

**NAME**

git-rev-parse - Pick out and massage parameters

**SYNOPSIS**

*git rev-parse* [ --option ] <args>...

**DESCRIPTION**

Many Git porcelainish commands take mixture of flags (i.e. parameters that begin with a dash -) and parameters meant for the underlying *git rev-list* command they use internally and flags and parameters for the other commands they use downstream of *git rev-list*. This command is used to distinguish between them.

**OPTIONS****Operation Modes**

Each of these options must appear first on the command line.

--parseopt

Use *git rev-parse* in option parsing mode (see PARSEOPT section below).

--sq-quote

Use *git rev-parse* in shell quoting mode (see SQ-QUOTE section below). In contrast to the --sq option below, this mode does only quoting. Nothing else is done to command input.

**Options for --parseopt**

--keep-dashdash

Only meaningful in --parseopt mode. Tells the option parser to echo out the first -- met instead of skipping it.

--stop-at-non-option

Only meaningful in --parseopt mode. Lets the option parser stop at the first non-option argument. This can be used to parse sub-commands that take options themselves.

--stuck-long

Only meaningful in --parseopt mode. Output the options in their long form if available, and with their arguments stuck.

**Options for Filtering**

--revs-only

Do not output flags and parameters not meant for *git rev-list* command.

--no-revs

Do not output flags and parameters meant for *git rev-list* command.

--flags

Do not output non-flag parameters.

--no-flags

Do not output flag parameters.

**Options for Output**

--default <arg>

If there is no parameter given by the user, use <arg> instead.

--prefix <arg>

Behave as if *git rev-parse* was invoked from the <arg> subdirectory of the working tree. Any relative filenames are resolved as if they are prefixed by <arg> and will be printed in that form.

This can be used to convert arguments to a command run in a subdirectory so that they can still be used after moving to the top-level of the repository. For example:

```
prefix=$(git rev-parse --show-prefix)
cd $(git rev-parse --show-toplevel)
```

eval set -- \$(git rev-parse --sq --prefix \$prefix \$@)

--verify

Verify that exactly one parameter is provided, and that it can be turned into a raw 20-byte SHA-1 that can be used to access the object database. If so, emit it to the standard output; otherwise, error out.

If you want to make sure that the output actually names an object in your object database and/or can be used as a specific type of object For example, `git rev-parse $VAR^{commit}` will make sure \$VAR names an existing object that is a commit-ish (i.e. a commit, or an annotated tag that points at a commit). To make sure that \$VAR names an existing object of any type, `git rev-parse $VAR^{object}` can be used.

-q, --quiet

Only meaningful in --verify mode. Do not output an error message if the first argument is not a valid object name; instead exit with non-zero status silently.

--sq

Usually the output is made one line per flag and parameter. This option makes output a single line, properly quoted for consumption by shell. Useful when you expect your parameter to contain whitespaces and newlines (e.g. when using pickaxe -S with *git diff*-\*). In contrast to the --sq-quote option, the command input is still interpreted as usual.

--not

When showing object names, prefix them with ^ and strip ^ prefix from the object names that already have one.

--abbrev-ref[=(strict|loose)]

A non-ambiguous short name of the objects name. The option `core.warnAmbiguousRefs` is used to select the strict abbreviation mode.

--short, --short=number

Instead of outputting the full SHA-1 values of object names try to abbreviate them to a shorter unique name. When no length is specified 7 is used. The minimum length is 4.

--symbolic

Usually the object names are output in SHA-1 form (with possible ^ prefix); this option makes them output in a form as close to the original input as possible.

--symbolic-full-name

This is similar to --symbolic, but it omits input that are not refs (i.e. branch or tag names; or more explicitly disambiguating heads/master form, when you want to name the master branch when there is an unfortunately named tag master), and show them as full refnames (e.g. refs/heads/master).

### Options for Objects

--all

Show all refs found in refs/.

--branches[=pattern], --tags[=pattern], --remotes[=pattern]

Show all branches, tags, or remote-tracking branches, respectively (i.e., refs found in refs/heads, refs/tags, or refs/remotes, respectively).

If a pattern is given, only refs matching the given shell glob are shown. If the pattern does not contain a globbing character (? , \* , or []), it is turned into a prefix match by appending /\*.

--glob=pattern

Show all refs matching the shell glob pattern pattern. If the pattern does not start with refs/, this is automatically prepended. If the pattern does not contain a globbing character (? , \* , or []), it is turned into a prefix match by appending /\*.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing `/*` is intended, it must be given explicitly.

`--disambiguate=<prefix>`

Show every object whose name begins with the given prefix. The `<prefix>` must be at least 4 hexadecimal digits long to avoid listing each and every object in the repository by mistake.

### Options for Files

`--local-env-vars`

List the `GIT_*` environment variables that are local to the repository (e.g. `GIT_DIR` or `GIT_WORK_TREE`, but not `GIT_EDITOR`). Only the names of the variables are listed, not their value, even if they are set.

`--git-dir`

Show `$GIT_DIR` if defined. Otherwise show the path to the `.git` directory. The path shown, when relative, is relative to the current working directory.

If `$GIT_DIR` is not defined and the current directory is not detected to lie in a Git repository or work tree print a message to `stderr` and exit with nonzero status.

`--is-inside-git-dir`

When the current working directory is below the repository directory print true, otherwise false.

`--is-inside-work-tree`

When the current working directory is inside the work tree of the repository print true, otherwise false.

`--is-bare-repository`

When the repository is bare print true, otherwise false.

`--resolve-git-dir <path>`

Check if `<path>` is a valid repository or a gitfile that points at a valid repository, and print the location of the repository. If `<path>` is a gitfile then the resolved path to the real repository is printed.

`--show-cdup`

When the command is invoked from a subdirectory, show the path of the top-level directory relative to the current directory (typically a sequence of `../`, or an empty string).

`--show-prefix`

When the command is invoked from a subdirectory, show the path of the current directory relative to the top-level directory.

`--show-toplevel`

Show the absolute path of the top-level directory.

`--shared-index-path`

Show the path to the shared index file in split index mode, or empty if not in split-index mode.

### Other Options

`--since=datestring`, `--after=datestring`

Parse the date string, and output the corresponding `--max-age=` parameter for *git rev-list*.

`--until=datestring`, `--before=datestring`

Parse the date string, and output the corresponding `--min-age=` parameter for *git rev-list*.

<args>...

Flags and parameters to be parsed.

## SPECIFYING REVISIONS

A revision parameter <rev> typically, but not necessarily, names a commit object. It uses what is called an *extended SHA-1* syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

<sha1>, e.g. *dae86e1950b1277e545cee180551750029cfe735*, *dae86e*

The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is unique within the repository. E.g. *dae86e1950b1277e545cee180551750029cfe735* and *dae86e* both name the same commit object if there is no other object in your repository whose object name starts with *dae86e*.

<describeOutput>, e.g. *v1.7.4.2-679-g3bee7fb*

Output from `git describe`; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a *g*, and an abbreviated object name.

<refname>, e.g. *master*, *heads/master*, *refs/heads/master*

A symbolic ref name. E.g. *master* typically means the commit object referenced by *refs/heads/master*. If you happen to have both *heads/master* and *tags/master*, you can explicitly say *heads/master* to tell Git which one you mean. When ambiguous, a <refname> is disambiguated by taking the first match in the following rules:

1. If *GIT\_DIR/<refname>* exists, that is what you mean (this is usually useful only for *HEAD*, *FETCH\_HEAD*, *ORIG\_HEAD*, *MERGE\_HEAD* and *CHERRY\_PICK\_HEAD*);
2. otherwise, *refs/<refname>* if it exists;
3. otherwise, *refs/tags/<refname>* if it exists;
4. otherwise, *refs/heads/<refname>* if it exists;
5. otherwise, *refs/remotes/<refname>* if it exists;
6. otherwise, *refs/remotes/<refname>/HEAD* if it exists.

*HEAD* names the commit on which you based the changes in the working tree. *FETCH\_HEAD* records the branch which you fetched from a remote repository with your last `git fetch` invocation. *ORIG\_HEAD* is created by commands that move your *HEAD* in a drastic way, to record the position of the *HEAD* before their operation, so that you can easily change the tip of the branch back to the state before you ran them. *MERGE\_HEAD* records the commit(s) which you are merging into your branch when you run `git merge`. *CHERRY\_PICK\_HEAD* records the commit which you are cherry-picking when you run `git cherry-pick`.

Note that any of the *refs/\** cases above may come either from the *GIT\_DIR/refs* directory or from the *GIT\_DIR/packed-refs* file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

@

@ alone is a shortcut for *HEAD*.

<refname>@{<date>}, e.g. *master@{yesterday}*, *HEAD@{5 minutes ago}*

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. *{yesterday}*, *{1 month 2 weeks 3 days 1 hour 1 second ago}* or *{1979-02-26 18:30:00}*) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log

(*GIT\_DIR/logs/<ref>*). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *master* branch last week. If you want to look at commits made during certain times, see `--since` and `--until`.

<refname>@{<n>}, e.g. *master@{1}*

A ref followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g. *{1}*,

{15}) specifies the n-th prior value of that ref. For example *master@{1}* is the immediate prior value of *master* while *master@{5}* is the 5th prior value of *master*. This suffix may only be used immediately following a ref name and the ref must have an existing log (*GIT\_DIR/logs/<refname>*).

@{<n>}, e.g. @{1}

You can use the @ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch *blabla* then @{1} means the same as *blabla@{1}*.

@{-<n>}, e.g. @{-1}

The construct @{-<n>} means the <n>th branch/commit checked out before the current one.

<branchname>@{upstream}, e.g. *master@{upstream}*, @{u}

The suffix @{upstream} to a branchname (short form <branchname>@{u}) refers to the branch that the branch specified by branchname is set to build on top of (configured with *branch.<name>.remote* and *branch.<name>.merge*). A missing branchname defaults to the current one.

<rev>^, e.g. *HEAD^*, *v1.5.1^0*

A suffix ^ to a revision parameter means the first parent of that commit object. ^<n> means the <n>th parent (i.e. <rev>^ is equivalent to <rev>^1). As a special rule, <rev>^0 means the commit itself and is used when <rev> is the object name of a tag object that refers to a commit object.

<rev>~<n>, e.g. *master~3*

A suffix ~<n> to a revision parameter means the commit object that is the <n>th generation ancestor of the named commit object, following only the first parents. I.e. <rev>~3 is equivalent to <rev>^^^ which is equivalent to <rev>^1^1^1. See below for an illustration of the usage of this form.

<rev>^{<type>}, e.g. *v0.99.8^{commit}*

A suffix ^ followed by an object type name enclosed in brace pair means dereference the object at <rev> recursively until an object of type <type> is found or the object cannot be dereferenced anymore (in which case, barf). For example, if <rev> is a commit-ish, <rev>^{commit} describes the corresponding commit object. Similarly, if <rev> is a tree-ish, <rev>^{tree} describes the corresponding tree object. <rev>^0 is a short-hand for <rev>^{commit}.

*rev^{object}* can be used to make sure *rev* names an object that exists, without requiring *rev* to be a tag, and without dereferencing *rev*; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

*rev^{tag}* can be used to ensure that *rev* identifies an existing tag object.

<rev>^{}, e.g. *v0.99.8^{}*

A suffix ^ followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

<rev>^{<text>}, e.g. *HEAD^{/fix nasty bug}*

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the *:/fix nasty bug* syntax below except that it returns the youngest matching commit which is reachable from the <rev> before ^.

:/<text>, e.g. *:/fix nasty bug*

A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref. If the commit message starts with a ! you have to repeat that; the special sequence *:/!*, followed by something else than !, is reserved for now. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*.

`<rev>:<path>`, e.g. `HEAD:README`, `:README`, `master:./README`

A suffix `:` followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. `:path` (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. A path starting with `./` or `../` is relative to the current working directory. The given path will be converted to be relative to the working tree's root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

`:<n>:<path>`, e.g. `:0:README`, `:README`

A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch's version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.

```
G H I J
 / /
D E F
 | /
 | / |
 | / |
B C
 /
 /
A
```

$A = A^0$

$B = A^1 = A^1_1 = A^1_1$

$C = A^2 = A^2_2$

$D = A^{11} = A^{11}_1 = A^2_2$

$E = B^2 = A^{22}$

$F = B^3 = A^{33}$

$G = A^{111} = A^{111}_1 = A^3_3$

$H = D^2 = B^{22} = A^{222} = A^2_2^2$

$I = F^1 = B^{31} = A^{31}$

$J = F^2 = B^{32} = A^{32}$

## SPECIFYING RANGES

History traversing commands such as `git log` operate on a set of commits, not just a single commit. To these commands, specifying a single revision with the notation described in the previous section means the set of commits reachable from that commit, following the commit ancestry chain.

To exclude commits reachable from a commit, a prefix `^` notation is used. E.g. `^r1 r2` means commits reachable from `r2` but exclude the ones reachable from `r1`.

This set operation appears so often that there is a shorthand for it. When you have two commits `r1` and `r2` (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from `r2` excluding those that are reachable from `r1` by `^r1 r2` and it can be written as `r1..r2`.

A similar notation `r1...r2` is called symmetric difference of `r1` and `r2` and is defined as `r1 r2 --not (git merge-base --all r1 r2)`. It is the set of commits that are reachable from either one of `r1` or `r2` but not from both.

In these two shorthands, you can omit one end and let it default to HEAD. For example, `origin..`

is a shorthand for *origin..HEAD* and asks What did I do since I forked from the origin branch? Similarly, *..origin* is a shorthand for *HEAD..origin* and asks What did the origin do since I forked from them? Note that *..* would mean *HEAD..HEAD* which is an empty range that is both reachable and unreachable from HEAD.

Two other shorthands for naming a set that is formed by a commit and its parent commits exist. The *r1^@* notation means all parents of *r1*. *r1^!* includes commit *r1* but excludes all of its parents.

To summarize:

*<rev>*

Include commits that are reachable from (i.e. ancestors of) *<rev>*.

*^<rev>*

Exclude commits that are reachable from (i.e. ancestors of) *<rev>*.

*<rev1>..<rev2>*

Include commits that are reachable from *<rev2>* but exclude those that are reachable from *<rev1>*. When either *<rev1>* or *<rev2>* is omitted, it defaults to *HEAD*.

*<rev1>...<rev2>*

Include commits that are reachable from either *<rev1>* or *<rev2>* but exclude those that are reachable from both. When either *<rev1>* or *<rev2>* is omitted, it defaults to *HEAD*.

*<rev>^@*, e.g. *HEAD^@*

A suffix *^* followed by an at sign is the same as listing all parents of *<rev>* (meaning, include anything reachable from its parents, but not the commit itself).

*<rev>^!*, e.g. *HEAD^!*

A suffix *^* followed by an exclamation mark is the same as giving commit *<rev>* and then all its parents prefixed with *^* to exclude them (and their ancestors).

Here are a handful of examples:

```
D G H D
D F G H I J D F
^G D H D
^D B E I J F B
B..C C
B...C G H D E B C
^D B C E I J F B C
C I J F C
C^@ I J F
C^! C
F^! D G H D F
```

## PARSEOPT

In *--parseopt* mode, *git rev-parse* helps massaging options to bring to shell scripts the same facilities *C* builtins have. It works as an option normalizer (e.g. splits single switches aggregate values), a bit like [getopt\(1\)](#) does.

It takes on the standard input the specification of the options to parse and understand, and echoes on the standard output a string suitable for [sh\(1\)](#) *eval* to replace the arguments with normalized ones. In case of error, it outputs usage on the standard error stream, and exits with code 129.

Note: Make sure you quote the result when passing it to *eval*. See below for an example.

### Input Format

*git rev-parse --parseopt* input format is fully text based. It has two parts, separated by a line that contains only *--*. The lines before the separator (should be one or more) are used for the usage.

The lines after the separator describe the options.

Each line of options has this format:

```
<opt-spec><flags>*<arg-hint>? SP+ help LF
```

<opt-spec>

its format is the short option character, then the long option name separated by a comma. Both parts are not required, though at least one is necessary. h,help, dry-run and f are all three correct <opt-spec>.

<flags>

<flags> are of \*, =, ? or !.

- Use = if the option takes an argument.
- Use ? to mean that the option takes an optional argument. You probably want to use the --stuck-long mode to be able to unambiguously parse the optional argument.
- Use \* to mean that this option should not be listed in the usage generated for the -h argument. It's shown for --help-all as documented in [gitcli\(7\)](#).
- Use ! to not make the corresponding negated long option available.

<arg-hint>

<arg-hint>, if specified, is used as a name of the argument in the help output, for options that take arguments. <arg-hint> is terminated by the first whitespace. It is customary to use a dash to separate words in a multi-word argument hint.

The remainder of the line, after stripping the spaces, is used as the help associated to the option.

Blank lines are ignored, and lines that don't match this specification are used as option group headers (start the line with a space to create such lines on purpose).

### Example

```
OPTS_SPEC=
some-command [options] <args>...

some-command does foo and bar!
--
h,help show the help

foo some nifty option --foo
bar= some cool option --bar with an argument
baz=arg another cool option --baz with a named argument
qux?path qux may take a path argument but has meaning by itself

An option group Header
C? option C with an optional argument

eval $(echo $OPTS_SPEC | git rev-parse --parseopt -- $@ || echo exit $?)
```

### Usage text

When \$@ is -h or --help in the above example, the following usage text would be shown:

```
usage: some-command [options] <args>...

some-command does foo and bar!

-h, --help show the help
--foo some nifty option --foo
--bar ... some cool option --bar with an argument
--baz <arg> another cool option --baz with a named argument
--qux[=<path>] qux may take a path argument but has meaning by itself

An option group Header
-C[...] option C with an optional argument
```

## SQ-QUOTE

In `--sq-quote` mode, *git rev-parse* echoes on the standard output a single line suitable for `sh(1)` `eval`. This line is made by normalizing the arguments following `--sq-quote`. Nothing other than quoting the arguments is done.

If you want command input to still be interpreted as usual by *git rev-parse* before the output is shell quoted, see the `--sq` option.

### Example

```
$ cat >your-git-script.sh <<EOF
#!/bin/sh
args=$(git rev-parse --sq-quote $@) # quote user-supplied arguments
command=git frotz -n24 $args # and use it inside a handcrafted
# command line
eval $command
EOF

$ sh your-git-script.sh a bc
```

## EXAMPLES

- Print the object name of the current commit:  

```
$ git rev-parse --verify HEAD
```
- Print the commit object name from the revision in the `$REV` shell variable:  

```
$ git rev-parse --verify $REV^{commit}
```

This will error out if `$REV` is empty or not a valid revision.
- Similar to above:  

```
$ git rev-parse --default master --verify $REV
```

but if `$REV` is empty, the commit object name from master will be printed.

## GIT

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