NAME

git-checkout - Checkout a branch or paths to the working tree

SYNOPSIS

git checkout [-q] [-f] [-m] [<branch>]
git checkout [-q] [-f] [-m] --detach [<branch>]
git checkout [-q] [-f] [-m] [--detach] <commit>
git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new_branch>] [<start_point>]
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>...
git checkout [-p|--patch] [<tree-ish>] [--] [<paths>...]

DESCRIPTION

Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, *git checkout* will also update HEAD to set the specified branch as the current branch.

git checkout <branch>

To prepare for working on <branch>, switch to it by updating the index and the files in the working tree, and by pointing HEAD at the branch. Local modifications to the files in the working tree are kept, so that they can be committed to the <branch>.

If <branch> is not found but there does exist a tracking branch in exactly one remote (call it <remote>) with a matching name, treat as equivalent to

$ git checkout -b <branch> --track <remote>/<branch>

You could omit <branch>, in which case the command degenerates to check out the current branch, which is a glorified no-op with a rather expensive side-effects to show only the tracking information, if exists, for the current branch.

git checkout -b|-B <new_branch> [<start point>]

Specifying -b causes a new branch to be created as if **git-branch(1)** were called and then checked out. In this case you can use the --track or --no-track options, which will be passed to *git branch*. As a convenience, --track without -b implies branch creation; see the description of --track below.

If -B is given, <new_branch> is created if it doesn't exist; otherwise, it is reset. This is the transactional equivalent of

$ git branch -f <branch> [<start point>]
$ git checkout <branch>

that is to say, the branch is not reset/created unless git checkout is successful.

git checkout --detach [<branch>], *git checkout* [--detach] <commit>

Prepare to work on top of <commit>, by detaching HEAD at it (see DETACHED HEAD section), and updating the index and the files in the working tree. Local modifications to the files in the working tree are kept, so that the resulting working tree will be the state recorded in the commit plus the local modifications.

When the <commit> argument is a branch name, the --detach option can be used to detach HEAD at the tip of the branch (git checkout <branch> would check out that branch without detaching HEAD).

Omitting <branch> detaches HEAD at the tip of the current branch.

git checkout [-p|--patch] [<tree-ish>] [--] <pathspec>...

When <paths> or --patch are given, *git checkout* does **not** switch branches. It updates the named paths in the working tree from the index file or from a named <tree-ish> (most often a commit). In this case, the -b and --track options are meaningless and giving either of them results in an error. The <tree-ish> argument can be used to specify a specific tree-ish (i.e. commit, tag or tree) to update the index for the given paths before updating the working

tree.

The index may contain unmerged entries because of a previous failed merge. By default, if you try to check out such an entry from the index, the checkout operation will fail and nothing will be checked out. Using -f will ignore these unmerged entries. The contents from a specific side of the merge can be checked out of the index by using --ours or --theirs. With -m, changes made to the working tree file can be discarded to re-create the original conflicted merge result.

OPTIONS

-q, --quiet
Quiet, suppress feedback messages.

-f, --force
When switching branches, proceed even if the index or the working tree differs from HEAD. This is used to throw away local changes.

When checking out paths from the index, do not fail upon unmerged entries; instead, unmerged entries are ignored.

--ours, --theirs
When checking out paths from the index, check out stage #2 (*ours*) or #3 (*theirs*) for unmerged paths.

-b <new_branch>
Create a new branch named <new_branch> and start it at <start_point>; see **git-branch(1)** for details.

-B <new_branch>
Creates the branch <new_branch> and start it at <start_point>; if it already exists, then reset it to <start_point>. This is equivalent to running git branch with -f; see **git-branch(1)** for details.

-t, --track
When creating a new branch, set up upstream configuration. See --track in **git-branch(1)** for details.

If no *-b* option is given, the name of the new branch will be derived from the remote-tracking branch, by looking at the local part of the refspec configured for the corresponding remote, and then stripping the initial part up to the *. This would tell us to use hack as the local branch when branching off of origin/hack (or remotes/origin/hack, or even refs/remotes/origin/hack). If the given name has no slash, or the above guessing results in an empty name, the guessing is aborted. You can explicitly give a name with *-b* in such a case.

--no-track
Do not set up upstream configuration, even if the branch.autosetupmerge configuration variable is true.

-l
Create the new branch's reflog; see **git-branch(1)** for details.

--detach
Rather than checking out a branch to work on it, check out a commit for inspection and discardable experiments. This is the default behavior of git checkout <commit> when <commit> is not a branch name. See the DETACHED HEAD section below for details.

--orphan <new_branch>
Create a new *orphan* branch, named <new_branch>, started from <start_point> and switch to it. The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected from all the other branches and commits.

The index and the working tree are adjusted as if you had previously run git checkout

<start_point>. This allows you to start a new history that records a set of paths similar to <start_point> by easily running git commit -a to make the root commit.

This can be useful when you want to publish the tree from a commit without exposing its full history. You might want to do this to publish an open source branch of a project whose current tree is clean, but whose full history contains proprietary or otherwise encumbered bits of code.

If you want to start a disconnected history that records a set of paths that is totally different from the one of <start_point>, then you should clear the index and the working tree right after creating the orphan branch by running git rm -rf . from the top level of the working tree. Afterwards you will be ready to prepare your new files, repopulating the working tree, by copying them from elsewhere, extracting a tarball, etc.

--ignore-skip-worktree-bits
In sparse checkout mode, git checkout -- <paths> would update only entries matched by <paths> and sparse patterns in $GIT_DIR/info/sparse-checkout. This option ignores the sparse patterns and adds back any files in <paths>.

-m, --merge
When switching branches, if you have local modifications to one or more files that are different between the current branch and the branch to which you are switching, the command refuses to switch branches in order to preserve your modifications in context. However, with this option, a three-way merge between the current branch, your working tree contents, and the new branch is done, and you will be on the new branch.

When a merge conflict happens, the index entries for conflicting paths are left unmerged, and you need to resolve the conflicts and mark the resolved paths with git add (or git rm if the merge should result in deletion of the path).

When checking out paths from the index, this option lets you recreate the conflicted merge in the specified paths.

--conflict=<style>
The same as --merge option above, but changes the way the conflicting hunks are presented, overriding the merge.conflictstyle configuration variable. Possible values are merge (default) and diff3 (in addition to what is shown by merge style, shows the original contents).

-p, --patch
Interactively select hunks in the difference between the <tree-ish> (or the index, if unspecified) and the working tree. The chosen hunks are then applied in reverse to the working tree (and if a <tree-ish> was specified, the index).

This means that you can use git checkout -p to selectively discard edits from your current working tree. See the “Interactive Mode” section of **git-add(1)** to learn how to operate the --patch mode.

<branch>
Branch to checkout; if it refers to a branch (i.e., a name that, when prepended with refs/heads/, is a valid ref), then that branch is checked out. Otherwise, if it refers to a valid commit, your HEAD becomes detached and you are no longer on any branch (see below for details).

As a special case, the @{-N} syntax for the N-th last branch/commit checks out branches (instead of detaching). You may also specify - which is synonymous with @{-1}.

As a further 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.

<new_branch>
Name for the new branch.

<start_point>
The name of a commit at which to start the new branch; see **git-branch(1)** for details. Defaults to HEAD.

<tree-ish>
Tree to checkout from (when paths are given). If not specified, the index will be used.

DETACHED HEAD

HEAD normally refers to a named branch (e.g. *master*). Meanwhile, each branch refers to a specific commit. Let's look at a repo with three commits, one of them tagged, and with branch *master* checked out:

HEAD (refers to branch master)
|
v
a---b---c branch master (refers to commit c)
^
|
tag v2.0 (refers to commit b)

When a commit is created in this state, the branch is updated to refer to the new commit. Specifically, *git commit* creates a new commit *d*, whose parent is commit *c*, and then updates branch *master* to refer to new commit *d*. HEAD still refers to branch *master* and so indirectly now refers to commit *d*:

$ edit; git add; git commit

HEAD (refers to branch master)
|
v
a---b---c---d branch master (refers to commit d)
^
|
tag v2.0 (refers to commit b)

It is sometimes useful to be able to checkout a commit that is not at the tip of any named branch, or even to create a new commit that is not referenced by a named branch. Let's look at what happens when we checkout commit *b* (here we show two ways this may be done):

$ git checkout v2.0 # or
$ git checkout master^^

HEAD (refers to commit b)
|
v
a---b---c---d branch master (refers to commit d)
^
|
tag v2.0 (refers to commit b)

Notice that regardless of which checkout command we use, HEAD now refers directly to commit *b*. This is known as being in detached HEAD state. It means simply that HEAD refers to a specific commit, as opposed to referring to a named branch. Let's see what happens when we create a commit:

$ edit; git add; git commit

HEAD (refers to commit e)
|
v
e

/
a---b---c---d branch master (refers to commit d)
^
|
tag v2.0 (refers to commit b)

There is now a new commit *e*, but it is referenced only by HEAD. We can of course add yet another commit in this state:

$ edit; git add; git commit

HEAD (refers to commit f)
|
v
e---f
/
a---b---c---d branch master (refers to commit d)
^
|
tag v2.0 (refers to commit b)

In fact, we can perform all the normal Git operations. But, let's look at what happens when we then checkout master:

$ git checkout master

HEAD (refers to branch master)
e---f |
/ v
a---b---c---d branch master (refers to commit d)
^
|
tag v2.0 (refers to commit b)

It is important to realize that at this point nothing refers to commit *f*. Eventually commit *f* (and by extension commit *e*) will be deleted by the routine Git garbage collection process, unless we create a reference before that happens. If we have not yet moved away from commit *f*, any of these will create a reference to it:

$ git checkout -b foo **(1)**
$ git branch foo **(2)**
$ git tag foo **(3)**

1. creates a new branch *foo*, which refers to commit *f*, and then updates HEAD to refer to branch *foo*. In other words, we'll no longer be in detached HEAD state after this command.
2. similarly creates a new branch *foo*, which refers to commit *f*, but leaves HEAD detached.
3. creates a new tag *foo*, which refers to commit *f*, leaving HEAD detached.

If we have moved away from commit *f*, then we must first recover its object name (typically by using git reflog), and then we can create a reference to it. For example, to see the last two commits to which HEAD referred, we can use either of these commands:

$ git reflog -2 HEAD # or
$ git log -g -2 HEAD

EXAMPLES

1. The following sequence checks out the master branch, reverts the Makefile to two revisions back, deletes hello.c by mistake, and gets it back from the index.

   $ git checkout master **(1)**
   $ git checkout master~2 Makefile **(2)**

$ rm -f hello.c
$ git checkout hello.c **(3)**

1. switch branch
2. take a file out of another commit
3. restore hello.c from the index

If you want to check out *all* C source files out of the index, you can say

$ git checkout -- *.c

Note the quotes around *.c. The file hello.c will also be checked out, even though it is no longer in the working tree, because the file globbing is used to match entries in the index (not in the working tree by the shell).

If you have an unfortunate branch that is named hello.c, this step would be confused as an instruction to switch to that branch. You should instead write:

$ git checkout -- hello.c

2. After working in the wrong branch, switching to the correct branch would be done using:

   $ git checkout mytopic

   However, your wrong branch and correct mytopic branch may differ in files that you have modified locally, in which case the above checkout would fail like this:

   $ git checkout mytopic
   error: You have local changes to frotz; not switching branches.

   You can give the -m flag to the command, which would try a three-way merge:

   $ git checkout -m mytopic
   Auto-merging frotz

   After this three-way merge, the local modifications are *not* registered in your index file, so git diff would show you what changes you made since the tip of the new branch.
3. When a merge conflict happens during switching branches with the -m option, you would see something like this:

   $ git checkout -m mytopic
   Auto-merging frotz
   ERROR: Merge conflict in frotz
   fatal: merge program failed

   At this point, git diff shows the changes cleanly merged as in the previous example, as well as the changes in the conflicted files. Edit and resolve the conflict and mark it resolved with git add as usual:

   $ edit frotz
   $ git add frotz

GIT

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