git-rebase(1) Manual Page

NAME

git-rebase - Reapply commits on top of another base tip

SYNOPSIS

git rebase [-i | --interactive] [<options>] [--exec <cmd>]
        [--onto <newbase> | --keep-base] [<upstream> [<branch>]]
git rebase [-i | --interactive] [<options>] [--exec <cmd>] [--onto <newbase>]
        --root [<branch>]
git rebase (--continue | --skip | --abort | --quit | --edit-todo | --show-current-patch)

DESCRIPTION

If <branch> is specified, *git rebase* will perform an automatic `git switch <branch>` before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote and branch.<name>.merge options will be used (see git-config(1) for details) and the `--fork-point` option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the rebase will abort.

All changes made by commits in the current branch but that are not in <upstream> are saved to a temporary area. This is the same set of commits that would be shown by `git log <upstream>..HEAD`; or by `git log 'fork_point'..HEAD`, if `--fork-point` is active (see the description on `--fork-point` below); or by `git log HEAD`, if the `--root` option is specified.

The current branch is reset to <upstream>, or <newbase> if the --onto option was supplied. This has the exact same effect as `git reset --hard <upstream>` (or <newbase>). ORIG_HEAD is set to point at the tip of the branch before the reset.

The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order. Note that any commits in HEAD which introduce the same textual changes as a commit in HEAD..<upstream> are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run `git rebase --continue`. Another option is to bypass the commit that caused the merge failure with `git rebase --skip`. To check out the original <branch> and remove the .git/rebase-apply working files, use the command `git rebase --abort` instead.

Assume the following history exists and the current branch is "topic":

```
          A---B---C topic
         /
    D---E---F---G master
```

From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
                  A'--B'--C' topic
                 /
    D---E---F---G master
```

NOTE: The latter form is just a short-hand of `git checkout topic` followed by `git rebase master`. When rebase exits `topic` will remain the checked-out branch.

If the upstream branch already contains a change you have made (e.g., because you mailed a patch which was applied upstream), then that commit will be skipped. For example, running `git rebase master` on the following history (in which `A'` and `A` introduce the same set of changes, but have different committer information):

```
          A---B---C topic
         /
    D---E---A'---F master
```

will result in:

```
                   B'---C' topic
                  /
    D---E---A'---F master
```

Here is how you would transplant a topic branch based on one branch to another, to pretend that you forked the topic branch from the latter branch, using `rebase --onto`.

First let’s assume your *topic* is based on branch *next*. For example, a feature developed in *topic* depends on some functionality which is found in *next*.

```
    o---o---o---o---o  master
         \
          o---o---o---o---o  next
                           \
                            o---o---o  topic
```

We want to make *topic* forked from branch *master*; for example, because the functionality on which *topic* depends was merged into the more stable *master* branch. We want our tree to look like this:

```
    o---o---o---o---o  master
        |            \
        |             o'--o'--o'  topic
         \
          o---o---o---o---o  next
```

We can get this using the following command:

```
git rebase --onto master next topic
```

Another example of --onto option is to rebase part of a branch. If we have the following situation:

```
                            H---I---J topicB
                           /
                  E---F---G  topicA
                 /
    A---B---C---D  master
```

then the command

```
git rebase --onto master topicA topicB
```

would result in:

```
                 H'--I'--J'  topicB
                /
                | E---F---G  topicA
                |/
    A---B---C---D  master
```

This is useful when topicB does not depend on topicA.

A range of commits could also be removed with rebase. If we have the following situation:

```
    E---F---G---H---I---J  topicA
```

then the command

```
git rebase --onto topicA~5 topicA~3 topicA
```

would result in the removal of commits F and G:

```
    E---H'---I'---J'  topicA
```

This is useful if F and G were flawed in some way, or should not be part of topicA. Note that the argument to --onto and the <upstream> parameter can be any valid commit-ish.

In case of conflict, *git rebase* will stop at the first problematic commit and leave conflict markers in the tree. You can use *git diff* to locate the markers (<<<<<<) and make edits to resolve the conflict. For each file you edit, you need to tell Git that the conflict has been resolved, typically this would be done with

```
git add <filename>
```

After resolving the conflict manually and updating the index with the desired resolution, you can continue the rebasing process with

```
git rebase --continue
```

Alternatively, you can undo the *git rebase* with

```
git rebase --abort
```

CONFIGURATION

rebase.useBuiltin

Unused configuration variable. Used in Git versions 2.20 and 2.21 as an escape hatch to enable the legacy shellscript implementation of rebase. Now the built-in rewrite of it in C is always used. Setting this will emit a warning, to alert any remaining users that setting this now does nothing.

rebase.backend

Default backend to use for rebasing. Possible choices are *apply* or *merge*. In the future, if the merge backend gains all remaining capabilities of the apply backend, this setting may become unused.

rebase.stat

Whether to show a diffstat of what changed upstream since the last rebase. False by default.

rebase.autoSquash

If set to true enable `--autosquash` option by default.

rebase.autoStash

When set to true, automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts. This option can be overridden by the `--no-autostash` and `--autostash` options of git-rebase(1). Defaults to false.

rebase.missingCommitsCheck

If set to "warn", git rebase -i will print a warning if some commits are removed (e.g. a line was deleted), however the rebase will still proceed. If set to "error", it will print the previous warning and stop the rebase, *git rebase --edit-todo* can then be used to correct the error. If set to "ignore", no checking is done. To drop a commit without warning or error, use the `drop` command in the todo list. Defaults to "ignore".

rebase.instructionFormat

A format string, as specified in git-log(1), to be used for the todo list during an interactive rebase. The format will automatically have the long commit hash prepended to the format.

rebase.abbreviateCommands

If set to true, `git rebase` will use abbreviated command names in the todo list resulting in something like this:

```
p deadbee The oneline of the commit
p fa1afe1 The oneline of the next commit
...
```

instead of:

```
pick deadbee The oneline of the commit
pick fa1afe1 The oneline of the next commit
...
```

Defaults to false.

rebase.rescheduleFailedExec

Automatically reschedule `exec` commands that failed. This only makes sense in interactive mode (or when an `--exec` option was provided). This is the same as specifying the `--reschedule-failed-exec` option.

OPTIONS

--onto <newbase>

Starting point at which to create the new commits. If the --onto option is not specified, the starting point is <upstream>. May be any valid commit, and not just an existing branch name.

As a special case, you may use "A...B" as a shortcut for the merge base of A and B if there is exactly one merge base. You can leave out at most one of A and B, in which case it defaults to HEAD.

--keep-base

Set the starting point at which to create the new commits to the merge base of <upstream> <branch>. Running *git rebase --keep-base <upstream> <branch>* is equivalent to running *git rebase --onto <upstream>… <upstream>*.

This option is useful in the case where one is developing a feature on top of an upstream branch. While the feature is being worked on, the upstream branch may advance and it may not be the best idea to keep rebasing on top of the upstream but to keep the base commit as-is.

Although both this option and --fork-point find the merge base between <upstream> and <branch>, this option uses the merge base as the *starting point* on which new commits will be created, whereas --fork-point uses the merge base to determine the *set of commits* which will be rebased.

See also INCOMPATIBLE OPTIONS below.

<upstream>

Upstream branch to compare against. May be any valid commit, not just an existing branch name. Defaults to the configured upstream for the current branch.

<branch>

Working branch; defaults to HEAD.

--continue

Restart the rebasing process after having resolved a merge conflict.

--abort

Abort the rebase operation and reset HEAD to the original branch. If <branch> was provided when the rebase operation was started, then HEAD will be reset to <branch>. Otherwise HEAD will be reset to where it was when the rebase operation was started.

--quit

Abort the rebase operation but HEAD is not reset back to the original branch. The index and working tree are also left unchanged as a result. If a temporary stash entry was created using --autostash, it will be saved to the stash list.

--apply

Use applying strategies to rebase (calling `git-am` internally). This option may become a no-op in the future once the merge backend handles everything the apply one does.

See also INCOMPATIBLE OPTIONS below.

--empty={drop,keep,ask}

How to handle commits that are not empty to start and are not clean cherry-picks of any upstream commit, but which become empty after rebasing (because they contain a subset of already upstream changes). With drop (the default), commits that become empty are dropped. With keep, such commits are kept. With ask (implied by --interactive), the rebase will halt when an empty commit is applied allowing you to choose whether to drop it, edit files more, or just commit the empty changes. Other options, like --exec, will use the default of drop unless -i/--interactive is explicitly specified.

Note that commits which start empty are kept (unless --no-keep-empty is specified), and commits which are clean cherry-picks (as determined by `git log --cherry-mark ...`) are detected and dropped as a preliminary step (unless --reapply-cherry-picks is passed).

See also INCOMPATIBLE OPTIONS below.

--no-keep-empty

--keep-empty

Do not keep commits that start empty before the rebase (i.e. that do not change anything from its parent) in the result. The default is to keep commits which start empty, since creating such commits requires passing the --allow-empty override flag to `git commit`, signifying that a user is very intentionally creating such a commit and thus wants to keep it.

Usage of this flag will probably be rare, since you can get rid of commits that start empty by just firing up an interactive rebase and removing the lines corresponding to the commits you don’t want. This flag exists as a convenient shortcut, such as for cases where external tools generate many empty commits and you want them all removed.

For commits which do not start empty but become empty after rebasing, see the --empty flag.

See also INCOMPATIBLE OPTIONS below.

--reapply-cherry-picks

--no-reapply-cherry-picks

Reapply all clean cherry-picks of any upstream commit instead of preemptively dropping them. (If these commits then become empty after rebasing, because they contain a subset of already upstream changes, the behavior towards them is controlled by the `--empty` flag.)

By default (or if `--no-reapply-cherry-picks` is given), these commits will be automatically dropped. Because this necessitates reading all upstream commits, this can be expensive in repos with a large number of upstream commits that need to be read.

`--reapply-cherry-picks` allows rebase to forgo reading all upstream commits, potentially improving performance.

See also INCOMPATIBLE OPTIONS below.

--allow-empty-message

No-op. Rebasing commits with an empty message used to fail and this option would override that behavior, allowing commits with empty messages to be rebased. Now commits with an empty message do not cause rebasing to halt.

See also INCOMPATIBLE OPTIONS below.

--skip

Restart the rebasing process by skipping the current patch.

--edit-todo

Edit the todo list during an interactive rebase.

--show-current-patch

Show the current patch in an interactive rebase or when rebase is stopped because of conflicts. This is the equivalent of `git show REBASE_HEAD`.

-m

--merge

Use merging strategies to rebase. When the recursive (default) merge strategy is used, this allows rebase to be aware of renames on the upstream side. This is the default.

Note that a rebase merge works by replaying each commit from the working branch on top of the <upstream> branch. Because of this, when a merge conflict happens, the side reported as *ours* is the so-far rebased series, starting with <upstream>, and *theirs* is the working branch. In other words, the sides are swapped.

See also INCOMPATIBLE OPTIONS below.

-s <strategy>

--strategy=<strategy>

Use the given merge strategy. If there is no `-s` option *git merge-recursive* is used instead. This implies --merge.

Because *git rebase* replays each commit from the working branch on top of the <upstream> branch using the given strategy, using the *ours* strategy simply empties all patches from the <branch>, which makes little sense.

See also INCOMPATIBLE OPTIONS below.

-X <strategy-option>

--strategy-option=<strategy-option>

Pass the <strategy-option> through to the merge strategy. This implies `--merge` and, if no strategy has been specified, `-s recursive`. Note the reversal of *ours* and *theirs* as noted above for the `-m` option.

See also INCOMPATIBLE OPTIONS below.

--rerere-autoupdate

--no-rerere-autoupdate

Allow the rerere mechanism to update the index with the result of auto-conflict resolution if possible.

-S[<keyid>]

--gpg-sign[=<keyid>]

--no-gpg-sign

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. `--no-gpg-sign` is useful to countermand both `commit.gpgSign` configuration variable, and earlier `--gpg-sign`.

-q

--quiet

Be quiet. Implies --no-stat.

-v

--verbose

Be verbose. Implies --stat.

--stat

Show a diffstat of what changed upstream since the last rebase. The diffstat is also controlled by the configuration option rebase.stat.

-n

--no-stat

Do not show a diffstat as part of the rebase process.

--no-verify

This option bypasses the pre-rebase hook. See also githooks(5).

--verify

Allows the pre-rebase hook to run, which is the default. This option can be used to override --no-verify. See also githooks(5).

-C<n>

Ensure at least <n> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored. Implies --apply.

See also INCOMPATIBLE OPTIONS below.

--no-ff

--force-rebase

-f

Individually replay all rebased commits instead of fast-forwarding over the unchanged ones. This ensures that the entire history of the rebased branch is composed of new commits.

You may find this helpful after reverting a topic branch merge, as this option recreates the topic branch with fresh commits so it can be remerged successfully without needing to "revert the reversion" (see the revert-a-faulty-merge How-To for details).

--fork-point

--no-fork-point

Use reflog to find a better common ancestor between <upstream> and <branch> when calculating which commits have been introduced by <branch>.

When --fork-point is active, *fork_point* will be used instead of <upstream> to calculate the set of commits to rebase, where *fork_point* is the result of `git merge-base --fork-point <upstream> <branch>` command (see git-merge-base(1)). If *fork_point* ends up being empty, the <upstream> will be used as a fallback.

If <upstream> is given on the command line, then the default is `--no-fork-point`, otherwise the default is `--fork-point`.

If your branch was based on <upstream> but <upstream> was rewound and your branch contains commits which were dropped, this option can be used with `--keep-base` in order to drop those commits from your branch.

See also INCOMPATIBLE OPTIONS below.

--ignore-whitespace

--whitespace=<option>

These flags are passed to the *git apply* program (see git-apply(1)) that applies the patch. Implies --apply.

See also INCOMPATIBLE OPTIONS below.

--committer-date-is-author-date

--ignore-date

These flags are passed to *git am* to easily change the dates of the rebased commits (see git-am(1)).

See also INCOMPATIBLE OPTIONS below.

--signoff

Add a Signed-off-by: trailer to all the rebased commits. Note that if `--interactive` is given then only commits marked to be picked, edited or reworded will have the trailer added.

See also INCOMPATIBLE OPTIONS below.

-i

--interactive

Make a list of the commits which are about to be rebased. Let the user edit that list before rebasing. This mode can also be used to split commits (see SPLITTING COMMITS below).

The commit list format can be changed by setting the configuration option rebase.instructionFormat. A customized instruction format will automatically have the long commit hash prepended to the format.

See also INCOMPATIBLE OPTIONS below.

-r

--rebase-merges[=(rebase-cousins|no-rebase-cousins)]

By default, a rebase will simply drop merge commits from the todo list, and put the rebased commits into a single, linear branch. With `--rebase-merges` , the rebase will instead try to preserve the branching structure within the commits that are to be rebased, by recreating the merge commits. Any resolved merge conflicts or manual amendments in these merge commits will have to be resolved/re-applied manually.

By default, or when `no-rebase-cousins` was specified, commits which do not have `<upstream>` as direct ancestor will keep their original branch point, i.e. commits that would be excluded by git-log(1)'s `--ancestry-path` option will keep their original ancestry by default. If the `rebase-cousins` mode is turned on, such commits are instead rebased onto `<upstream>` (or `<onto>` , if specified).

The `--rebase-merges` mode is similar in spirit to the deprecated `--preserve-merges` but works with interactive rebases, where commits can be reordered, inserted and dropped at will.

It is currently only possible to recreate the merge commits using the `recursive` merge strategy; Different merge strategies can be used only via explicit `exec git merge -s <strategy> [...]` commands.

See also REBASING MERGES and INCOMPATIBLE OPTIONS below.

-p

--preserve-merges

[DEPRECATED: use `--rebase-merges` instead] Recreate merge commits instead of flattening the history by replaying commits a merge commit introduces. Merge conflict resolutions or manual amendments to merge commits are not preserved.

This uses the `--interactive` machinery internally, but combining it with the `--interactive` option explicitly is generally not a good idea unless you know what you are doing (see BUGS below).

See also INCOMPATIBLE OPTIONS below.

-x <cmd>

--exec <cmd>

Append "exec <cmd>" after each line creating a commit in the final history. <cmd> will be interpreted as one or more shell commands. Any command that fails will interrupt the rebase, with exit code 1.

You may execute several commands by either using one instance of `--exec` with several commands:

```
git rebase -i --exec "cmd1 && cmd2 && ..."
```

or by giving more than one `--exec` :

```
git rebase -i --exec "cmd1" --exec "cmd2" --exec ...
```

If `--autosquash` is used, "exec" lines will not be appended for the intermediate commits, and will only appear at the end of each squash/fixup series.

This uses the `--interactive` machinery internally, but it can be run without an explicit `--interactive`.

See also INCOMPATIBLE OPTIONS below.

--root

Rebase all commits reachable from <branch>, instead of limiting them with an <upstream>. This allows you to rebase the root commit(s) on a branch. When used with --onto, it will skip changes already contained in <newbase> (instead of <upstream>) whereas without --onto it will operate on every change. When used together with both --onto and --preserve-merges, *all* root commits will be rewritten to have <newbase> as parent instead.

See also INCOMPATIBLE OPTIONS below.

--autosquash

--no-autosquash

When the commit log message begins with "squash! …" (or "fixup! …"), and there is already a commit in the todo list that matches the same `...`, automatically modify the todo list of rebase -i so that the commit marked for squashing comes right after the commit to be modified, and change the action of the moved commit from `pick` to `squash` (or `fixup`). A commit matches the `...` if the commit subject matches, or if the `...` refers to the commit's hash. As a fall-back, partial matches of the commit subject work, too. The recommended way to create fixup/squash commits is by using the `--fixup`/`--squash` options of git-commit(1).

If the `--autosquash` option is enabled by default using the configuration variable `rebase.autoSquash`, this option can be used to override and disable this setting.

See also INCOMPATIBLE OPTIONS below.

--autostash

--no-autostash

Automatically create a temporary stash entry before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts.

--reschedule-failed-exec

--no-reschedule-failed-exec

Automatically reschedule `exec` commands that failed. This only makes sense in interactive mode (or when an `--exec` option was provided).

INCOMPATIBLE OPTIONS

The following options:

- --apply
- --committer-date-is-author-date
- --ignore-date
- --ignore-whitespace
- --whitespace

- -C

are incompatible with the following options:

- --merge
- --strategy
- --strategy-option
- --allow-empty-message
- --[no-]autosquash
- --rebase-merges
- --preserve-merges
- --interactive
- --exec
- --no-keep-empty
- --empty=
- --reapply-cherry-picks
- --edit-todo
- --root when used in combination with --onto

In addition, the following pairs of options are incompatible:

- --preserve-merges and --interactive
- --preserve-merges and --signoff
- --preserve-merges and --rebase-merges
- --preserve-merges and --empty=
- --keep-base and --onto
- --keep-base and --root
- --fork-point and --root

BEHAVIORAL DIFFERENCES

git rebase has two primary backends: apply and merge. (The apply backend used to be known as the *am* backend, but the name led to confusion as it looks like a verb instead of a noun. Also, the merge backend used to be known as the interactive backend, but it is now used for non-interactive cases as well. Both were renamed based on lower-level functionality that underpinned each.) There are some subtle differences in how these two backends behave:

Empty commits

The apply backend unfortunately drops intentionally empty commits, i.e. commits that started empty, though these are rare in practice. It also drops commits that become empty and has no option for controlling this behavior.

The merge backend keeps intentionally empty commits by default (though with -i they are marked as empty in the todo list editor, or they can be dropped automatically with --no-keep-empty).

Similar to the apply backend, by default the merge backend drops commits that become empty unless -i/--interactive is specified (in which case it stops and asks the user what to do). The merge backend also has an --empty={drop,keep,ask} option for changing the behavior of handling commits that become empty.

Directory rename detection

Due to the lack of accurate tree information (arising from constructing fake ancestors with the limited information available in patches), directory rename detection is disabled in the apply backend. Disabled directory rename detection means that if one side of history renames a directory and the other adds new files to the old directory, then the new files will be left behind in the old directory without any warning at the time of rebasing that you may want to move these files into the new directory.

Directory rename detection works with the merge backend to provide you warnings in such cases.

Context

The apply backend works by creating a sequence of patches (by calling `format-patch` internally), and then applying the patches in sequence (calling `am` internally). Patches are composed of multiple hunks, each with line numbers, a context region, and the actual changes. The line numbers have to be taken with some fuzz, since the other side will likely have inserted or deleted lines earlier in the file. The context region is meant to help find how to adjust the line numbers in order to apply the changes to the right lines. However, if multiple areas of the code have the same surrounding lines of context, the wrong one can be picked. There are real-world cases where this has caused commits to be reapplied incorrectly with no conflicts reported. Setting diff.context to a larger value may prevent such types of problems, but increases the chance of spurious conflicts (since it will require more lines of matching context to apply).

The merge backend works with a full copy of each relevant file, insulating it from these types of problems.

Labelling of conflicts markers

When there are content conflicts, the merge machinery tries to annotate each side's conflict markers with the commits where the content came from. Since the apply backend drops the original information about the rebased commits and their parents (and instead generates new fake commits based off limited information in the generated patches), those commits cannot be identified; instead it has to fall back to a commit summary. Also, when merge.conflictStyle is set to diff3, the apply backend will use "constructed merge base" to label the content from the merge base, and thus provide no information about the merge base commit whatsoever.

The merge backend works with the full commits on both sides of history and thus has no such limitations.

Hooks

The apply backend has not traditionally called the post-commit hook, while the merge backend has. Both have called the post-checkout hook, though the merge backend has squelched its output. Further, both backends only call the post-checkout hook with the starting point commit of the rebase, not the intermediate commits nor the final commit. In each case, the calling of these hooks was by accident of implementation rather than by design (both backends were originally implemented as shell scripts and happened to invoke other commands like *git checkout* or *git commit* that would call the hooks). Both backends should have the same behavior, though it is not entirely clear which, if any, is correct. We will likely make rebase stop calling either of these hooks in the future.

Interruptability

The apply backend has safety problems with an ill-timed interrupt; if the user presses Ctrl-C at the wrong time to try to abort the rebase, the rebase can enter a state where it cannot be aborted with a subsequent `git rebase --abort`. The merge backend does not appear to suffer from the same shortcoming. (See https://lore.kernel.org/git/20200207132152.GC2868@szeder.dev/ for details.)

Commit Rewording

When a conflict occurs while rebasing, rebase stops and asks the user to resolve. Since the user may need to make notable changes while resolving conflicts, after conflicts are resolved and the user has run `git rebase --continue`, the rebase should open an editor and ask the user to update the commit message. The merge backend does this, while the apply backend blindly applies the original commit message.

Miscellaneous differences

There are a few more behavioral differences that most folks would probably consider inconsequential but which are mentioned for completeness:

- Reflog: The two backends will use different wording when describing the changes made in the reflog, though both will make use of the word "rebase".
- Progress, informational, and error messages: The two backends provide slightly different progress and informational messages. Also, the apply backend writes error messages (such as "Your files would be overwritten…") to stdout, while the merge backend writes them to stderr.
- State directories: The two backends keep their state in different directories under .git/

MERGE STRATEGIES

The merge mechanism (`git merge` and `git pull` commands) allows the backend *merge strategies* to be chosen with `-s` option. Some strategies can also take their own options, which can be passed by giving `-X<option>` arguments to `git merge` and/or `git pull`.

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities and is considered generally safe and fast.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames, but currently cannot make use of detected copies. This is the default merge strategy when pulling or merging one branch.

The *recursive* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected in the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*; note that, unlike *ours*, there is no *theirs* merge strategy to confuse this merge option with.

patience

With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff(1) `--patience`.

diff-algorithm=[patience|minimal|histogram|myers]

Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff(1) `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-at-eol

ignore-cr-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also git-diff(1) `-b`, `-w`, `--ignore-space-at-eol`, and `--ignore-cr-at-eol`.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in gitattributes(5) for details.

no-renormalize

Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

no-renames

Turn off rename detection. This overrides the `merge.renames` configuration variable. See also git-diff(1) `--no-renames`.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. This overrides the *merge.renames* configuration variable. See also git-diff(1) `--find-renames`.

rename-threshold=<n>

Deprecated synonym for `find-renames=<n>`.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified recursive strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *recursive*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

NOTES

You should understand the implications of using *git rebase* on a repository that you share. See also RECOVERING FROM UPSTREAM REBASE below.

When the git-rebase command is run, it will first execute a "pre-rebase" hook if one exists. You can use this hook to do sanity checks and reject the rebase if it isn't appropriate. Please see the template pre-rebase hook script for an example.

Upon completion, <branch> will be the current branch.

INTERACTIVE MODE

Rebasing interactively means that you have a chance to edit the commits which are rebased. You can reorder the commits, and you can remove them (weeding out bad or otherwise unwanted patches).

The interactive mode is meant for this type of workflow:

1. have a wonderful idea
2. hack on the code

3. prepare a series for submission
4. submit

where point 2. consists of several instances of

a) regular use

1. finish something worthy of a commit
2. commit

b) independent fixup

1. realize that something does not work
2. fix that
3. commit it

Sometimes the thing fixed in b.2. cannot be amended to the not-quite perfect commit it fixes, because that commit is buried deeply in a patch series. That is exactly what interactive rebase is for: use it after plenty of "a"s and "b"s, by rearranging and editing commits, and squashing multiple commits into one.

Start it with the last commit you want to retain as-is:

```
git rebase -i <after-this-commit>
```

An editor will be fired up with all the commits in your current branch (ignoring merge commits), which come after the given commit. You can reorder the commits in this list to your heart’s content, and you can remove them. The list looks more or less like this:

```
pick deadbee The oneline of this commit
pick fa1afe1 The oneline of the next commit
...
```

The oneline descriptions are purely for your pleasure; *git rebase* will not look at them but at the commit names ("deadbee" and "fa1afe1" in this example), so do not delete or edit the names.

By replacing the command "pick" with the command "edit", you can tell *git rebase* to stop after applying that commit, so that you can edit the files and/or the commit message, amend the commit, and continue rebasing.

To interrupt the rebase (just like an "edit" command would do, but without cherry-picking any commit first), use the "break" command.

If you just want to edit the commit message for a commit, replace the command "pick" with the command "reword".

To drop a commit, replace the command "pick" with "drop", or just delete the matching line.

If you want to fold two or more commits into one, replace the command "pick" for the second and subsequent commits with "squash" or "fixup". If the commits had different authors, the folded commit will be attributed to the author of the first commit. The suggested commit message for the folded commit is the concatenation of the commit messages of the first commit and of those with the "squash" command, but omits the commit messages of commits with the "fixup" command.

git rebase will stop when "pick" has been replaced with "edit" or when a command fails due to merge errors. When you are done editing and/or resolving conflicts you can continue with `git rebase --continue`.

For example, if you want to reorder the last 5 commits, such that what was HEAD~4 becomes the new HEAD. To achieve that, you would call *git rebase* like this:

```
$ git rebase -i HEAD~5
```

And move the first patch to the end of the list.

You might want to recreate merge commits, e.g. if you have a history like this:

```
           X
            \
         A---M---B
        /
---o---O---P---Q
```

Suppose you want to rebase the side branch starting at "A" to "Q". Make sure that the current HEAD is "B", and call

```
$ git rebase -i -r --onto Q O
```

Reordering and editing commits usually creates untested intermediate steps. You may want to check that your history editing did not break anything by running a test, or at least recompiling at intermediate points in history by using the "exec" command (shortcut "x"). You may do so by creating a todo list like this one:

```
pick deadbee Implement feature XXX
fixup f1a5c00 Fix to feature XXX
exec make
pick c0ffeee The oneline of the next commit
edit deadbab The oneline of the commit after
exec cd subdir; make test
...
```

The interactive rebase will stop when a command fails (i.e. exits with non-0 status) to give you an opportunity to fix the problem. You can continue with `git rebase --continue`.

The "exec" command launches the command in a shell (the one specified in `$SHELL`, or the default shell if `$SHELL` is not set), so you can use shell features (like "cd", ">", ";" …). The command is run from the root of the working tree.

```
$ git rebase -i --exec "make test"
```

This command lets you check that intermediate commits are compilable. The todo list becomes like that:

```
pick 5928aea one
exec make test
pick 04d0fda two
exec make test
pick ba46169 three
exec make test
pick f4593f9 four
exec make test
```

SPLITTING COMMITS

In interactive mode, you can mark commits with the action "edit". However, this does not necessarily mean that *git rebase* expects the result of this edit to be exactly one commit. Indeed, you can undo the commit, or you can add other commits. This can be used to split a commit into two:

- Start an interactive rebase with `git rebase -i <commit>^`, where <commit> is the commit you want to split. In fact, any commit range will do, as long as it contains that commit.
- Mark the commit you want to split with the action "edit".
- When it comes to editing that commit, execute `git reset HEAD^`. The effect is that the HEAD is rewound by one, and the index follows suit. However, the working tree stays the same.
- Now add the changes to the index that you want to have in the first commit. You can use `git add` (possibly interactively) or *git gui* (or both) to do that.
- Commit the now-current index with whatever commit message is appropriate now.
- Repeat the last two steps until your working tree is clean.
- Continue the rebase with `git rebase --continue`.

If you are not absolutely sure that the intermediate revisions are consistent (they compile, pass the testsuite, etc.) you should use *git stash* to stash away the not-yet-committed changes after each commit, test, and amend the commit if fixes are necessary.

RECOVERING FROM UPSTREAM REBASE

Rebasing (or any other form of rewriting) a branch that others have based work on is a bad idea: anyone downstream of it is forced to manually fix their history. This section explains how to do the fix from the downstream's point of view. The real fix, however, would be to avoid rebasing the upstream in the first place.

To illustrate, suppose you are in a situation where someone develops a *subsystem* branch, and you are working on a *topic* that is dependent on this *subsystem*. You might end up with a history like the following:

```
    o---o---o---o---o---o---o---o  master
         \
          o---o---o---o---o  subsystem
                           \
                            *---*---*  topic
```

If *subsystem* is rebased against *master*, the following happens:

```
o---o---o---o---o---o---o---o  master
     \                       \
      o---o---o---o---o       o'--o'--o'--o'--o'  subsystem
                       \
                        *---*---*  topic
```

If you now continue development as usual, and eventually merge *topic* to *subsystem*, the commits from *subsystem* will remain duplicated forever:

```
o---o---o---o---o---o---o---o  master
     \                       \
      o---o---o---o---o       o'--o'--o'--o'--o'--M  subsystem
                       \                         /
                        *---*---*-..........-*--*  topic
```

Such duplicates are generally frowned upon because they clutter up history, making it harder to follow. To clean things up, you need to transplant the commits on *topic* to the new *subsystem* tip, i.e., rebase *topic*. This becomes a ripple effect: anyone downstream from *topic* is forced to rebase too, and so on!

There are two kinds of fixes, discussed in the following subsections:

Easy case: The changes are literally the same.

This happens if the *subsystem* rebase was a simple rebase and had no conflicts.

Hard case: The changes are not the same.

This happens if the *subsystem* rebase had conflicts, or used `--interactive` to omit, edit, squash, or fixup commits; or if the upstream used one of `commit --amend`, `reset`, or a full history rewriting command like `filter-repo` (https://github.com/newren/git-filter-repo).

The easy case

Only works if the changes (patch IDs based on the diff contents) on *subsystem* are literally the same before and after the rebase *subsystem* did.

In that case, the fix is easy because *git rebase* knows to skip changes that are already present in the new upstream (unless `--reapply-cherry-picks` is given). So if you say (assuming you're on *topic*)

```
$ git rebase subsystem
```

you will end up with the fixed history

```
o---o---o---o---o---o---o---o  master
                             \
                              o'--o'--o'--o'--o'  subsystem
                                               \
                                                *---*---*  topic
```

The hard case

Things get more complicated if the *subsystem* changes do not exactly correspond to the ones before the rebase.

NOTE While an "easy case recovery" sometimes appears to be successful even in the hard case, it may have unintended consequences. For example, a commit that was removed via `git rebase --interactive` will be **resurrected**!

The idea is to manually tell *git rebase* "where the old *subsystem* ended and your *topic* began", that is, what the old merge base between them was. You will have to find a way to name the last commit of the old *subsystem*, for example:

- With the *subsystem* reflog: after *git fetch*, the old tip of *subsystem* is at `subsystem@{1}`. Subsequent fetches will increase the number. (See git-reflog(1).)
- Relative to the tip of *topic*: knowing that your *topic* has three commits, the old tip of *subsystem* must be `topic~3`.

You can then transplant the old `subsystem..topic` to the new tip by saying (for the reflog case, and assuming you are on *topic* already):

```
$ git rebase --onto subsystem subsystem@{1}
```

The ripple effect of a "hard case" recovery is especially bad: *everyone* downstream from *topic* will now have to perform a "hard case" recovery too!

REBASING MERGES

The interactive rebase command was originally designed to handle individual patch series. As such, it makes sense to exclude merge commits from the todo list, as the developer may have merged the then-current `master` while working on the branch, only to rebase all the commits onto `master` eventually (skipping the merge commits).

However, there are legitimate reasons why a developer may want to recreate merge commits: to keep the branch structure (or "commit topology") when working on multiple, inter-related branches.

In the following example, the developer works on a topic branch that refactors the way buttons are defined, and on another topic branch that uses that refactoring to implement a "Report a bug" button. The output of `git log --graph --format=%s -5` may look like this:

```
*   Merge branch 'report-a-bug'
|\
| * Add the feedback button
* | Merge branch 'refactor-button'
|\ \
| |/
| * Use the Button class for all buttons
| * Extract a generic Button class from the DownloadButton one
```

The developer might want to rebase those commits to a newer `master` while keeping the branch topology, for example when the first topic branch is expected to be integrated into `master` much earlier than the second one, say, to resolve merge conflicts with changes to the DownloadButton class that made it into `master`.

This rebase can be performed using the `--rebase-merges` option. It will generate a todo list looking like this:

```
label onto

# Branch: refactor-button
reset onto
pick 123456 Extract a generic Button class from the DownloadButton one
pick 654321 Use the Button class for all buttons
label refactor-button

# Branch: report-a-bug
reset refactor-button # Use the Button class for all buttons
pick abcdef Add the feedback button
label report-a-bug

reset onto
merge -C a1b2c3 refactor-button # Merge 'refactor-button'
merge -C 6f5e4d report-a-bug # Merge 'report-a-bug'
```

In contrast to a regular interactive rebase, there are `label` , `reset` and `merge` commands in addition to `pick` ones.

The `label` command associates a label with the current HEAD when that command is executed. These labels are created as worktree-local refs (`refs/rewritten/<label>`) that will be deleted when the rebase finishes. That way, rebase operations in multiple worktrees linked to the same repository do not interfere with one another. If the `label` command fails, it is rescheduled immediately, with a helpful message how to proceed.

The `reset` command resets the HEAD, index and worktree to the specified revision. It is similar to an `exec git reset --hard <label>` , but refuses to overwrite untracked files. If the `reset` command fails, it is rescheduled immediately, with a helpful message how to edit the todo list (this typically happens when a `reset` command was inserted into the todo list manually and contains a typo).

The `merge` command will merge the specified revision(s) into whatever is HEAD at that time. With `-C <original-commit>` , the commit message of the specified merge commit will be used. When the `-C` is changed to a lower-case `-c` , the message will be opened in an editor after a successful merge so that the user can edit the message.

If a `merge` command fails for any reason other than merge conflicts (i.e. when the merge operation did not even start), it is rescheduled immediately.

At this time, the `merge` command will **always** use the `recursive` merge strategy for regular merges, and `octopus` for octopus merges, with no way to choose a different one. To work around this, an `exec` command can be used to call `git merge` explicitly, using the fact that the labels are worktree-local refs (the ref `refs/rewritten/onto` would correspond to the label `onto` , for example).

Note: the first command (`label onto`) labels the revision onto which the commits are rebased; The name `onto` is just a convention, as a nod to the `--onto` option.

It is also possible to introduce completely new merge commits from scratch by adding a command of the form `merge <merge-head>` . This form will generate a tentative commit message and always open an editor to let the user edit it. This can be useful e.g. when a topic branch turns out to address more than a single concern and wants to be split into two or even more topic branches. Consider this todo list:

```
pick 192837 Switch from GNU Makefiles to CMake
pick 5a6c7e Document the switch to CMake
pick 918273 Fix detection of OpenSSL in CMake
pick afbecd http: add support for TLS v1.3
pick fdbaec Fix detection of cURL in CMake on Windows
```

The one commit in this list that is not related to CMake may very well have been motivated by working on fixing all those bugs introduced by switching to CMake, but it addresses a different concern. To split this branch into two topic branches, the todo list could be edited like this:

```
label onto

pick afbecd http: add support for TLS v1.3
label tlsv1.3

reset onto
pick 192837 Switch from GNU Makefiles to CMake
pick 918273 Fix detection of OpenSSL in CMake
pick fdbaec Fix detection of cURL in CMake on Windows
pick 5a6c7e Document the switch to CMake
label cmake

reset onto
merge tlsv1.3
merge cmake
```

BUGS

The todo list presented by the deprecated `--preserve-merges --interactive` does not represent the topology of the revision graph (use `--rebase-merges` instead). Editing commits and rewording their commit messages should work fine, but attempts to reorder commits tend to produce counterintuitive results. Use `--rebase-merges` in such scenarios instead.

For example, an attempt to rearrange

```
1 --- 2 --- 3 --- 4 --- 5
```

to

```
1 --- 2 --- 4 --- 3 --- 5
```

by moving the "pick 4" line will result in the following history:

```
        3
       /
1 --- 2 --- 4 --- 5
```

GIT

Part of the git(1) suite

Last updated 2020-07-28 08:59:54 UTC