NAME

git-rebase - Forward-port local commits to the updated upstream head

SYNOPSIS

git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo

DESCRIPTION

If <branch> is specified, *git rebase* will perform an automatic git checkout <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. 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 git log HEAD, if --root 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.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
: If set to true enable *--autostash* option by default.

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.

<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.

--keep-empty
: Keep the commits that do not change anything from its parents in the result.

--skip
: Restart the rebasing process by skipping the current patch.

--edit-todo
Edit the todo list during an interactive rebase.

-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.

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.

-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 discards all patches from the <branch>, which makes little sense.

-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.

-S[<keyid>], --gpg-sign[=<keyid>]
GPG-sign commits.

-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.

-f, --force-rebase
Force a rebase even if the current branch is up-to-date and the command without --force would return without doing anything.

You may find this (or --no-ff with an interactive rebase) 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**[1] for details).

--fork-point, --no-fork-point
Use *git merge-base --fork-point* to find a better common ancestor between upstream and

branch when calculating which commits have have been introduced by branch (see **git-merge-base(1)**).

If no non-option arguments are given on the command line, then the default is --fork-point @{u} otherwise the upstream argument is interpreted literally unless the --fork-point option is specified.

--ignore-whitespace, --whitespace=<option>
These flag are passed to the *git apply* program (see **git-apply(1)**) that applies the patch. Incompatible with the --interactive option.

--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)**). Incompatible with the --interactive option.

-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).

-p, --preserve-merges
Instead of ignoring merges, try to recreate them.

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).

-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.

This option can only be used with the --interactive option (see INTERACTIVE MODE below).

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.

--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.

--autosquash, --no-autosquash
When the commit log message begins with squash! ... (or fixup! ...), and there is a commit whose title begins with 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). Ignores subsequent fixup! or squash! after the first, in case you referred to an earlier fixup/squash with git commit --fixup/--squash.

This option is only valid when the *--interactive* option is used.

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.

--[no-]autostash
Automatically create a temporary stash 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.

--no-ff
With --interactive, cherry-pick 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.

Without --interactive, this is a synonym for --force-rebase.

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**[1] for details).

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. 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 to 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*.

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
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, and --ignore-space-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.

rename-threshold=<n>
: Controls the similarity threshold used for rename detection. See also **git-diff(1)**-M.

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 as and bs, 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.

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

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 preserve merges, 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 -p --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---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---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---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 filter-branch.

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. 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!

BUGS

The todo list presented by --preserve-merges --interactive does not represent the topology of the revision graph. Editing commits and rewording their commit messages should work fine, but attempts to reorder commits tend to produce counterintuitive results.

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

NOTES

1. revert-a-faulty-merge How-To
   file:///usr/share/doc/git/html/howto/revert-a-faulty-merge.html