



Linux Ubuntu 26.04 Manual Pages on command 'git-replay.1'

\$ man git-replay.1

GIT-REPLAY(1)

Git Manual

GIT-REPLAY(1)

NAME

git-replay - EXPERIMENTAL: Replay commits on a new base, works with bare repos too

SYNOPSIS

(EXPERIMENTAL!) git replay [--contained] --onto <newbase> | --advance <branch> [--ref-action[=<mode>]]

<revision-range>

DESCRIPTION

Takes a range of commits and replays them onto a new location. Leaves the working tree and the index untouched. By default, updates the relevant references using an atomic transaction (all refs update or none). Use --ref-action=print to avoid automatic ref updates and instead get update commands that can be piped to git update-ref --stdin (see the OUTPUT section below).

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

OPTIONS

--onto <newbase>

Starting point at which to create the new commits. May be any valid commit, and not just an existing branch name.

When --onto is specified, the branch(es) in the revision range will be updated to point at the new commits, similar to the way git rebase --update-refs updates multiple branches in the affected range.

--advance <branch>

Starting point at which to create the new commits; must be a branch name.

The history is replayed on top of the <branch> and <branch> is updated to point at the

tip of the resulting history. This is different from `--onto`, which uses the target only as a starting point without updating it.

`--contained`

Update all branches that point at commits in `<revision-range>`. Requires `--onto`.

`--ref-action[=<mode>]`

Control how references are updated. The mode can be:

? `update` (default): Update refs directly using an atomic transaction. All refs are updated or none are (all-or-nothing behavior).

? `print`: Output update-ref commands for pipeline use. This is the traditional behavior where output can be piped to `git update-ref --stdin`.

The default mode can be configured via the `replay.refAction` configuration variable.

`<revision-range>`

Range of commits to replay; see "Specifying Ranges" in `git-rev-parse(1)`. In `--advance <branch>` mode, the range should have a single tip, so that it's clear to which tip the advanced `<branch>` should point.

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 output to `<number>` commits.

`--skip=<number>`

Skip `<number>` commits before starting to show the commit output.

`--since=<date>`, `--after=<date>`

Show commits more recent than `<date>`.

`--since-as-filter=<date>`

Show all commits more recent than `<date>`. This visits all commits in the range, rather than stopping at the first commit which is older than `<date>`.

`--until=<date>`, `--before=<date>`

Show commits older than <date>.

--author=<pattern>, --committer=<pattern>

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

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the <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 a log message that matches the <pattern> regular expression. With more than one --grep=<pattern>, commits whose message matches any of the <pattern> 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 a 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`

When finding commits to include, 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.

`--exclude-first-parent-only`

When finding commits to exclude (with a `^`), follow only the first parent commit upon

seeing a merge commit. This can be used to find the set of changes in a topic branch

from the point where it diverged from the remote branch, given that arbitrary merges can

be valid topic branch changes.

`--not`

Reverses the meaning of the `^` prefix (or lack thereof) for all following revision

specifiers, up to the next `--not`. When used on the command line before `--stdin`, the

revisions passed through stdin will not be affected by it. Conversely, when passed via standard input, the revisions passed on the command line will not be affected by it.

`--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.

`--exclude-hidden=(fetch|receive|uploadpack)`

Do not include refs that would be hidden by git-fetch, git-receive-pack or git-upload-pack by consulting the appropriate fetch.hideRefs, receive.hideRefs or

uploadpack.hideRefs configuration along with transfer.hideRefs (see git-config(1)). This option affects the next pseudo-ref option --all or --glob and is cleared after processing them.

--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 getting arguments from the command line, read them from standard input as well. This accepts commits and pseudo-options like --all and --glob=. When a -- separator is seen, the following input is treated as paths and used to limit the result. Flags like --not which are read via standard input are only respected for arguments passed in the same way and will not influence any subsequent command line arguments.

--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`

Show commits touching conflicted paths in the range `HEAD...<other>`, where `<other>` is the first existing pseudoref in `MERGE_HEAD`, `CHERRY_PICK_HEAD`, `REVERT_HEAD` or `REBASE_HEAD`.

Only works when the index has unmerged entries. This option can be used to show relevant commits when resolving conflicts from a 3-way 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[=<commit>]

When given a range of commits to display (e.g. <commit1>..<commit2> or <commit2> ^<commit1>), and a commit <commit> in that range, only display commits in that range that are ancestors of <commit>, descendants of <commit>, or <commit> itself. If no commit is specified, use <commit1> (the excluded part of the range) as <commit>. Can be passed multiple times; if so, a commit is included if it is any of the commits given or if it is an ancestor or descendant of one of them.

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 xyzy, and its merge P combines the strings to quux xyzy. 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[=<commit>]

Limit the displayed commits to those which are an ancestor of <commit>, or which are a descendant of <commit>, or are <commit> itself.

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

```

D---E-----F
 /   \   \
B---C---G---H---I---J
 /           \
A-----K-----L--M

```

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

When we want to find out what commits in M are contaminated with the bug introduced by D and need fixing, however, we might want to view only the subset of D..M that are actually descendants of D, i.e. excluding C and K. This is exactly what the

--ancestry-path option does. Applied to the D..M range, it results in:

```

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

```

We can also use `--ancestry-path=D` instead of `--ancestry-path` which means the same thing when applied to the D..M range but is just more explicit.

If we instead are interested in a given topic within this range, and all commits affected by that topic, we may only want to view the subset of D..M which contain that topic in their ancestry path. So, using `--ancestry-path=H D..M` for example would result in:

```

E
 \
  C---G---H---I---J
           \
            L--M

```

Whereas `--ancestry-path=K D..M` would result in

```

K-----L--M

```

Before discussing another option, `--show-pulls`, we need to create a new example history. A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file's simplified history. Let's demonstrate a new example and show how options such as `--full-history` and `--simplify-merges` works in that case:

```

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

```

For this example, suppose I created file.txt which was modified by A, B, and X in different ways. The single-parent commits C, Z, and Y do not change file.txt. The merge commit M was created by resolving the merge conflict to include both changes from A and B and hence is not TREESAME to either. The merge commit R, however, was created by ignoring the contents of

file.txt at M and taking only the contents of file.txt at X. Hence, R is TREESAME to X but not M. Finally, the natural merge resolution to create N is to take the contents of file.txt at R, so N is TREESAME to R but not C. The merge commits O and P are TREESAME to their first parents, but not to their second parents, Z and Y respectively.

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

```
I---X
```

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

```

.-A---M-----N---O---P
/   /\  \ V  /  /
I   B  \ R-'^--' /
\  /   V      /
\ /    ^      /
`---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: many 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`.

OUTPUT

By default, or with `--ref-action=update`, this command produces no output on success, as refs are updated directly using an atomic transaction.

When using `--ref-action=print`, the output is usable as input to `git update-ref --stdin`. It is of the form:

```
update refs/heads/branch1 ${NEW_branch1_HASH} ${OLD_branch1_HASH}
```

```
update refs/heads/branch2 ${NEW_branch2_HASH} ${OLD_branch2_HASH}
```

```
update refs/heads/branch3 ${NEW_branch3_HASH} ${OLD_branch3_HASH}
```

where the number of refs updated depends on the arguments passed and the shape of the history being replayed. When using `--advance`, the number of refs updated is always one, but for `--onto`, it can be one or more (rebasing multiple branches simultaneously is supported).

There is no stderr output on conflicts; see the EXIT STATUS section below.

EXIT STATUS

For a successful, non-conflicted replay, the exit status is 0. When the replay has conflicts, the exit status is 1. If the replay is not able to complete (or start) due to some kind of error, the exit status is something other than 0 or 1.

EXAMPLES

To simply rebase mybranch onto target:

```
$ git replay --onto target origin/main..mybranch
```

The refs are updated atomically and no output is produced on success.

To see what would be updated without actually updating:

```
$ git replay --ref-action=print --onto target origin/main..mybranch
```

```
update refs/heads/mybranch ${NEW_mybranch_HASH} ${OLD_mybranch_HASH}
```

To cherry-pick the commits from mybranch onto target:

```
$ git replay --advance target origin/main..mybranch
```

Note that the first two examples replay the exact same commits and on top of the exact same new base, they only differ in that the first updates mybranch to point at the new commits and the second updates target to point at them.

What if you have a stack of branches, one depending upon another, and you'd really like to rebase the whole set?

```
$ git replay --contained --onto origin/main origin/main..tipbranch
```

All three branches (branch1, branch2, and tipbranch) are updated atomically.

When calling git replay, one does not need to specify a range of commits to replay using the syntax A..B; any range expression will do:

```
$ git replay --onto origin/main ^base branch1 branch2 branch3
```

This will simultaneously rebase branch1, branch2, and branch3, all commits they have since base, playing them on top of origin/main. These three branches may have commits on top of base that they have in common, but that does not need to be the case.

GIT

Part of the git(1) suite

Git 2.53.0

03/02/2026

GIT-REPLAY(1)