NAME

git-cherry-pick - Apply the changes introduced by some existing commits

SYNOPSIS

git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff] [-S[<key-id>]] <commit>...
git cherry-pick--con tinue
git cherry-pick--quit
git cherry-pick--abort

DESCRIPTION

Given one or more existing commits, apply the change each one introduces, recording a new commit for each. This requires your working tree to be clean (no modifications from the HEAD commit).

When it is not obvious how to apply a change, the following happens:

1. The current branch and HEAD pointer stay at the last commit successfully made.
2. The CHERRY_PICK_HEAD ref is set to point at the commit that introduced the change that is difficult to apply.
3. Paths in which the change applied cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions, as described in the TRUE MERGE section of **git-merge(1)**. The working tree files will include a description of the conflict bracketed by the usual conflict markers <<<<<<< and >>>>>>>.
5. No other modifications are made.

See **git-merge(1)** for some hints on resolving such conflicts.

OPTIONS

<commit>...
: Commits to cherry-pick. For a more complete list of ways to spell commits, see **gitrevisions(7)**. Sets of commits can be passed but no traversal is done by default, as if the *--no-walk* option was specified, see **git-rev-list(1)**. Note that specifying a range will feed all <commit>... arguments to a single revision walk (see a later example that uses *maint master..next*).

-e, --edit
: With this option, *git cherry-pick* will let you edit the commit message prior to committing.

-x
: When recording the commit, append a line that says (cherry picked from commit ...) to the original commit message in order to indicate which commit this change was cherry-picked from. This is done only for cherry picks without conflicts. Do not use this option if you are cherry-picking from your private branch because the information is useless to the recipient. If on the other hand you are cherry-picking between two publicly visible branches (e.g. backporting a fix to a maintenance branch for an older release from a development branch), adding this information can be useful.

-r
: It used to be that the command defaulted to do -x described above, and -r was to disable it. Now the default is not to do -x so this option is a no-op.

-m parent-number, --mainline parent-number
: Usually you cannot cherry-pick a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows cherry-pick to replay the change relative to the specified parent.

-n, --no-commit
: Usually the command automatically creates a sequence of commits. This flag applies the

changes necessary to cherry-pick each named commit to your working tree and the index, without making any commit. In addition, when this option is used, your index does not have to match the HEAD commit. The cherry-pick is done against the beginning state of your index.

This is useful when cherry-picking more than one commits effect to your index in a row.

-s, --signoff
Add Signed-off-by line at the end of the commit message.

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

--ff
If the current HEAD is the same as the parent of the cherry-pick'ed commit, then a fast forward to this commit will be performed.

--allow-empty
By default, cherry-picking an empty commit will fail, indicating that an explicit invocation of git commit --allow-empty is required. This option overrides that behavior, allowing empty commits to be preserved automatically in a cherry-pick. Note that when --ff is in effect, empty commits that meet the fast-forward requirement will be kept even without this option. Note also, that use of this option only keeps commits that were initially empty (i.e. the commit recorded the same tree as its parent). Commits which are made empty due to a previous commit are dropped. To force the inclusion of those commits use --keep-redundant-commits.

--allow-empty-message
By default, cherry-picking a commit with an empty message will fail. This option overrides that behaviour, allowing commits with empty messages to be cherry picked.

--keep-redundant-commits
If a commit being cherry picked duplicates a commit already in the current history, it will become empty. By default these redundant commits are ignored. This option overrides that behavior and creates an empty commit object. Implies --allow-empty.

--strategy=<strategy>
Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in **git-merge(1)** for details.

-X<option>, --strategy-option=<option>
Pass the merge strategy-specific option through to the merge strategy. See **git-merge(1)** for details.

SEQUENCER SUBCOMMANDS

--continue
Continue the operation in progress using the information in *.git/sequencer*. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--quit
Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--abort
Cancel the operation and return to the pre-sequence state.

EXAMPLES

git cherry-pick master
Apply the change introduced by the commit at the tip of the master branch and create a new commit with this change.

git cherry-pick ..master, git cherry-pick ^HEAD master
Apply the changes introduced by all commits that are ancestors of master but not of HEAD

to produce new commits.

git cherry-pick maint next ^master, git cherry-pick maint master..next
: Apply the changes introduced by all commits that are ancestors of maint or next, but not master or any of its ancestors. Note that the latter does not mean maint and everything between master and next; specifically, maint will not be used if it is included in master.

git cherry-pick master~4 master~2
: Apply the changes introduced by the fifth and third last commits pointed to by master and create 2 new commits with these changes.

git cherry-pick -n master~1 next
: Apply to the working tree and the index the changes introduced by the second last commit pointed to by master and by the last commit pointed to by next, but do not create any commit with these changes.

git cherry-pick --ff ..next
: If history is linear and HEAD is an ancestor of next, update the working tree and advance the HEAD pointer to match next. Otherwise, apply the changes introduced by those commits that are in next but not HEAD to the current branch, creating a new commit for each new change.

git rev-list --reverse master -- README | git cherry-pick -n --stdin
: Apply the changes introduced by all commits on the master branch that touched README to the working tree and index, so the result can be inspected and made into a single new commit if suitable.

The following sequence attempts to backport a patch, bails out because the code the patch applies to has changed too much, and then tries again, this time exercising more care about matching up context lines.

$ git cherry-pick topic^ **(1)**
$ git diff **(2)**
$ git reset --merge ORIG_HEAD **(3)**
$ git cherry-pick -Xpatience topic^ **(4)**

1. apply the change that would be shown by git show topic^. In this example, the patch does not apply cleanly, so information about the conflict is written to the index and working tree and no new commit results.
2. summarize changes to be reconciled
3. cancel the cherry-pick. In other words, return to the pre-cherry-pick state, preserving any local modifications you had in the working tree.
4. try to apply the change introduced by topic^ again, spending extra time to avoid mistakes based on incorrectly matching context lines.

SEE ALSO

git-revert(1)

GIT

Part of the **git(1)** suite