

**NAME**

git-pull - Fetch from and integrate with another repository or a local branch

**SYNOPSIS**

```
git pull [options] [<repository> [<refspec>...]]
```

**DESCRIPTION**

Incorporates changes from a remote repository into the current branch. In its default mode, git pull is shorthand for git fetch followed by git merge FETCH\_HEAD.

More precisely, *git pull* runs *git fetch* with the given parameters and calls *git merge* to merge the retrieved branch heads into the current branch. With `--rebase`, it runs *git rebase* instead of *git merge*.

<repository> should be the name of a remote repository as passed to [git-fetch\(1\)](#). <refspec> can name an arbitrary remote ref (for example, the name of a tag) or even a collection of refs with corresponding remote-tracking branches (e.g., refs/heads/\*:refs/remotes/origin/\*), but usually it is the name of a branch in the remote repository.

Default values for <repository> and <branch> are read from the remote and merge configuration for the current branch as set by [git-branch\(1\)](#) `--track`.

Assume the following history exists and the current branch is master:

```
A---B---C master on origin
/
D---E---F---G master
^
origin/master in your repository
```

Then git pull will fetch and replay the changes from the remote master branch since it diverged from the local master (i.e., E) until its current commit (C) on top of master and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.

```
A---B---C origin/master
/
D---E---F---G---H master
```

See [git-merge\(1\)](#) for details, including how conflicts are presented and handled.

In Git 1.7.0 or later, to cancel a conflicting merge, use `git reset --merge`. **Warning:** In older versions of Git, running *git pull* with uncommitted changes is discouraged: while possible, it leaves you in a state that may be hard to back out of in the case of a conflict.

If any of the remote changes overlap with local uncommitted changes, the merge will be automatically cancelled and the work tree untouched. It is generally best to get any local changes in working order before pulling or stash them away with [git-stash\(1\)](#).

**OPTIONS**

Options meant for *git pull* itself and the underlying *git merge* must be given before the options meant for *git fetch*.

`-q, --quiet`

This is passed to both underlying git-fetch to squelch reporting of during transfer, and underlying git-merge to squelch output during merging.

`-v, --verbose`

Pass `--verbose` to git-fetch and git-merge.

`--[no]-recurse-submodules[=yes|on-demand|no]`

This option controls if new commits of all populated submodules should be fetched too (see [git-config\(1\)](#) and [gitmodules\(5\)](#)). That might be necessary to get the data needed for

merging submodule commits, a feature Git learned in 1.7.3. Notice that the result of a merge will not be checked out in the submodule, `git submodule update` has to be called afterwards to bring the work tree up to date with the merge result.

### Options related to merging

`--commit`, `--no-commit`

Perform the merge and commit the result. This option can be used to override `--no-commit`.

With `--no-commit` perform the merge but pretend the merge failed and do not autoccommit, to give the user a chance to inspect and further tweak the merge result before committing.

`--edit`, `-e`, `--no-edit`

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. The `--no-edit` option can be used to accept the auto-generated message (this is generally discouraged).

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened when they run `git merge`. To make it easier to adjust such scripts to the updated behaviour, the environment variable `GIT_MERGE_AUTOEDIT` can be set to `no` at the beginning of them.

`--ff`

When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

`--no-ff`

Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag.

`--ff-only`

Refuse to merge and exit with a non-zero status unless the current HEAD is already up-to-date or the merge can be resolved as a fast-forward.

`--log[=<n>]`, `--no-log`

In addition to branch names, populate the log message with one-line descriptions from at most `<n>` actual commits that are being merged. See also [git-fmt-merge-msg\(1\)](#).

With `--no-log` do not list one-line descriptions from the actual commits being merged.

`--stat`, `-n`, `--no-stat`

Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option `merge.stat`.

With `-n` or `--no-stat` do not show a diffstat at the end of the merge.

`--squash`, `--no-squash`

Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit, move the HEAD, or record `$GIT_DIR/MERGE_HEAD` (to cause the next `git commit` command to create a merge commit). This allows you to create a single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With `--no-squash` perform the merge and commit the result. This option can be used to override `--squash`.

`-s <strategy>`, `--strategy=<strategy>`

Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no `-s` option, a built-in list of strategies is used instead (*git merge-recursive* when merging a single head, *git merge-octopus* otherwise).

`-X <option>`, `--strategy-option=<option>`

Pass merge strategy specific option through to the merge strategy.

`--verify-signatures`, `--no-verify-signatures`

Verify that the commits being merged have good and trusted GPG signatures and abort the merge in case they do not.

--summary, --no-summary

Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

-r, --rebase[=*false|true|preserve*]

When true, rebase the current branch on top of the upstream branch after fetching. If there is a remote-tracking branch corresponding to the upstream branch and the upstream branch was rebased since last fetched, the rebase uses that information to avoid rebasing non-local changes.

When preserve, also rebase the current branch on top of the upstream branch, but pass --preserve-merges along to git rebase so that locally created merge commits will not be flattened.

When false, merge the current branch into the upstream branch.

See pull.rebase, branch.<name>.rebase and branch.autosetuprebase in [git-config\(1\)](#) if you want to make git pull always use --rebase instead of merging.

#### Note

This is a potentially *dangerous* mode of operation. It rewrites history, which does not bode well when you published that history already. Do **not** use this option unless you have read [git-rebase\(1\)](#) carefully.

--no-rebase

Override earlier --rebase.

### Options related to fetching

--all

Fetch all remotes.

-a, --append

Append ref names and object names of fetched refs to the existing contents of .git/FETCH\_HEAD. Without this option old data in .git/FETCH\_HEAD will be overwritten.

--depth=*<depth>*

Deepen or shorten the history of a *shallow* repository created by git clone with --depth=*<depth>* option (see [git-clone\(1\)](#)) to the specified number of commits from the tip of each remote branch history. Tags for the deepened commits are not fetched.

--unshallow

If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow

By default when fetching from a shallow repository, git fetch refuses refs that require updating .git/shallow. This option updates .git/shallow and accept such refs.

-f, --force

When *git fetch* is used with *<rbranch>:<lbranch>* refspect, it refuses to update the local branch *<lbranch>* unless the remote branch *<rbranch>* it fetches is a descendant of *<lbranch>*. This option overrides that check.

-k, --keep

Keep downloaded pack.

--no-tags

By default, tags that point at objects that are downloaded from the remote repository are

fetches and stores locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the `remote.<name>.tagopt` setting. See [git-config\(1\)](#).

`-u, --update-head-ok`

By default *git fetch* refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for *git pull* to communicate with *git fetch*, and unless you are implementing your own Porcelain you are not supposed to use it.

`--upload-pack <upload-pack>`

When given, and the repository to fetch from is handled by *git fetch-pack*, `--exec=<upload-pack>` is passed to the command to specify non-default path for the command run on the other end.

`--progress`

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless `-q` is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

`<repository>`

The remote repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

`<refspec>`

Specifies which refs to fetch and which local refs to update. When no `<refspec>`s appear on the command line, the refs to fetch are read from `remote.<repository>.fetch` variables instead (see [git-fetch\(1\)](#)).

The format of a `<refspec>` parameter is an optional plus `+`, followed by the source ref `<src>`, followed by a colon `:`, followed by the destination ref `<dst>`. The colon can be omitted when `<dst>` is empty.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`; it requests fetching everything up to the given tag.

The remote ref that matches `<src>` is fetched, and if `<dst>` is not empty string, the local ref that matches it is fast-forwarded using `<src>`. If the optional plus `+` is used, the local ref is updated even if it does not result in a fast-forward update.

#### Note

When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the `+` sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.

#### Note

There is a difference between listing multiple `<refspec>` directly on *git pull* command line and having multiple `remote.<repository>.fetch` entries in your configuration for a `<repository>` and running a *git pull* command without any explicit `<refspec>` parameters. `<refspec>`s listed explicitly on the command line are always merged into the current branch after fetching. In other words, if you list more than one remote ref, *git pull* will create an Octopus merge. On the other hand, if you do not list any explicit `<refspec>` parameter on the command line, *git pull* will fetch all the `<refspec>`s it finds in the `remote.<repository>.fetch` configuration and merge only the first `<refspec>` found into the current branch. This is because making an Octopus from remote refs is rarely done, while keeping track of multiple remote heads in one-go by fetching more than one is often useful.

## GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp, and ftps can be used for fetching and rsync can be used for fetching and pushing, but these are inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- ssh://[user@]host.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/
- rsync://host.xz/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path foo:bar could be specified as an absolute path or ./foo:bar to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support ~username expansion:

- ssh://[user@]host.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

For local repositories, also supported by Git natively, the following syntaxes may be used:

- /path/to/repo.git/
- file:///path/to/repo.git/

These two syntaxes are mostly equivalent, except when cloning, when the former implies --local option. See [git-clone\(1\)](#) for details.

When Git doesn't know how to handle a certain transport protocol, it attempts to use the *remote-<transport>* remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- <transport>::<address>

where <address> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See [gitremote-helpers\(1\)](#) for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url <actual url base>]
insteadOf = <other url base>
```

For example, with this:

```
[url git://git.host.xz/]
insteadOf = host.xz:/path/to/
insteadOf = work:
```

a URL like work:repo.git or like host.xz:/path/to/repo.git will be rewritten in any context that takes a URL to be git://git.host.xz/repo.git.

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url <actual url base>]
pushInsteadOf = <other url base>
```

For example, with this:

```
[url ssh://example.org/]
pushInsteadOf = git://example.org/
```

a URL like `git://example.org/path/to/repo.git` will be rewritten to `ssh://example.org/path/to/repo.git` for pushes, but pulls will still use the original URL.

## REMOTES

The name of one of the following can be used instead of a URL as `<repository>` argument:

- a remote in the Git configuration file: `$GIT_DIR/config`,
- a file in the `$GIT_DIR/remotes` directory, or
- a file in the `$GIT_DIR/branches` directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

### Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using [git-remote\(1\)](#), [git-config\(1\)](#) or even by a manual edit to the `$GIT_DIR/config` file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote <name>]
url = <url>
pushurl = <pushurl>
push = <refspec>
fetch = <refspec>
```

The `<pushurl>` is used for pushes only. It is optional and defaults to `<url>`.

### Named file in `$GIT_DIR/remotes`

You can choose to provide the name of a file in `$GIT_DIR/remotes`. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL format
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by `git push` and Pull: lines are used by `git pull` and `git fetch`. Multiple Push: and Pull: lines may be specified for additional branch mappings.

### Named file in `$GIT_DIR/branches`

You can choose to provide the name of a file in `$GIT_DIR/branches`. The URL in this file will be used to access the repository. This file should have the following format:

```
<url>#<head>
```

`<url>` is required; `#<head>` is optional.

Depending on the operation, git will use one of the following refspecs, if you don't provide one on the command line. `<branch>` is the name of this file in `$GIT_DIR/branches` and `<head>` defaults to master.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

HEAD:refs/heads/<head>

## 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.

## DEFAULT BEHAVIOUR

Often people use `git pull` without giving any parameter. Traditionally, this has been equivalent to saying `git pull origin`. However, when configuration `branch.<name>.remote` is present while on branch `<name>`, that value is used instead of `origin`.

In order to determine what URL to use to fetch from, the value of the configuration `remote.<origin>.url` is consulted and if there is not any such variable, the value on URL: ' line in `$(GIT_DIR)/remotes/<origin>` file is used.

In order to determine what remote branches to fetch (and optionally store in the remote-tracking branches) when the command is run without any `refspec` parameters on the command line, values of the configuration variable `remote.<origin>.fetch` are consulted, and if there aren't any, `$(GIT_DIR)/remotes/<origin>` file is consulted and its 'Pull: ' lines are used. In addition to the `refspec` formats described in the `OPTIONS` section, you can have a globbing `refspec` that looks like this:

```
refs/heads/*:refs/remotes/origin/*
```

A globbing `refspec` must have a non-empty RHS (i.e. must store what were fetched in remote-tracking branches), and its LHS and RHS must end with `/*`. The above specifies that all remote branches are tracked using remote-tracking branches in `refs/remotes/origin/` hierarchy under the same name.

The rule to determine which remote branch to merge after fetching is a bit involved, in order not to break backward compatibility.



If explicit refsspecs were given on the command line of `git pull`, they are all merged.

When no refspect was given on the command line, then `git pull` uses the refspect from the configuration or `$GIT_DIR/remotes/<origin>`. In such cases, the following rules apply:

1. If `branch.<name>.merge` configuration for the current branch `<name>` exists, that is the name of the branch at the remote site that is merged.
2. If the refspect is a globbing one, nothing is merged.
3. Otherwise the remote branch of the first refspect is merged.

## EXAMPLES

- Update the remote-tracking branches for the repository you cloned from, then merge one of them into your current branch:

```
$ git pull, git pull origin
```

Normally the branch merged in is the HEAD of the remote repository, but the choice is determined by the `branch.<name>.remote` and `branch.<name>.merge` options; see [git-config\(1\)](#) for details.

- Merge into the current branch the remote branch next:

```
$ git pull origin next
```

This leaves a copy of next temporarily in `FETCH_HEAD`, but does not update any remote-tracking branches. Using remote-tracking branches, the same can be done by invoking `fetch` and `merge`:

```
$ git fetch origin
$ git merge origin/next
```

If you tried a pull which resulted in complex conflicts and would want to start over, you can recover with `git reset`.

## BUGS

Using `--recurse-submodules` can only fetch new commits in already checked out submodules right now. When e.g. upstream added a new submodule in the just fetched commits of the superproject the submodule itself can not be fetched, making it impossible to check out that submodule later without having to do a fetch again. This is expected to be fixed in a future Git version.

## SEE ALSO

[git-fetch\(1\)](#), [git-merge\(1\)](#), [git-config\(1\)](#)

## GIT

Part of the [git\(1\)](#) suite