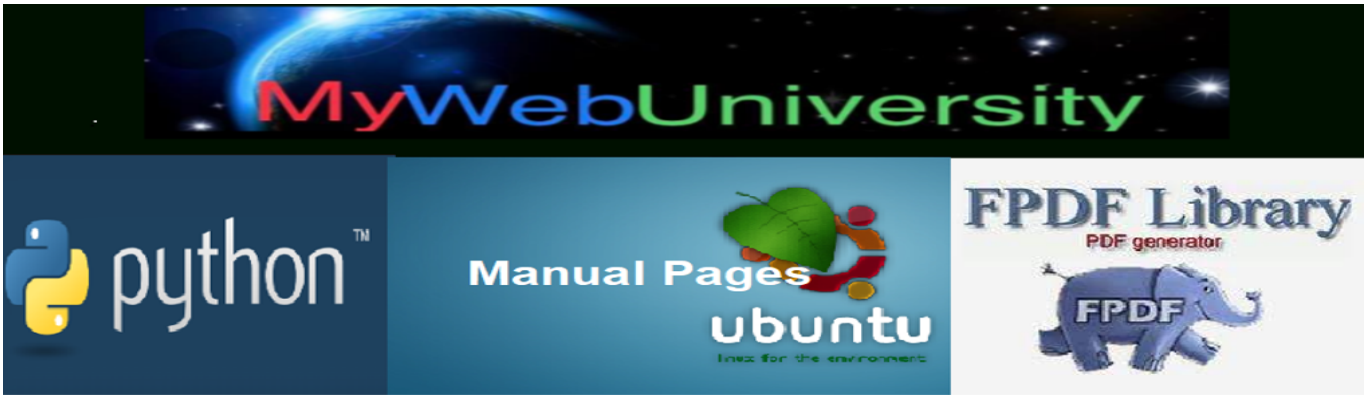


Rocky Enterprise Linux 9.2 Manual Pages on command 'git-shortlog.1'

$ man git-shortlog.1

GIT-SHORTLOG(1) Git Manual GIT-SHORTLOG(1)

NAME

git-shortlog - Summarize 'git log' output

SYNOPSIS

git shortlog [<options>] [<revision range>] [[--] <path>...]

git log --pretty=short | git shortlog [<options>]

DESCRIPTION

Summarizes git log output in a format suitable for inclusion in release announcements.

Each commit will be grouped by author and title.

Additionally, "[PATCH]" will be stripped from the commit description.

If no revisions are passed on the command line and either standard input is not a terminal

or there is no current branch, git shortlog will output a summary of the log read from

standard input, without reference to the current repository.

OPTIONS

-n, --numbered

Sort output according to the number of commits per author instead of author alphabetic

order.

-s, --summary

Suppress commit description and provide a commit count summary only.

-e, --email

Show the email address of each author.

--format[=<format>]

Instead of the commit subject, use some other information to describe each commit.

<format> can be any string accepted by the --format option of git log, such as * [%h] %s. (See the "PRETTY FORMATS" section of git-log(1).)

Each pretty-printed commit will be rewrapped before it is shown.

--group=<type>

Group commits based on <type>. If no --group option is specified, the default is author. <type> is one of:

? author, commits are grouped by author

? committer, commits are grouped by committer (the same as -c)

? trailer:<field>, the <field> is interpreted as a case-insensitive commit message trailer (see git-interpret-trailers(1)). For example, if your project uses Reviewed-by trailers, you might want to see who has been reviewing with git shortlog -ns --group=trailer:reviewed-by.

Note that commits that do not include the trailer will not be counted. Likewise, commits with multiple trailers (e.g., multiple signoffs) may be counted more than once (but only once per unique trailer value in that commit).

Shortlog will attempt to parse each trailer value as a name <email> identity. If successful, the mailmap is applied and the email is omitted unless the --email option is specified. If the value cannot be parsed as an identity, it will be taken literally and completely.

If --group is specified multiple times, commits are counted under each value (but again, only once per unique value in that commit). For example, git shortlog --group=author --group=trailer:co-authored-by counts both authors and co-authors.

-c, --committer

This is an alias for --group=committer.

-w[<width>[,<indent1>[,<indent2>]]]

Linewrap the output by wrapping each line at width. The first line of each entry is indented by indent1 spaces, and the second and subsequent lines are indented by indent2 spaces. width, indent1, and indent2 default to 76, 6 and 9 respectively.

If width is 0 (zero) then indent the lines of the output without wrapping them.

<revision range>

Show only commits in the specified revision range. When no <revision range> is specified, it defaults to HEAD (i.e. the whole history leading to the current commit).

origin..HEAD specifies all the commits reachable from the current commit (i.e. HEAD),

but not from origin. For a complete list of ways to spell <revision range>, see the "Specifying Ranges" section of gitrevisions(7).

[--] <path>...

Consider only commits that are enough to explain how the files that match the specified paths came to be.

Paths may need to be prefixed with -- to separate them from options or the revision range, when confusion arises.

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.

--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 --notes is in effect, the message from the notes is matched as if it were 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.

--invert-grep

Limit the commits output to ones with log message that do not match the pattern specified with --grep=<pattern>.

-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).

-P, --perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency.

If Git wasn?t compiled with support for them providing this option will cause it to die.

--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/, along with HEAD, 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 accumulated 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.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as <commit>.

--alternate-refs

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in objects/info/alternates. The set of included objects may be modified by core.alternateRefsCommand, etc. See git-config(1).

--single-worktree

By default, all working trees will be examined by the following options when there are more than one (see git-worktree(1)): --all, --reflog and --indexed-objects. This option forces them to examine the current working tree only.

--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). 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 difference, 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 and reference (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as ref@{Nth} (where Nth is the reverse-chronological index in the reflog) or as ref@{timestamp} (with the timestamp

for that entry), depending on a few rules:

1. If the starting point is specified as ref@{Nth}, show the index format.
2. If the starting point was specified as ref@{now}, show the timestamp format.
3. If neither was used, but --date was given on the command line, show the timestamp in the format requested by --date.
4. Otherwise, show the index format.

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).

Under --pretty=reference, this information will not be shown at all.

--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 -.

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)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful

for showing the merge commits that "first introduced" a change to a branch.

--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---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 E's 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:

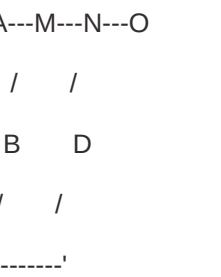
? Set C' to C.

? 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:



Note the major differences in N, P, and Q 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.

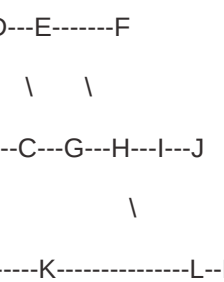
? Q's parent list had Y simplified to X. X was then removed, because it was a TREESAME root. Q was then removed completely, because it had one parent and is TREESAME.

There is another 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:

```
        E-------F
        \       \
         G---H---I---J
                     \
                      L--M
```

Before discussing another option, --show-pulls, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file?s simplified history. Let?s demonstrate a new example and show how options such as --full-history and --simplify-merges works in that case:

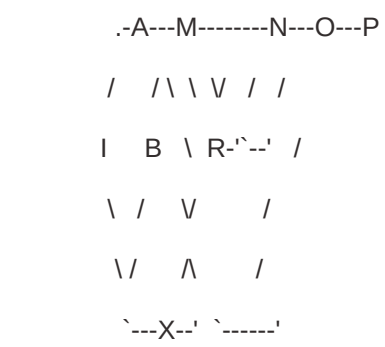
```
          .-A---M-----C--N---O---P
         /     / \  \  \/   /   /
        I     B   \  R-'`-Z'   /
         \   /     \/         /
          \ /      /\        /
           `---X--'  `---Y--'
```

For this example, suppose I created file.txt which was modified by A, B, and X in different ways. The single-parent commits C, Z, and Y do not change file.txt. The merge commit M was created by resolving the merge conflict to include both changes from A and B and hence is not TREESAME to either. The merge commit R, however, was created by ignoring the contents of file.txt at M and taking only the contents of file.txt at X. Hence, R is TREESAME to X but not M. Finally, the natural merge resolution to create N is to take the contents of file.txt at R, so N is TREESAME to R but not C. The merge commits O and P are TREESAME to their first parents, but not to their second parents, Z and Y respectively.

When using the default mode, N and R both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

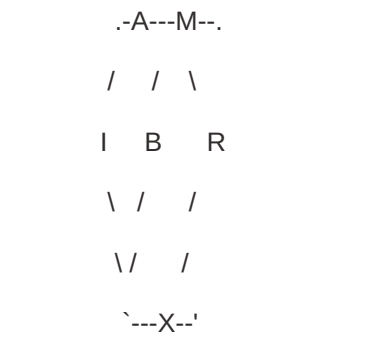
```
        I---X
```

When using --full-history, Git walks every edge. This will discover the commits A and B and the merge M, but also will reveal the merge commits O and P. With parent rewriting, the resulting graph is:



Here, the merge commits O and P contribute extra noise, as they did not actually contribute a change to file.txt. They only merged a topic that was based on an older version of file.txt. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: manu unrelated merges appear in the --full-history results.

When using the --simplify-merges option, the commits O and P disappear from the results. This is because the rewritten second parents of O and P are reachable from their first parents. Those edges are removed and then the commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit N, resulting in a history view as follows:



In this view, we see all of the important single-parent changes from A, B, and X. We also see the carefully-resolved merge M and the not-so-carefully-resolved merge R. This is usually enough information to determine why the commits A and B "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the --simplify-merges option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same

repository, it is important which merge commits introduced a change into an important branch. The problematic merge R above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge N was used to merge R and X into the important branch. This commit may have information about why the change X came to override the changes from A and B in its commit message.

--show-pulls

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by --show-pulls, the merge is treated as if it "pulled" the change from another branch. When using --show-pulls on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits R and N are included because they pulled the commits X and R into the base branch, respectively. These merges are the reason the commits A and B do not appear in the default history.

When --show-pulls is paired with --simplify-merges, the graph includes all of the necessary information:

```
 .-A---M--.   N
/     /    \ /
I     B     R
 \   /     /
  \ /     /
   `---X--'
```

Notice that since M is reachable from R, the edge from N to M was simplified away. However, N still appears in the history as an important commit because it "pulled" the change R into the main branch.

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).

See gitmailmap(5).

Note that if git shortlog is run outside of a repository (to process log contents on standard input), it will look for a .mailmap file in the current directory.

GIT

Part of the git(1) suite