git-reset(1) Manual Page

NAME

git-reset - Reset current HEAD to the specified state

SYNOPSIS

git reset [-q] [<tree-ish>] [--] <pathspec>...
git reset [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]
git reset (--patch | -p) [<tree-ish>] [--] [<pathspec>...]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
DEPRECATED: *git reset* [-q] [--stdin [-z]] [<tree-ish>]

DESCRIPTION

In the first three forms, copy entries from `<tree-ish>` to the index. In the last form, set the current branch head (`HEAD`) to `<commit>` , optionally modifying index and working tree to match. The `<tree-ish>` / `<commit>` defaults to `HEAD` in all forms.

***git reset* [-q] [<tree-ish>] [--] <pathspec>...**

***git reset* [-q] [--pathspec-from-file=<file> [--pathspec-file-nul]] [<tree-ish>]**

These forms reset the index entries for all paths that match the `<pathspec>` to their state at `<tree-ish>` . (It does not affect the working tree or the current branch.)

This means that `git reset <pathspec>` is the opposite of `git add <pathspec>` . This command is equivalent to `git restore [--source=<tree-ish>] --staged <pathspec>...` .

After running `git reset <pathspec>` to update the index entry, you can use git-restore(1) to check the contents out of the index to the working tree. Alternatively, using git-restore(1) and specifying a commit with `--source` , you can copy the contents of a path out of a commit to the index and to the working tree in one go.

***git reset* (--patch | -p) [<tree-ish>] [--] [<pathspec>...]**

Interactively select hunks in the difference between the index and `<tree-ish>` (defaults to `HEAD`). The chosen hunks are applied in reverse to the index.

This means that `git reset -p` is the opposite of `git add -p` , i.e. you can use it to selectively reset hunks. See the “Interactive Mode” section of git-add(1) to learn how to operate the `--patch` mode.

***git reset* [<mode>] [<commit>]**

This form resets the current branch head to `<commit>` and possibly updates the index (resetting it to the tree of `<commit>`) and the working tree depending on `<mode>` . If `<mode>` is omitted, defaults to `--mixed` . The `<mode>` must be one of the following:

--soft

Does not touch the index file or the working tree at all (but resets the head to `<commit>`, just like all modes do). This leaves all your changed files "Changes to be committed", as `git status` would put it.

--mixed

Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.

If `-N` is specified, removed paths are marked as intent-to-add (see git-add(1)).

--hard

Resets the index and working tree. Any changes to tracked files in the working tree since `<commit>` are discarded.

--merge

Resets the index and updates the files in the working tree that are different between `<commit>` and `HEAD`, but keeps those which are different between the index and working tree (i.e. which have changes which have not been added). If a file that is different between `<commit>` and the index has unstaged changes, reset is aborted.

In other words, `--merge` does something like a `git read-tree -u -m <commit>`, but carries forward unmerged index entries.

--keep

Resets index entries and updates files in the working tree that are different between `<commit>` and `HEAD`. If a file that is different between `<commit>` and `HEAD` has local changes, reset is aborted.

--[no-]recurse-submodules

When the working tree is updated, using --recurse-submodules will also recursively reset the working tree of all active submodules according to the commit recorded in the superproject, also setting the submodules' HEAD to be detached at that commit.

See "Reset, restore and revert" in git(1) for the differences between the three commands.

OPTIONS

-q

--quiet

--no-quiet

Be quiet, only report errors. The default behavior is set by the `reset.quiet` config option. `--quiet` and `--no-quiet` will override the default behavior.

--pathspec-from-file=<**file**>

Pathspec is passed in `<file>` instead of commandline args. If `<file>` is exactly `-` then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable `core.quotePath` (see git-config(1)). See also `--pathspec-file-nul` and global `--literal-pathspecs`.

--pathspec-file-nul

Only meaningful with `--pathspec-from-file`. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

--

Do not interpret any more arguments as options.

<pathspec>...

Limits the paths affected by the operation.

For more details, see the *pathspec* entry in gitglossary(7).

--stdin

DEPRECATED (use `--pathspec-from-file=-` instead): Instead of taking list of paths from the command line, read list of paths from the standard input. Paths are separated by LF (i.e. one path per line) by default.

-z

DEPRECATED (use `--pathspec-file-nul` instead): Only meaningful with `--stdin`; paths are separated with NUL character instead of LF.

EXAMPLES

Undo add

```
$ edit                                     (1)
$ git add frotz.c filfre.c
$ mailx                                    (2)
$ git reset                                (3)
$ git pull git://info.example.com/ nitfol  (4)
```

1. You are happily working on something, and find the changes in these files are in good order. You do not want to see them when you run `git diff`, because you plan to work on other files and changes with these files are distracting.
2. Somebody asks you to pull, and the changes sound worthy of merging.
3. However, you already dirtied the index (i.e. your index does not match the `HEAD` commit). But you know the pull you are going to make does not affect `frotz.c` or `filfre.c`, so you revert the index changes for these two files. Your changes in working tree remain there.
4. Then you can pull and merge, leaving `frotz.c` and `filfre.c` changes still in the working tree.

Undo a commit and redo

```
$ git commit ...
$ git reset --soft HEAD^      (1)
$ edit                        (2)
$ git commit -a -c ORIG_HEAD  (3)
```

1. This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message, or both. Leaves working tree as it was before "reset".
2. Make corrections to working tree files.
3. "reset" copies the old head to `.git/ORIG_HEAD`; redo the commit by starting with its log message. If you do not need to edit the message further, you can give `-C` option instead.

See also the `--amend` option to git-commit(1).

Undo a commit, making it a topic branch

```
$ git branch topic/wip          (1)
$ git reset --hard HEAD~3       (2)
$ git switch topic/wip          (3)
```

1. You have made some commits, but realize they were premature to be in the `master` branch. You want to continue polishing them in a topic branch, so create `topic/wip` branch off of the current `HEAD`.
2. Rewind the master branch to get rid of those three commits.
3. Switch to `topic/wip` branch and keep working.

Undo commits permanently

```
$ git commit ...
$ git reset --hard HEAD~3   (1)
```

1. The last three commits (`HEAD`, `HEAD^`, and `HEAD~2`) were bad and you do not want to ever see them again. Do **not** do this if you have already given these commits to somebody else. (See the "RECOVERING FROM UPSTREAM REBASE" section in git-rebase(1) for the implications of doing so.)

Undo a merge or pull

```
$ git pull                         (1)
Auto-merging nitfol
CONFLICT (content): Merge conflict in nitfol
Automatic merge failed; fix conflicts and then commit the result.
$ git reset --hard                 (2)
$ git pull . topic/branch          (3)
Updating from 41223... to 13134...
Fast-forward
$ git reset --hard ORIG_HEAD       (4)
```

1. Try to update from the upstream resulted in a lot of conflicts; you were not ready to spend a lot of time merging right now, so you decide to do that later.
2. "pull" has not made merge commit, so `git reset --hard` which is a synonym for `git reset --hard HEAD` clears the mess from the index file and the working tree.
3. Merge a topic branch into the current branch, which resulted in a fast-forward.
4. But you decided that the topic branch is not ready for public consumption yet. "pull" or "merge" always leaves the original tip of the current branch in `ORIG_HEAD`, so resetting hard to it brings your index file and the working tree back to that state, and resets the tip of the branch to that commit.

Undo a merge or pull inside a dirty working tree

```
$ git pull                         (1)
Auto-merging nitfol
Merge made by recursive.
 nitfol                |   20 +++++----
 ...
$ git reset --merge ORIG_HEAD      (2)
```

1. Even if you may have local modifications in your working tree, you can safely say `git pull` when you know that the change in the other branch does not overlap with them.

2. After inspecting the result of the merge, you may find that the change in the other branch is unsatisfactory. Running `git reset --hard ORIG_HEAD` will let you go back to where you were, but it will discard your local changes, which you do not want. `git reset --merge` keeps your local changes.

Interrupted workflow

Suppose you are interrupted by an urgent fix request while you are in the middle of a large change. The files in your working tree are not in any shape to be committed yet, but you need to get to the other branch for a quick bugfix.

```
$ git switch feature  ;# you were working in "feature" branch and
$ work work work      ;# got interrupted
$ git commit -a -m "snapshot WIP"                 (1)
$ git switch master
$ fix fix fix
$ git commit ;# commit with real log
$ git switch feature
$ git reset --soft HEAD^ ;# go back to WIP state  (2)
$ git reset                                       (3)
```

1. This commit will get blown away so a throw-away log message is OK.
2. This removes the *WIP* commit from the commit history, and sets your working tree to the state just before you made that snapshot.
3. At this point the index file still has all the WIP changes you committed as *snapshot WIP*. This updates the index to show your WIP files as uncommitted.

See also git-stash(1).

Reset a single file in the index

Suppose you have added a file to your index, but later decide you do not want to add it to your commit. You can remove the file from the index while keeping your changes with git reset.

```
$ git reset -- frotz.c                      (1)
$ git commit -m "Commit files in index"     (2)
$ git add frotz.c                           (3)
```

1. This removes the file from the index while keeping it in the working directory.
2. This commits all other changes in the index.
3. Adds the file to the index again.

Keep changes in working tree while discarding some previous commits

Suppose you are working on something and you commit it, and then you continue working a bit more, but now you think that what you have in your working tree should be in another branch that has nothing to do with what you committed previously. You can start a new branch and reset it while keeping the changes in your working tree.

```
$ git tag start
$ git switch -c branch1
$ edit
$ git commit ...                        (1)
$ edit
$ git switch -c branch2                 (2)
$ git reset --keep start                (3)
```

1. This commits your first edits in `branch1`.
2. In the ideal world, you could have realized that the earlier commit did not belong to the new topic when you created and switched to `branch2` (i.e. `git switch -c branch2 start`), but nobody is perfect.
3. But you can use `reset --keep` to remove the unwanted commit after you switched to `branch2`.

Split a commit apart into a sequence of commits

Suppose that you have created lots of logically separate changes and committed them together. Then, later you decide that it might be better to have each logical chunk associated with its own commit. You can use git reset to rewind history without changing the contents of your local files, and then successively use `git add -p` to interactively select which hunks to include into each commit, using `git commit -c` to pre-populate the commit message.

```
$ git reset -N HEAD^                    (1)
$ git add -p                            (2)
$ git diff --cached                     (3)
$ git commit -c HEAD@{1}                (4)
...                                     (5)
$ git add ...                           (6)
$ git diff --cached                     (7)
$ git commit ...                        (8)
```

1. First, reset the history back one commit so that we remove the original commit, but leave the working tree with all the changes. The -N ensures that any new files added with `HEAD` are still marked so that `git add -p` will find them.
2. Next, we interactively select diff hunks to add using the `git add -p` facility. This will ask you about each diff hunk in sequence and you can use simple commands such as "yes, include this", "No don't include this" or even the very powerful "edit" facility.
3. Once satisfied with the hunks you want to include, you should verify what has been prepared for the first commit by using `git diff --cached`. This shows all the changes that have been moved into the index and are about to be committed.
4. Next, commit the changes stored in the index. The `-c` option specifies to pre-populate the commit message from the original message that you started with in the first commit. This is helpful to avoid retyping it. The `HEAD@{1}` is a special notation for the commit that `HEAD` used to be at prior to the original reset commit (1 change ago). See git-reflog(1) for more details. You may also use any other valid commit reference.
5. You can repeat steps 2-4 multiple times to break the original code into any number of commits.
6. Now you've split out many of the changes into their own commits, and might no longer use the patch mode of `git add`, in order to select all remaining uncommitted changes.
7. Once again, check to verify that you've included what you want to. You may also wish to verify that git diff doesn't show any remaining changes to be committed later.
8. And finally create the final commit.

DISCUSSION

The tables below show what happens when running:

```
git reset --option target
```

to reset the `HEAD` to another commit (`target`) with the different reset options depending on the state of the files.

In these tables, `A`, `B`, `C` and `D` are some different states of a file. For example, the first line of the first table means that if a file is in state `A` in the working tree, in state `B` in the index, in state `C` in `HEAD` and in state `D` in the target, then `git reset --soft target` will leave the file in the working tree in state `A` and in the index in state `B`. It resets (i.e. moves) the `HEAD` (i.e. the tip of the current branch, if you are on one) to `target` (which has the file in state `D`).

```
working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    D     --soft   A       B     D
                          --mixed  A       D     D
                          --hard   D       D     D
                          --merge (disallowed)
                          --keep  (disallowed)


working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    C     --soft   A       B     C
                          --mixed  A       C     C
                          --hard   C       C     C
                          --merge (disallowed)
                          --keep   A       C     C


working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    D     --soft   B       B     D
                          --mixed  B       D     D
                          --hard   D       D     D
                          --merge  D       D     D
                          --keep  (disallowed)


working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    C     --soft   B       B     C
                          --mixed  B       C     C
                          --hard   C       C     C
                          --merge  C       C     C
                          --keep   B       C     C


working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    D     --soft   B       C     D
                          --mixed  B       D     D
                          --hard   D       D     D
                          --merge (disallowed)
                          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    C     --soft   B       C     C
                          --mixed  B       C     C
                          --hard   C       C     C
                          --merge  B       C     C
                          --keep   B       C     C
```

`reset --merge` is meant to be used when resetting out of a conflicted merge. Any mergy operation guarantees that the working tree file that is involved in the merge does not have a local change with respect to the index before it starts, and that it writes the result out to the working tree. So if we see some difference between the index and the target and also between the index and the working tree, then it means that we are not resetting out from a state that a mergy operation left after failing with a conflict. That is why we disallow `--merge` option in this case.

`reset --keep` is meant to be used when removing some of the last commits in the current branch while keeping changes in the working tree. If there could be conflicts between the changes in the commit we want to remove and the changes in the working tree we want to keep, the reset is disallowed. That's why it is disallowed if there are both changes between the working tree and `HEAD`, and between `HEAD` and the target. To be safe, it is also disallowed when there are unmerged entries.

The following tables show what happens when there are unmerged entries:

```
working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    B     --soft  (disallowed)
                          --mixed  X       B     B
                          --hard   B       B     B
                          --merge  B       B     B
                          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    A     --soft  (disallowed)
                          --mixed  X       A     A
                          --hard   A       A     A
                          --merge  A       A     A
                          --keep  (disallowed)
```

`X` means any state and `U` means an unmerged index.

GIT

Part of the git(1) suite

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