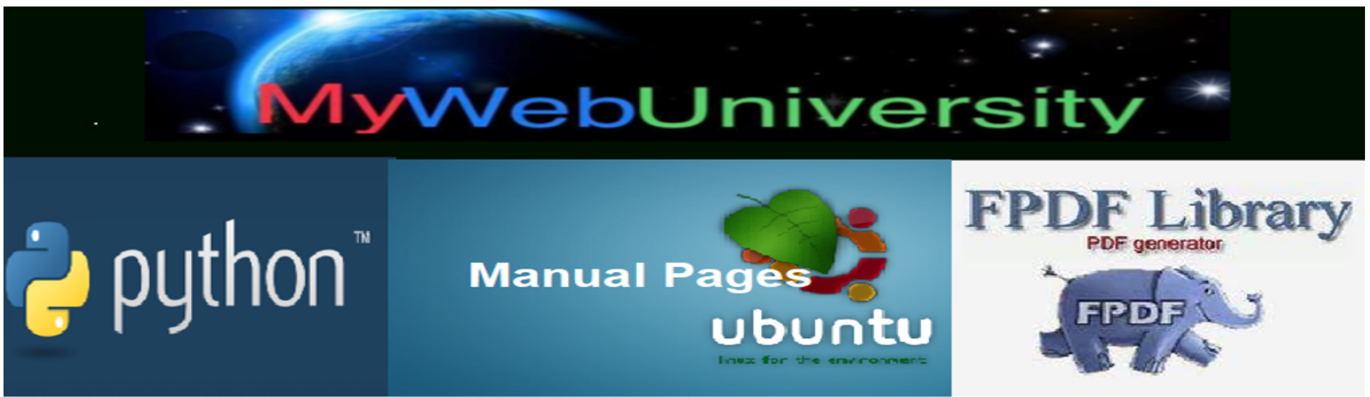


Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

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

$ man git-log.1

GIT-LOG(1) Git Manual GIT-LOG(1)

NAME

git-log - Show commit logs

SYNOPSIS

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

DESCRIPTION

Shows the commit logs.

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 reachable from any of the commits given

on the command line form a set, 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 log 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 log origin..HEAD
$ git log 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 log A B --not $(git merge-base --all A B)
$ git log A...B
```

The command takes options applicable to the git-rev-list(1) command to control what is shown and how, and options applicable to the git-diff(1) command to control how the changes each commit introduces are shown.

OPTIONS

--follow

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

--no-decorate, --decorate[=short|full|auto|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. If auto is specified, then if the output is going to a terminal, the ref names are shown as if short were given, otherwise no ref names are shown. The option --decorate is short-hand for --decorate=short. Default to configuration value of log.decorate if configured, otherwise, auto.

--decorate-refs=<pattern>, --decorate-refs-exclude=<pattern>

If no --decorate-refs is given, pretend as if all refs were included. For each candidate, do not use it for decoration if it matches any patterns given to --decorate-refs-exclude or if it doesn?t match any of the patterns given to --decorate-refs. The log.excludeDecoration config option allows excluding refs from the decorations, but an explicit --decorate-refs pattern will override a match in log.excludeDecoration.

--source

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

--[no-]mailmap, --[no-]use-mailmap

Use mailmap file to map author and committer names and email addresses to canonical real names and email addresses. See git-shortlog(1).

--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

Include a line ?log size <number>? in the output for each commit, where <number> is the length of that commit?s message in bytes. Intended to speed up tools that read log messages from git log output by allowing them to allocate space in advance.

-L<start>,<end>:<file>, -L:<funcname>:<file>

Trace the evolution of the line range given by <start>,<end>, or by the function name regex <funcname>, within the <file>. You may not give any pathspec limiters. This is currently limited to a walk starting from a single revision, i.e., you may only give zero or one positive revision arguments, and <start> and <end> (or <funcname>) must exist in the starting revision. You can specify this option more than once. Implies --patch. Patch output can be suppressed using --no-patch, but other diff formats (namely --raw, --numstat, --shortstat, --dirstat, --summary, --name-only, --name-status, --check) are not currently implemented.

<start> and <end> can take one of these forms:

? number

If <start> or <end> is a number, it specifies an absolute line number (lines count from 1).

? /regex/

This form will use the first line matching the given POSIX regex. If <start> is a regex, it will search from the end of the previous -L range, if any, otherwise from the start of file. If <start> is ^/regex/, it will search from the start of file. If <end> is a regex, it will search starting at the line given by <start>.

? +offset or -offset

This is only valid for <end> and will specify a number of lines before or after the line given by <start>.

If :<funcname> is given in place of <start> and <end>, it is a regular expression that denotes the range from the first funcname line that matches <funcname>, up to the next funcname line. :<funcname> searches from the end of the previous -L range, if any, otherwise from the start of file. ^:<funcname> searches from the start of file. The

function names are determined in the same way as git diff works out patch hunk headers (see Defining a custom hunk-header in gitattributes(5)).

<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>...

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.

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.

This option also changes default diff format for merge commits to first-parent, see --diff-merges=first-parent for details.

--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:

```
  .-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 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:

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

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.

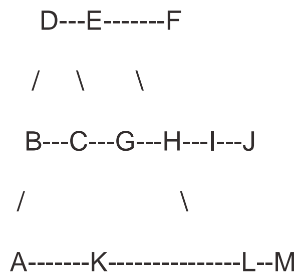
? Q's parent list had Y simplified to X. X was then removed, because it was a TREESAME root. Q was then removed completely, because it had one parent and is TREESAME.

There is another simplification mode available:

--ancestry-path

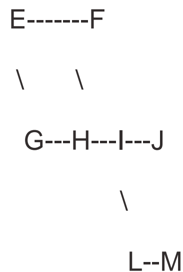
Limit the displayed commits to those directly on the ancestry chain between the ?from? and ?to? commits in the given commit range. I.e. only display commits that are ancestor of the ?to? commit and descendants of the ?from? commit.

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



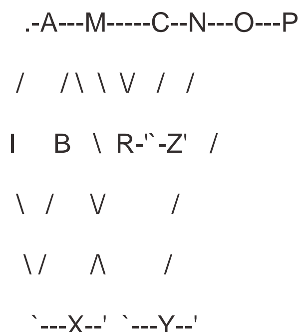
A regular D..M computes the set of commits that are ancestors of M, but excludes the ones that are ancestors of D. This is useful to see what happened to the history leading to M since D, in the sense that ?what does M have that did not exist in D?. The result in this example would be all the commits, 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:



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:



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:

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

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:

```
 .-A---M--.
/     /    \
I     B     R
\   /      /
 \ /      /
```

```
    `---X--'
```

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).

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 chosen to be shown (see Commit Limiting section above) in reverse order. Cannot be combined with --walk-reflogs.

Object Traversal

These options are mostly targeted for packing of Git repositories.

--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. Cannot be combined with --graph.

--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, reference, email, raw, format:<string> and tformat:<string>. When <format> is none of the above, and has %placeholder in it, it acts as if --pretty=tformat:<format> were given.

See the "PRETTY FORMATS" section for some additional details for each format. When =<format> part is omitted, it 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 a prefix that names the object uniquely. "--abbrev=<n>" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

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, either explicit or implied by other options such as "--oneline". It also overrides the log.abbrevCommit variable.

--oneline

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

--encoding=<encoding>

Commit objects record the character 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. Note that if an object claims to be encoded in X and we are outputting in X, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if iconv(3) fails to convert the commit, we will quietly output the original object verbatim.

--expand-tabs=<n>, --expand-tabs, --no-expand-tabs

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is multiple of <n>) in the log message before showing it in the output. --expand-tabs is a short-hand for --expand-tabs=8, and --no-expand-tabs is a short-hand for --expand-tabs=0, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. medium, which is the default, full, and fuller).

--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, use the ref to find the notes to display. The ref can specify the full refname when it begins with refs/notes/; when it begins with notes/, refs/ and otherwise refs/notes/ is prefixed to form a full name of the ref.

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=<format>

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. By default, dates are shown in the original time zone (either committer?s or
author?s). If -local is appended to the format (e.g., iso-local), the user?s local
time zone is used instead.

--date=relative shows dates relative to the current time, e.g. ?2 hours ago?. The
-local option has no effect for --date=relative.

--date=local is an alias for --date=default-local.

--date=iso (or --date=iso8601) shows timestamps in a ISO 8601-like format. The
differences to the strict ISO 8601 format are:

? a space instead of the T date/time delimiter

? a space between time and time zone

? no colon between hours and minutes of the time zone

--date=iso-strict (or --date=iso8601-strict) shows timestamps in strict 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 as seconds since the epoch (1970-01-01 00:00:00 UTC),
followed by a space, and then the timezone as an offset from UTC (a + or - with four
digits; the first two are hours, and the second two are minutes). I.e., as if the
timestamp were formatted with strftime("%s %z")). Note that the -local option does not
affect the seconds-since-epoch value (which is always measured in UTC), but does
switch the accompanying timezone value.

--date=human shows the timezone if the timezone does not match the current time-zone,
and doesn?t print the whole date if that matches (ie skip printing year for dates that
are "this year", but also skip the whole date itself if it?s in the last few days and
we can just say what weekday it was). For older dates the hour and minute is also

omitted.

--date=unix shows the date as a Unix epoch timestamp (seconds since 1970). As with

--raw, this is always in UTC and therefore -local has no effect.

--date=format:... feeds the format ... to your system strftime, except for %z and

%Z, which are handled internally. Use --date=format:%c to show the date in your system

locale?s preferred format. See the strftime manual for a complete list of format

placeholders. When using -local, the correct syntax is --date=format-local:....

--date=default is the default format, and is similar to --date=rfc2822, with a few

exceptions:

? there is no comma after the day-of-week

? the time zone is omitted when the local time zone is used

--parents

Print also the parents of the commit (in the form "commit parent..."). Also enables

parent rewriting, see History Simplification above.

--children

Print also the children of the commit (in the form "commit child..."). Also enables

parent rewriting, see History Simplification above.

--left-right

Mark which side of a symmetric difference 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. Cannot be combined with --no-walk.

This enables parent rewriting, see History Simplification above.

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.

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 hashes 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

<hash> <title line>

This is designed to be as compact as possible.

? short

commit <hash>

Author: <author>

<title line>

? medium

commit <hash>

Author: <author>

Date: <author date>

<title line>

<full commit message>

? full

commit <hash>

Author: <author>

Commit: <committer>

<title line>

<full commit message>

? fuller

commit <hash>

Author: <author>

AuthorDate: <author date>

Commit: <committer>

CommitDate: <committer date>

<title line>

<full commit message>

? reference

<abbrev hash> (<title line>, <short author date>)

This format is used to refer to another commit in a commit message and is the same as

--pretty='format:%C(auto)%h (%s, %ad)'. By default, the date is formatted with

--date=short unless another --date option is explicitly specified. As with any format:

with format placeholders, its output is not affected by other options like --decorate

and --walk-reflogs.

? email

From <hash> <date>

From: <author>

Date: <author date>

Subject: [PATCH] <title line>

<full commit message>

? mboxrd

Like email, but lines in the commit message starting with "From " (preceded by zero or

more ">") are quoted with ">" so they aren?t confused as starting a new commit.

? raw

The raw format shows the entire commit exactly as stored in the commit object.

Notably, the hashes 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. Note that this format affects

the way commits are displayed, but not the way the diff is shown e.g. with git log

--raw. To get full object names in a raw diff format, use --no-abbrev.

? 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:

? Placeholders that expand to a single literal character:

%n

newline

%%

a raw %

%x00

print a byte from a hex code

? Placeholders that affect formatting of later placeholders:

%Cred

switch color to red

%Cgreen

switch color to green

%Cblue

switch color to blue

%Creset

reset color

%C(...)

color specification, as described under Values in the "CONFIGURATION FILE" section of git-config(1). By default, colors are shown only when 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). %C(auto,...) is accepted as a historical synonym for the default (e.g., %C(auto,red)). Specifying %C(always,...) will show the colors even when color is not otherwise enabled (though consider just using --color=always to enable color for the whole output, including this format and anything else git might color). 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

%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)

? Placeholders that expand to information extracted from the commit:

%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))

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see %al) 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-like format

%aI

author date, strict ISO 8601 format

%as

author date, short format (YYYY-MM-DD)

%ah

author date, human style (like the --date=human option of git-rev-list(1))

%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))

%cl

committer email local-part (the part before the @ sign)

%cL

committer local-part (see %cl) respecting .mailmap, see git-shortlog(1) or git-blame(1))

%cd

committer date (format respects --date= option)

%cD

committer date, RFC2822 style

%cr

committer date, relative

%ct

committer date, UNIX timestamp

%ci

committer date, ISO 8601-like format

%cI

committer date, strict ISO 8601 format

%cs

committer date, short format (YYYY-MM-DD)

%ch

committer date, human style (like the --date=human option of git-rev-list(1))

%d

ref names, like the --decorate option of git-log(1)

%D

ref names without the " (", ")" wrapping.

%(describe[:options])

human-readable name, like git-describe(1); empty string for undescribable commits. The describe string may be followed by a colon and zero or more comma-separated options. Descriptions can be inconsistent when tags are added or removed at the same time.

? match=<pattern>: Only consider tags matching the given glob(7) pattern, excluding the "refs/tags/" prefix.

? exclude=<pattern>: Do not consider tags matching the given glob(7) pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like git log --source), only works with git log

%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 (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) 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

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., refs/stash@{1} or refs/stash@{2 minutes ago}; the format follows the rules described for the -g option. The portion before the @ is the refname as given on the command line (so git log -g refs/heads/master would yield refs/heads/master@{0}).

%gd

shortened reflog selector; same as %gD, but the refname portion is shortened for human readability (so refs/heads/master becomes just master).

%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

%(trailers[:options])

display the trailers of the body as interpreted by git-interpret-trailers(1). The trailers string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times the last occurrence wins.

The boolean options accept an optional value [=<BOOL>]. The values true, false, on, off etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in git-config(1). If a boolean option is given with no value, it?s enabled.

? key=<K>: only show trailers with specified key. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the only option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with only=false. E.g., %(trailers:key=Reviewed-by) shows trailer lines with key Reviewed-by.

? only[=<BOOL>]: select whether non-trailer lines from the trailer block should be included.

? separator=<SEP>: specify a separator inserted between trailer lines. When this option is not given each trailer line is terminated with a line feed character. The string SEP may contain the literal formatting codes described above. To use comma as separator one must use %x2C as it would otherwise be parsed as next option. E.g., %(trailers:key=Ticket,separator=%x2C) shows all trailer lines whose key is "Ticket" separated by a comma and a space.

? unfold[=<BOOL>]: make it behave as if interpret-trailer?s --unfold option was given. E.g., %(trailers:only,unfold=true) unfolds and shows all trailer lines.

? keyonly[=<BOOL>]: only show the key part of the trailer.

? valueonly[=<BOOL>]: only show the value part of the trailer.

? key_value_separator=<SEP>: specify a separator inserted between trailer lines. When this option is not given each trailer key-value pair is separated by ": ". Otherwise it shares the same semantics as separator=<SEP> above.

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 and %D placeholders 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, all consecutive line-feeds immediately preceding 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
```

DIFF FORMATTING

By default, git log does not generate any diff output. The options below can be used to show the changes made by each commit.

Note that unless one of --diff-merges variants (including short -m, -c, and --cc options) is explicitly given, merge commits will not show a diff, even if a diff format like --patch is selected, nor will they match search options like -S. The exception is when --first-parent is in use, in which case first-parent is the default format.

-p, -u, --patch

Generate patch (see section on generating patches).

-s, --no-patch

Suppress diff output. Useful for commands like git show that show the patch by default, or to cancel the effect of --patch.

--diff-merges=(off|none|on|first-parent|1|separate|m|combined|c|dense-combined|cc),

--no-diff-merges

Specify diff format to be used for merge commits. Default is off unless --first-parent is in use, in which case first-parent is the default.

--diff-merges=(off|none), --no-diff-merges

Disable output of diffs for merge commits. Useful to override implied value.

--diff-merges=on, --diff-merges=m, -m

This option makes diff output for merge commits to be shown in the default format. -m will produce the output only if -p is given as well. The default format could be changed using log.diffMerges configuration parameter, which default value is separate.

--diff-merges=first-parent, --diff-merges=1

This option makes merge commits show the full diff with respect to the first parent only.

--diff-merges=separate

This makes merge commits show the full diff with respect to each of the parents.

Separate log entry and diff is generated for each parent.

--diff-merges=combined, --diff-merges=c, -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. -c implies -p.

--diff-merges=dense-combined, --diff-merges=cc, --cc

With this option the output produced by --diff-merges=combined is further compressed by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification. --cc implies -p.

--combined-all-paths

This flag causes combined diffs (used for merge commits) to list the name of the file from all parents. It thus only has effect when --diff-merges=[dense-]combined is in use, and is likely only useful if filename changes are detected (i.e. when either rename or copy detection have been requested).

-U<n>, --unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies --patch.

--output=<file>

Output to a specific file instead of stdout.

--output-indicator-new=<char>, --output-indicator-old=<char>,

--output-indicator-context=<char>

Specify the character used to indicate new, old or context lines in the generated patch. Normally they are +, - and ' ' respectively.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of git-diff(1). This is different from showing the log itself in raw format, which you can achieve with --format=raw.

--patch-with-raw

Synonym for -p --raw.

-t

Show the tree objects in the diff output.

--indent-heuristic

Enable the heuristic that shifts diff hunk boundaries to make patches easier to read.

This is the default.

--no-indent-heuristic

Disable the indent heuristic.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--anchored=<text>

Generate a diff using the "anchored diff" algorithm.

This option may be specified more than once.

If a line exists in both the source and destination, exists only once, and starts with this text, this algorithm attempts to prevent it from appearing as a deletion or addition in the output. It uses the "patience diff" algorithm internally.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

default, myers

The basic greedy diff algorithm. Currently, this is the default.

minimal

Spend extra time to make sure the smallest possible diff is produced.

patience

Use "patience diff" algorithm when generating patches.

histogram

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured the diff.algorithm variable to a non-default value and want to use the default one, then you have to use --diff-algorithm=default option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by <width>.

The width of the filename part can be limited by giving another width <name-width> after a comma. The width of the graph part can be limited by using --stat-graph-width=<width> (affects all commands generating a stat graph) or by setting diff.statGraphWidth=<width> (does not affect git format-patch). By giving a third parameter <count>, you can limit the output to the first <count> lines, followed by ... if there are more.

These parameters can also be set individually with --stat-width=<width>, --stat-name-width=<name-width> and --stat-count=<count>.

--compact-summary

Output a condensed summary of extended header information such as file creations or deletions ("new" or "gone", optionally "+l" if it?s a symlink) and mode changes ("+x" or "-x" for adding or removing executable bit respectively) in diffstat. The information is put between the filename part and the graph part. Implies --stat.

--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.

-X[<param1,param2,...>], --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.

--cumulative

Synonym for --dirstat=cumulative

--dirstat-by-file[=<param1,param2>...]

Synonym for --dirstat=files,param1,param2...

--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, pathnames with "unusual" characters are quoted as explained for the configuration variable core.quotePath (see git-config(1)).

--name-only

Show only names of changed files. The file names are often encoded in UTF-8. For more information see the discussion about encoding in the git-log(1) manual page.

--name-status

Show only names and status of changed files. See the description of the --diff-filter option on what the status letters mean. Just like --name-only the file names are often encoded in UTF-8.

--submodule[=<format>]

Specify how differences in submodules are shown. When specifying --submodule=short the short format is used. This format just shows the names of the commits at the beginning and end of the range. When --submodule or --submodule=log is specified, the log format is used. This format lists the commits in the range like git-submodule(1) summary does. When --submodule=diff is specified, the diff format is used. This format shows an inline diff of the changes in the submodule contents between the commit range. Defaults to diff.submodule or the short format if the config option is unset.

--color[=<when>]

Show colored diff. --color (i.e. without =<when>) is the same as --color=always. <when> can be one of always, never, or auto.

--no-color

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

--color-moved[=<mode>]

Moved lines of code are colored differently. The <mode> defaults to no if the option is not given and to zebra if the option with no mode is given. The mode must be one of:

no

Moved lines are not highlighted.

default

Is a synonym for zebra. This may change to a more sensible mode in the future.

plain

Any line that is added in one location and was removed in another location will be colored with color.diff.newMoved. Similarly color.diff.oldMoved will be used for

removed lines that are added somewhere else in the diff. This mode picks up any moved line, but it is not very useful in a review to determine if a block of code was moved without permutation.

blocks

Blocks of moved text of at least 20 alphanumeric characters are detected greedily. The detected blocks are painted using either the color.diff.{old,new}Moved color. Adjacent blocks cannot be told apart.

zebra

Blocks of moved text are detected as in blocks mode. The blocks are painted using either the color.diff.{old,new}Moved color or color.diff.{old,new}MovedAlternative. The change between the two colors indicates that a new block was detected.

dimmed-zebra

Similar to zebra, but additional dimming of uninteresting parts of moved code is performed. The bordering lines of two adjacent blocks are considered interesting, the rest is uninteresting. dimmed_zebra is a deprecated synonym.

--no-color-moved

Turn off move detection. This can be used to override configuration settings. It is the same as --color-moved=no.

--color-moved-ws=<modes>

This configures how whitespace is ignored when performing the move detection for --color-moved. These modes can be given as a comma separated list:

no

Do not ignore whitespace when performing move detection.

ignore-space-at-eol

Ignore changes in whitespace at EOL.

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.

ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

allow-indentation-change

Initially ignore any whitespace in the move detection, then group the moved code blocks only into a block if the change in whitespace is the same per line. This is incompatible with the other modes.

--no-color-moved-ws

Do not ignore whitespace when performing move detection. This can be used to override configuration settings. It is the same as --color-moved-ws=no.

--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.

For example, --word-diff-regex=. will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes(5) 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.

--[no-]rename-empty

Whether to use empty blobs as rename source.

--check

Warn if changes introduce conflict markers or whitespace errors. What are considered whitespace errors is controlled by core.whitespace configuration. By default, trailing whitespaces (including lines that consist solely 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.

--ws-error-highlight=<kind>

Highlight whitespace errors in the context, old or new lines of the diff. Multiple values are separated by comma, none resets previous values, default reset the list to new and all is a shorthand for old,new,context. When this option is not given, and the configuration variable diff.wsErrorHighlight is not set, only whitespace errors in new lines are highlighted. The whitespace errors are colored with color.diff.whitespace.

--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. Implies --patch.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show the shortest prefix that is at least <n> hexdigits long that uniquely refers the object. In diff-patch output format, --full-index takes higher precedence, i.e. if --full-index is specified, full blob names will be shown regardless of --abbrev. 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. Without a % sign, the number is to be read as a fraction, with a decimal point before it. I.e., -M5 becomes 0.5, and is thus the same as -M50%. Similarly, -M05 is the same as -M5%. To limit detection to exact renames, use -M100%. The default similarity index is 50%.

-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 or git apply; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lacks 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 involve some preliminary steps that can detect subsets of renames/copies cheaply, followed by an exhaustive fallback portion that compares all remaining unpaired destinations to all relevant sources. (For renames, only remaining unpaired sources are relevant; for copies, all original sources are relevant.) For N sources and destinations, this exhaustive check is O(N^2). This option prevents the exhaustive portion of rename/copy detection from running if the number of source/destination files involved exceeds the specified number. Defaults to diff.renameLimit. Note that a value of 0 is treated as unlimited.

--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.

Also, these upper-case letters can be downcased to exclude. E.g. --diff-filter=ad excludes added and deleted paths.

Note that not all diffs can feature all types. For instance, diffs from the index to the working tree can never have Added entries (because the set of paths included in the diff is limited by what is in the index). Similarly, copied and renamed entries cannot appear if detection for those types is disabled.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter?s use.

It is useful when you?re looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into -S, and keep going until you get the very first version of the block.

Binary files are searched as well.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between -S<regex> --pickaxe-regex and -G<regex>, consider a commit with the following diff in the same file:

```
+    return frotz(nitfol, two->ptr, 1, 0);
...
-    hit = frotz(nitfol, mf2.ptr, 1, 0);
```

While git log -G"frotz\(nitfol" will show this commit, git log -S"frotz\(nitfol" --pickaxe-regex will not (because the number of occurrences of that string did not change).

Unless --text is supplied patches of binary files without a textconv filter will be ignored.

See the pickaxe entry in gitdiffcore(7) for more information.

--find-object=<object-id>

Look for differences that change the number of occurrences of the specified object.

Similar to -S, just the argument is different in that it doesn?t search for a specific string but for a specific object id.

The object can be a blob or a submodule commit. It implies the -t option in git-log to

also find trees.

--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

Treat the <string> given to -S as an extended POSIX regular expression to match.

-O<orderfile>

Control the order in which files appear in the output. This overrides the diff.orderFile configuration variable (see git-config(1)). To cancel diff.orderFile, use -O/dev/null.

The output order is determined by the order of glob patterns in <orderfile>. All files with pathnames that match the first pattern are output first, all files with pathnames that match the second pattern (but not the first) are output next, and so on. All files with pathnames that do not match any pattern are output last, as if there was an implicit match-all pattern at the end of the file. If multiple pathnames have the same rank (they match the same pattern but no earlier patterns), their output order relative to each other is the normal order.

<orderfile> is parsed as follows:

? Blank lines are ignored, so they can be used as separators for readability.

? Lines starting with a hash ("#") are ignored, so they can be used for comments. Add a backslash ("\") to the beginning of the pattern if it starts with a hash.

? Each other line contains a single pattern.

Patterns have the same syntax and semantics as patterns used for fnmatch(3) without the FNM_PATHNAME flag, except a pathname also matches a pattern if removing any number of the final pathname components matches the pattern. For example, the pattern "foo*bar" matches "fooasdfbar" and "foo/bar/baz/asdf" but not "foobarx".

--skip-to=<file>, --rotate-to=<file>

Discard the files before the named <file> from the output (i.e. skip to), or move them to the end of the output (i.e. rotate to). These were invented primarily for use of the git difftool command, and may not be very useful otherwise.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>], --no-relative

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. --no-relative can be used to countermand both diff.relative config option and previous --relative.

-a, --text

Treat all files as text.

--ignore-cr-at-eol

Ignore carriage-return at the end of line when doing a comparison.

--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.

--ignore-blank-lines

Ignore changes whose lines are all blank.

-I<regex>, --ignore-matching-lines=<regex>

Ignore changes whose all lines match <regex>. This option may be specified more than once.

--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. Defaults to diff.interHunkContext or 0 if the config option is unset.

-W, --function-context

Show whole function as context lines for each change. The function names are determined in the same way as git diff works out patch hunk headers (see Defining a custom hunk-header in gitattributes(5)).

--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.

--line-prefix=<prefix>

Prepend an additional prefix to every line of output.

--ita-invisible-in-index

By default entries added by "git add -N" appear as an existing empty file in "git diff" and a new file in "git diff --cached". This option makes the entry appear as a new file in "git diff" and non-existent in "git diff --cached". This option could be reverted with --ita-visible-in-index. Both options are experimental and could be

removed in future.

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

GENERATING PATCH TEXT WITH -P

Running git-diff(1), git-log(1), git-show(1), git-diff-index(1), git-diff-tree(1), or git-diff-files(1) with the -p option produces patch text. You can customize the creation of patch text via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables (see git(1)), and the diff attribute (see gitattributes(5)).

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 blob object names 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. Pathnames with "unusual" characters are quoted as explained for the configuration variable core.quotePath (see git-config(1)).

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
```

5. Hunk headers mention the name of the function to which the hunk applies. See "Defining a custom hunk-header" in gitattributes(5) for details of how to tailor to this to specific languages.

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 suitable --diff-merges option to any of these commands to force generation of diffs in specific format.

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 the -c option is used):

   diff --combined file

   or like this (when the --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.

However, if the --combined-all-paths option is provided, instead of a two-line from-file/to-file you get a N+1 line from-file/to-file header, where N is the number of parents in the merge commit

--- a/file

--- a/file

--- a/file

+++ b/file

This extended format can be useful if rename or copy detection is active, to allow you to see the original name of the file in different parents.

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 to be applied. 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 or 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.

git log -L '/int main/',/^}/:main.c

Shows how the function main() in the file main.c evolved over time.

git log -3

Limits the number of commits to show to 3.

DISCUSSION

Git is to some extent character encoding agnostic.

? The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.

? Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (.git/config (see git-config(1)), gitignore(5), gitattributes(5) and gitmodules(5)).

Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

? Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but not UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

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, and 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.

If the format is set to "auto:foo" and the pager is in use, format "foo" will be the used for the date format. Otherwise "default" will be used.

log.follow

If true, git log will act as if the --follow option was used when a single <path> is given. This has the same limitations as --follow, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

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.

log.showSignature

If true, git log and related commands will act as if the --show-signature option was passed to them.

mailmap.*

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

Git 2.34.1 07/07/2023 GIT-LOG(1)