



Linux Ubuntu 26.04 Manual Pages on command 'git.1'

\$ man git.1

```
GIT(1)                  Git Manual                  GIT(1)

NAME

    git - the stupid content tracker

SYNOPSIS

    git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]

        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]

    [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]

    [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]

    [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]

    <command> [<args>]
```

DESCRIPTION

Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals.

See gittutorial(7) to get started, then see giteveryday(7) for a useful minimum set of commands. The Git User's Manual[1] has a more in-depth introduction.

After you mastered the basic concepts, you can come back to this page to learn what commands Git offers. You can learn more about individual Git commands with "git help command".

gitcli(7) manual page gives you an overview of the command-line command syntax.

A formatted and hyperlinked copy of the latest Git documentation can be viewed at <https://git.github.io/html/docs/git.html> or <https://git-scm.com/docs>.

OPTIONS

-v, --version

Prints the Git suite version that the git program came from.

This option is internally converted to `git version ...` and accepts the same options as the `git-version(1)` command. If `--help` is also given, it takes precedence over `--version`.

`-h, --help`

Prints the synopsis and a list of the most commonly used commands. If the option `--all` or `-a` is given then all available commands are printed. If a Git command is named this option will bring up the manual page for that command.

Other options are available to control how the manual page is displayed. See `git-help(1)` for more information, because `git --help ...` is converted internally into `git help ...`

`-C <path>`

Run as if git was started in `<path>` instead of the current working directory. When multiple `-C` options are given, each subsequent non-absolute `-C <path>` is interpreted relative to the preceding `-C <path>`. If `<path>` is present but empty, e.g. `-C ""`, then the current working directory is left unchanged.

This option affects options that expect path name like `--git-dir` and `--work-tree` in that their interpretations of the path names would be made relative to the working directory caused by the `-C` option. For example the following invocations are equivalent:

```
git --git-dir=a.git --work-tree=b -C c status
```

```
git --git-dir=c/a.git --work-tree=c/b status
```

`-c <name>=<value>`

Pass a configuration parameter to the command. The value given will override values from configuration files. The `<name>` is expected in the same format as listed by `git config` (subkeys separated by dots).

Note that omitting the `=` in `git -c foo.bar ...` is allowed and sets `foo.bar` to the boolean true value (just like `[foo]bar` would in a config file). Including the equals but with an empty value (like `git -c foo.bar= ...`) sets `foo.bar` to the empty string which `git config --type=bool` will convert to false.

`--config-env=<name>=<envvar>`

Like `-c <name>=<value>`, give configuration variable `<name>` a value, where `<envvar>` is the name of an environment variable from which to retrieve the value. Unlike `-c` there is no shortcut for directly setting the value to an empty string, instead the environment variable itself must be set to the empty string. It is an error if the `<envvar>` does not exist in the environment. `<envvar>` may not contain an equals sign to avoid ambiguity with `<name>` containing one.

This is useful for cases where you want to pass transitory configuration options to git, but are doing so on operating systems where other processes might be able to read your command line (e.g. `/proc/self/cmdline`), but not your environment (e.g. `/proc/self/environ`). That behavior is the default on Linux, but may not be on your system.

Note that this might add security for variables such as `http.extraHeader` where the sensitive information is part of the value, but not e.g. `url.<base>.insteadOf` where the sensitive information can be part of the key.

`--exec-path[=<path>]`

Path to wherever your core Git programs are installed. This can also be controlled by setting the `GIT_EXEC_PATH` environment variable. If no path is given, git will print the current setting and then exit.

`--html-path`

Print the path, without trailing slash, where Git's HTML documentation is installed and exit.

`--man-path`

Print the manpath (see `man(1)`) for the man pages for this version of Git and exit.

`--info-path`

Print the path where the Info files documenting this version of Git are installed and exit.

`-p, --paginate`

Pipe all output into less (or if set, `$PAGER`) if standard output is a terminal. This overrides the `pager.<cmd>` configuration options (see the "Configuration Mechanism" section below).

`-P, --no-pager`

Do not pipe Git output into a pager.

`--git-dir=<path>`

Set the path to the repository ("`.git`" directory). This can also be controlled by setting the `GIT_DIR` environment variable. It can be an absolute path or relative path to current working directory.

Specifying the location of the "`.git`" directory using this option (or `GIT_DIR` environment variable) turns off the repository discovery that tries to find a directory with "`.git`" subdirectory (which is how the repository and the top-level of the working

tree are discovered), and tells Git that you are at the top level of the working tree.

If you are not at the top-level directory of the working tree, you should tell Git where the top-level of the working tree is, with the `--work-tree=<path>` option (or `GIT_WORK_TREE` environment variable)

If you just want to run git as if it was started in `<path>` then use `git -C <path>`.

`--work-tree=<path>`

Set the path to the working tree. It can be an absolute path or a path relative to the current working directory. This can also be controlled by setting the `GIT_WORK_TREE` environment variable and the `core.worktree` configuration variable (see `core.worktree` in `git-config(1)` for a more detailed discussion).

`--namespace=<path>`

Set the Git namespace. See `gitnamespaces(7)` for more details. Equivalent to setting the `GIT_NAMESPACE` environment variable.

`--bare`

Treat the repository as a bare repository. If `GIT_DIR` environment is not set, it is set to the current working directory.

`--no-replace-objects`

Do not use replacement refs to replace Git objects. This is equivalent to exporting the `GIT_NO_REPLACE_OBJECTS` environment variable with any value. See `git-replace(1)` for more information.

`--no-lazy-fetch`

Do not fetch missing objects from the promisor remote on demand. Useful together with `git cat-file -e <object>` to see if the object is locally available. This is equivalent to setting the `GIT_NO_LAZY_FETCH` environment variable to 1.

`--no-optional-locks`

Do not perform optional operations that require locks. This is equivalent to setting the `GIT_OPTIONAL_LOCKS` to 0.

`--no-advice`

Disable all advice hints from being printed.

`--literal-pathsspecs`

Treat pathsspecs literally (i.e. no globbing, no pathspec magic). This is equivalent to setting the `GIT_LITERAL_PATHSPECS` environment variable to 1.

`--glob-pathsspecs`

Add "glob" magic to all pathspec. This is equivalent to setting the GIT_GLOB_PATHSPECS environment variable to 1. Disabling globbing on individual pathspecs can be done using pathspec magic ":(literal)"

--noglob-pathspecs

Add "literal" magic to all pathspec. This is equivalent to setting the GIT_NOGLOB_PATHSPECS environment variable to 1. Enabling globbing on individual pathspecs can be done using pathspec magic ":(glob)"

--icase-pathspecs

Add "icase" magic to all pathspec. This is equivalent to setting the GIT_ICASE_PATHSPECS environment variable to 1.

--list-cmds=<group>[,<group>...]

List commands by group. This is an internal/experimental option and may change or be removed in the future. Supported groups are: builtins, parseopt (builtin commands that use parse-options), deprecated (deprecated builtins), main (all commands in libexec directory), others (all other commands in \$PATH that have git- prefix), list-<category> (see categories in command-list.txt), nohelpers (exclude helper commands), alias and config (retrieve command list from config variable completion.commands)

--attr-source=<tree-ish>

Read gitattributes from <tree-ish> instead of the worktree. See gitattributes(5). This is equivalent to setting the GIT_ATTR_SOURCE environment variable.

GIT COMMANDS

We divide Git into high level ("porcelain") commands and low level ("plumbing") commands.

HIGH-LEVEL COMMANDS (PORCELAIN)

We separate the porcelain commands into the main commands and some ancillary user utilities.

Main porcelain commands

git-add(1)

Add file contents to the index.

git-am(1)

Apply a series of patches from a mailbox.

git-archive(1)

Create an archive of files from a named tree.

git-backfill(1)

Download missing objects in a partial clone.

git-bisect(1)

Use binary search to find the commit that introduced a bug.

git-branch(1)

List, create, or delete branches.

git-bundle(1)

Move objects and refs by archive.

git-checkout(1)

Switch branches or restore working tree files.

git-cherry-pick(1)

Apply the changes introduced by some existing commits.

git-citool(1)

Graphical alternative to git-commit.

git-clean(1)

Remove untracked files from the working tree.

git-clone(1)

Clone a repository into a new directory.

git-commit(1)

Record changes to the repository.

git-describe(1)

Give an object a human readable name based on an available ref.

git-diff(1)

Show changes between commits, commit and working tree, etc.

git-fetch(1)

Download objects and refs from another repository.

git-format-patch(1)

Prepare patches for e-mail submission.

git-gc(1)

Cleanup unnecessary files and optimize the local repository.

git-grep(1)

Print lines matching a pattern.

git-gui(1)

A portable graphical interface to Git.

git-init(1)

Create an empty Git repository or reinitialize an existing one.

`git-log(1)`

Show commit logs.

`git-maintenance(1)`

Run tasks to optimize Git repository data.

`git-merge(1)`

Join two or more development histories together.

`git-mv(1)`

Move or rename a file, a directory, or a symlink.

`git-notes(1)`

Add or inspect object notes.

`git-pull(1)`

Fetch from and integrate with another repository or a local branch.

`git-push(1)`

Update remote refs along with associated objects.

`git-range-diff(1)`

Compare two commit ranges (e.g. two versions of a branch).

`git-rebase(1)`

Reapply commits on top of another base tip.

`git-reset(1)`

Set HEAD or the index to a known state.

`git-restore(1)`

Restore working tree files.

`git-revert(1)`

Revert some existing commits.

`git-rm(1)`

Remove files from the working tree and from the index.

`git-shortlog(1)`

Summarize git log output.

`git-show(1)`

Show various types of objects.

`git-sparse-checkout(1)`

Reduce your working tree to a subset of tracked files.

git-stash(1)

Stash the changes in a dirty working directory away.

git-status(1)

Show the working tree status.

git-submodule(1)

Initialize, update or inspect submodules.

git-switch(1)

Switch branches.

git-tag(1)

Create, list, delete or verify tags.

git-worktree(1)

Manage multiple working trees.

gitk(1)

The Git repository browser.

scalar(1)

A tool for managing large Git repositories.

Ancillary Commands

Manipulators:

git-config(1)

Get and set repository or global options.

git-fast-export(1)

Git data exporter.

git-fast-import(1)

Backend for fast Git data importers.

git-filter-branch(1)

Rewrite branches.

git-mergetool(1)

Run merge conflict resolution tools to resolve merge conflicts.

git-pack-refs(1)

Pack heads and tags for efficient repository access.

git-prune(1)

Prune all unreachable objects from the object database.

git-reflog(1)

Manage reflog information.

git-refs(1)

Low-level access to refs.

git-remote(1)

Manage set of tracked repositories.

git-repack(1)

Pack unpacked objects in a repository.

git-replace(1)

Create, list, delete refs to replace objects.

Interrogators:

git-annotate(1)

Annotate file lines with commit information.

git-blame(1)

Show what revision and author last modified each line of a file.

git-bugreport(1)

Collect information for user to file a bug report.

git-count-objects(1)

Count unpacked number of objects and their disk consumption.

git-diagnose(1)

Generate a zip archive of diagnostic information.

git-difftool(1)

Show changes using common diff tools.

git-fsck(1)

Verifies the connectivity and validity of the objects in the database.

git-help(1)

Display help information about Git.

git-instaweb(1)

Instantly browse your working repository in gitweb.

git-merge-tree(1)

Perform merge without touching index or working tree.

git-rerere(1)

Reuse recorded resolution of conflicted merges.

git-show-branch(1)

Show branches and their commits.

`git-verify-commit(1)`

Check the GPG signature of commits.

`git-verify-tag(1)`

Check the GPG signature of tags.

`git-version(1)`

Display version information about Git.

`git-whatchanged(1)`

Show logs with differences each commit introduces.

`gitweb(1)`

Git web interface (web frontend to Git repositories).

Interacting with Others

These commands are to interact with foreign SCM and with other people via patch over e-mail.

`git-archimport(1)`

Import a GNU Arch repository into Git.

`git-cvsexportcommit(1)`

Export a single commit to a CVS checkout.

`git-cvsimport(1)`

Salvage your data out of another SCM people love to hate.

`git-cvsserver(1)`

A CVS server emulator for Git.

`git-imap-send(1)`

Send a collection of patches from stdin to an IMAP folder.

`git-p4(1)`

Import from and submit to Perforce repositories.

`git-quiltimport(1)`

Applies a quilt patchset onto the current branch.

`git-request-pull(1)`

Generates a summary of pending changes.

`git-send-email(1)`

Send a collection of patches as emails.

`git-svn(1)`

Bidirectional operation between a Subversion repository and Git.

Reset, restore and revert

There are three commands with similar names: `git reset`, `git restore` and `git revert`.

? `git-revert(1)` is about making a new commit that reverts the changes made by other commits.

? `git-restore(1)` is about restoring files in the working tree from either the index or another commit. This command does not update your branch. The command can also be used to restore files in the index from another commit.

? `git-reset(1)` is about updating your branch, moving the tip in order to add or remove commits from the branch. This operation changes the commit history.

`git reset` can also be used to restore the index, overlapping with `git restore`.

LOW-LEVEL COMMANDS (PLUMBING)

Although Git includes its own porcelain layer, its low-level commands are sufficient to support development of alternative porcelains. Developers of such porcelains might start by reading about `git-update-index(1)` and `git-read-tree(1)`.

The interface (input, output, set of options and the semantics) to these low-level commands are meant to be a lot more stable than Porcelain level commands, because these commands are primarily for scripted use. The interface to Porcelain commands on the other hand are subject to change in order to improve the end user experience.

The following description divides the low-level commands into commands that manipulate objects (in the repository, index, and working tree), commands that interrogate and compare objects, and commands that move objects and references between repositories.

Manipulation commands

`git-apply(1)`

Apply a patch to files and/or to the index.

`git-checkout-index(1)`

Copy files from the index to the working tree.

`git-commit-graph(1)`

Write and verify Git commit-graph files.

`git-commit-tree(1)`

Create a new commit object.

`git-hash-object(1)`

Compute object ID and optionally create an object from a file.

`git-index-pack(1)`

Build pack index file for an existing packed archive.

git-merge-file(1)

Run a three-way file merge.

git-merge-index(1)

Run a merge for files needing merging.

git-mktag(1)

Creates a tag object with extra validation.

git-mktree(1)

Build a tree-object from ls-tree formatted text.

git-multi-pack-index(1)

Write and verify multi-pack-indexes.

git-pack-objects(1)

Create a packed archive of objects.

git-prune-packed(1)

Remove extra objects that are already in pack files.

git-read-tree(1)

Reads tree information into the index.

git-replay(1)

EXPERIMENTAL: Replay commits on a new base, works with bare repos too.

git-symbolic-ref(1)

Read, modify and delete symbolic refs.

git-unpack-objects(1)

Unpack objects from a packed archive.

git-update-index(1)

Register file contents in the working tree to the index.

git-update-ref(1)

Update the object name stored in a ref safely.

git-write-tree(1)

Create a tree object from the current index.

Interrogation commands

git-cat-file(1)

Provide contents or details of repository objects.

git-cherry(1)

Find commits yet to be applied to upstream.

`git-diff-files(1)`

Compares files in the working tree and the index.

`git-diff-index(1)`

Compare a tree to the working tree or index.

`git-diff-pairs(1)`

Compare the content and mode of provided blob pairs.

`git-diff-tree(1)`

Compares the content and mode of blobs found via two tree objects.

`git-for-each-ref(1)`

Output information on each ref.

`git-for-each-repo(1)`

Run a Git command on a list of repositories.

`git-get-tar-commit-id(1)`

Extract commit ID from an archive created using `git-archive`.

`git-last-modified(1)`

EXPERIMENTAL: Show when files were last modified.

`git-ls-files(1)`

Show information about files in the index and the working tree.

`git-ls-remote(1)`

List references in a remote repository.

`git-ls-tree(1)`

List the contents of a tree object.

`git-merge-base(1)`

Find as good common ancestors as possible for a merge.

`git-name-rev(1)`

Find symbolic names for given revs.

`git-pack-redundant(1)`

Find redundant pack files.

`git-repo(1)`

Retrieve information about the repository.

`git-rev-list(1)`

Lists commit objects in reverse chronological order.

git-rev-parse(1)

Pick out and massage parameters.

git-show-index(1)

Show packed archive index.

git-show-ref(1)

List references in a local repository.

git-unpack-file(1)

Creates a temporary file with a blob's contents.

git-var(1)

Show a Git logical variable.

git-verify-pack(1)

Validate packed Git archive files.

In general, the interrogate commands do not touch the files in the working tree.

Syncing repositories

git-daemon(1)

A really simple server for Git repositories.

git-fetch-pack(1)

Receive missing objects from another repository.

git-http-backend(1)

Server side implementation of Git over HTTP.

git-send-pack(1)

Push objects over Git protocol to another repository.

git-update-server-info(1)

Update auxiliary info file to help dumb servers.

The following are helper commands used by the above; end users typically do not use them directly.

git-http-fetch(1)

Download from a remote Git repository via HTTP.

git-http-push(1)

Push objects over HTTP/DAV to another repository.

git-receive-pack(1)

Receive what is pushed into the repository.

git-shell(1)

Restricted login shell for Git-only SSH access.

git-upload-archive(1)

Send archive back to git-archive.

git-upload-pack(1)

Send objects packed back to git-fetch-pack.

Internal helper commands

These are internal helper commands used by other commands; end users typically do not use them directly.

git-check-attr(1)

Display gitattributes information.

git-check-ignore(1)

Debug gitignore / exclude files.

git-check-mailmap(1)

Show canonical names and email addresses of contacts.

git-check-ref-format(1)

Ensures that a reference name is well formed.

git-column(1)

Display data in columns.

git-credential(1)

Retrieve and store user credentials.

git-credential-cache(1)

Helper to temporarily store passwords in memory.

git-credential-store(1)

Helper to store credentials on disk.

git-fmt-merge-msg(1)

Produce a merge commit message.

git-hook(1)

Run git hooks.

git-interpret-trailers(1)

Add or parse structured information in commit messages.

git-mailinfo(1)

Extracts patch and authorship from a single e-mail message.

git-mailsplit(1)

Simple UNIX mbox splitter program.

[git-merge-one-file\(1\)](#)

The standard helper program to use with git-merge-index.

[git-patch-id\(1\)](#)

Compute unique ID for a patch.

[git-sh-i18n\(1\)](#)

Git's i18n setup code for shell scripts.

[git-sh-setup\(1\)](#)

Common Git shell script setup code.

[git-stripspace\(1\)](#)

Remove unnecessary whitespace.

GUIDES

The following documentation pages are guides about Git concepts.

[gitcore-tutorial\(7\)](#)

A Git core tutorial for developers.

[gitcredentials\(7\)](#)

Providing usernames and passwords to Git.

[gitcvsmigration\(7\)](#)

Git for CVS users.

[gitdiffcore\(7\)](#)

Tweaking diff output.

[giteveryday\(7\)](#)

A useful minimum set of commands for Everyday Git.

[gitfaq\(7\)](#)

Frequently asked questions about using Git.

[gitglossary\(7\)](#)

A Git Glossary.

[gitnamespaces\(7\)](#)

Git namespaces.

[gitremote-helpers\(7\)](#)

Helper programs to interact with remote repositories.

[gitsubmodules\(7\)](#)

Mounting one repository inside another.

gittutorial(7)

A tutorial introduction to Git.

gittutorial-2(7)

A tutorial introduction to Git: part two.

gitworkflows(7)

An overview of recommended workflows with Git.

REPOSITORY, COMMAND AND FILE INTERFACES

This documentation discusses repository and command interfaces which users are expected to interact with directly. See --user-formats in git-help(1) for more details on the criteria.

gitattributes(5)

Defining attributes per path.

gitcli(7)

Git command-line interface and conventions.

githooks(5)

Hooks used by Git.

gitignore(5)

Specifies intentionally untracked files to ignore.

gitmailmap(5)

Map author/committer names and/or E-Mail addresses.

gitmodules(5)

Defining submodule properties.

gitrepository-layout(5)

Git Repository Layout.

gitrevisions(7)

Specifying revisions and ranges for Git.

FILE FORMATS, PROTOCOLS AND OTHER DEVELOPER INTERFACES

This documentation discusses file formats, over-the-wire protocols and other git developer interfaces. See --developer-interfaces in git-help(1).

gitformat-bundle(5)

The bundle file format.

gitformat-chunk(5)

Chunk-based file formats.

gitformat-commit-graph(5)

Git commit-graph format.

gitformat-index(5)

Git index format.

gitformat-pack(5)

Git pack format.

gitformat-signature(5)

Git cryptographic signature formats.

gitprotocol-capabilities(5)

Protocol v0 and v1 capabilities.

gitprotocol-common(5)

Things common to various protocols.

gitprotocol-http(5)

Git HTTP-based protocols.

gitprotocol-pack(5)

How packs are transferred over-the-wire.

gitprotocol-v2(5)

Git Wire Protocol, Version 2.

CONFIGURATION MECHANISM

Git uses a simple text format to store customizations that are per repository and are per user. Such a configuration file may look like this:

```
#  
# A '#' or ';' character indicates a comment.  
#  
; core variables  
[core]  
    ; Don't trust file modes  
    filemode = false  
; user identity  
[user]  
    name = "Junio C Hamano"  
    email = "gitster@pobox.com"
```

Various commands read from the configuration file and adjust their operation accordingly.

See git-config(1) for a list and more details about the configuration mechanism.

IDENTIFIER TERMINOLOGY

<object>

Indicates the object name for any type of object.

<blob>

Indicates a blob object name.

<tree>

Indicates a tree object name.

<commit>

Indicates a commit object name.

<tree-ish>

Indicates a tree, commit or tag object name. A command that takes a <tree-ish> argument ultimately wants to operate on a <tree> object but automatically dereferences <commit> and <tag> objects that point at a <tree>.

<commit-ish>

Indicates a commit or tag object name. A command that takes a <commit-ish> argument ultimately wants to operate on a <commit> object but automatically dereferences <tag> objects that point at a <commit>.

<type>

Indicates that an object type is required. Currently one of: blob, tree, commit, or tag.

<file>

Indicates a filename - almost always relative to the root of the tree structure
GIT_INDEX_FILE describes.

SYMBOLIC IDENTIFIERS

Any Git command accepting any <object> can also use the following symbolic notation:

HEAD

indicates the head of the current branch.

<tag>

a valid tag name (i.e. a refs/tags/<tag> reference).

<head>

a valid head name (i.e. a refs/heads/<head> reference).

For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in gitrevisions(7).

Please see the gitrepository-layout(5) document.

Read githooks(5) for more details about each hook.

Higher level SCMs may provide and manage additional information in the \$GIT_DIR.

TERMINOLOGY

Please see gitglossary(7).

ENVIRONMENT VARIABLES

Various Git commands pay attention to environment variables and change their behavior. The environment variables marked as "Boolean" take their values the same way as Boolean valued configuration variables, i.e., "true", "yes", "on" and positive numbers are taken as "yes", while "false", "no", "off", and "0" are taken as "no".

Here are the variables:

System

HOME

Specifies the path to the user's home directory. On Windows, if unset, Git will set a process environment variable equal to: \$HOMEDRIVE\$HOMEPATH if both \$HOMEDRIVE and \$HOMEPATH exist; otherwise \$USERPROFILE if \$USERPROFILE exists.

The Git Repository

These environment variables apply to all core Git commands. Nb: it is worth noting that they may be used/overridden by SCMS sitting above Git so take care if using a foreign front-end.

GIT_INDEX_FILE

This environment variable specifies an alternate index file. If not specified, the default of \$GIT_DIR/index is used.

GIT_INDEX_VERSION

This environment variable specifies what index version is used when writing the index file out. It won't affect existing index files. By default index file version 2 or 3 is used. See git-update-index(1) for more information.

GIT_OBJECT_DIRECTORY

If the object storage directory is specified via this environment variable then the sha1 directories are created underneath - otherwise the default \$GIT_DIR/objects directory is used.

GIT_ALTERNATE_OBJECT_DIRECTORIES

Due to the immutable nature of Git objects, old objects can be archived into shared, read-only directories. This variable specifies a ":" separated (on Windows ";")

separated) list of Git object directories which can be used to search for Git objects.

New objects will not be written to these directories.

Entries that begin with " (double-quote) will be interpreted as C-style quoted paths, removing leading and trailing double-quotes and respecting backslash escapes. E.g., the value "path-with-\"-and-:-in-it":vanilla-path has two paths: path-with-\"-and-:-in-it and vanilla-path.

GIT_DIR

If the GIT_DIR environment variable is set then it specifies a path to use instead of the default .git for the base of the repository. The --git-dir command-line option also sets this value.

GIT_WORK_TREE

Set the path to the root of the working tree. This can also be controlled by the --work-tree command-line option and the core.worktree configuration variable.

GIT_NAMESPACE

Set the Git namespace; see gitnamespaces(7) for details. The --namespace command-line option also sets this value.

GIT_CEILING_DIRECTORIES

This should be a colon-separated list of absolute paths. If set, it is a list of directories that Git should not chdir up into while looking for a repository directory (useful for excluding slow-loading network directories). It will not exclude the current working directory or a GIT_DIR set on the command line or in the environment. Normally, Git has to read the entries in this list and resolve any symlink that might be present in order to compare them with the current directory. However, if even this access is slow, you can add an empty entry to the list to tell Git that the subsequent entries are not symlinks and needn't be resolved; e.g.,

GIT_CEILING_DIRECTORIES=/maybe/symlink::/very/slow/non/symlink.

GIT_DISCOVERY_ACROSS_FILESYSTEM

When run in a directory that does not have ".git" repository directory, Git tries to find such a directory in the parent directories to find the top of the working tree, but by default it does not cross filesystem boundaries. This Boolean environment variable can be set to true to tell Git not to stop at filesystem boundaries. Like GIT_CEILING_DIRECTORIES, this will not affect an explicit repository directory set via GIT_DIR or on the command line.

GIT_COMMON_DIR

If this variable is set to a path, non-worktree files that are normally in \$GIT_DIR will be taken from this path instead. Worktree-specific files such as HEAD or index are taken from \$GIT_DIR. See gitrepository-layout(5) and git-worktree(1) for details. This variable has lower precedence than other path variables such as GIT_INDEX_FILE, GIT_OBJECT_DIRECTORY...

GIT_DEFAULT_HASH

If this variable is set, the default hash algorithm for new repositories will be set to this value. This value is ignored when cloning and the setting of the remote repository is always used. The default is "sha1". See --object-format in git-init(1).

GIT_DEFAULT_REF_FORMAT

If this variable is set, the default reference backend format for new repositories will be set to this value. The default is "files". See --ref-format in git-init(1).

Git Commits

GIT_AUTHOR_NAME

The human-readable name used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the user.name and author.name configuration settings.

GIT_AUTHOR_EMAIL

The email address used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the user.email and author.email configuration settings.

GIT_AUTHOR_DATE

The date used for the author identity when creating commit or tag objects, or when writing reflogs. See git-commit(1) for valid formats.

GIT_COMMITTER_NAME

The human-readable name used in the committer identity when creating commit or tag objects, or when writing reflogs. Overrides the user.name and committer.name configuration settings.

GIT_COMMITTER_EMAIL

The email address used in the author identity when creating commit or tag objects, or when writing reflogs. Overrides the user.email and committer.email configuration settings.

GIT_COMMITTER_DATE

The date used for the committer identity when creating commit or tag objects, or when

writing reflogs. See `git-commit(1)` for valid formats.

EMAIL

The email address used in the author and committer identities if no other relevant environment variable or configuration setting has been set.

Git Diffs

GIT_DIFF_OPTS

Only valid setting is "--unified=??" or "-u??" to set the number of context lines shown when a unified diff is created. This takes precedence over any "-U" or "--unified" option value passed on the Git diff command line.

GIT_EXTERNAL_DIFF

When the environment variable `GIT_EXTERNAL_DIFF` is set, the program named by it is called to generate diffs, and Git does not use its builtin diff machinery. For a path that is added, removed, or modified, `GIT_EXTERNAL_DIFF` is called with 7 parameters:

path old-file old-hex old-mode new-file new-hex new-mode

where:

<old|new>-file

are files `GIT_EXTERNAL_DIFF` can use to read the contents of <old|new>,

<old|new>-hex

are the 40-hexdigit SHA-1 hashes,

<old|new>-mode

are the octal representation of the file modes.

The file parameters can point at the user's working file (e.g. new-file in "git-diff-files"), /dev/null (e.g. old-file when a new file is added), or a temporary file (e.g. old-file in the index). `GIT_EXTERNAL_DIFF` should not worry about unlinking the temporary file ? it is removed when `GIT_EXTERNAL_DIFF` exits.

For a path that is unmerged, `GIT_EXTERNAL_DIFF` is called with 1 parameter, <path>.

For each path `GIT_EXTERNAL_DIFF` is called, two environment variables,

`GIT_DIFF_PATH_COUNTER` and `GIT_DIFF_PATH_TOTAL` are set.

GIT_EXTERNAL_DIFF_TRUST_EXIT_CODE

If this Boolean environment variable is set to true then the `GIT_EXTERNAL_DIFF` command is expected to return exit code 0 if it considers the input files to be equal or 1 if it considers them to be different, like `diff(1)`. If it is set to false, which is the default, then the command is expected to return exit code 0 regardless of equality. Any

other exit code causes Git to report a fatal error.

`GIT_DIFF_PATH_COUNTER`

A 1-based counter incremented by one for every path.

`GIT_DIFF_PATH_TOTAL`

The total number of paths.

other

`GIT_MERGE_VERBOSITY`

A number controlling the amount of output shown by the recursive merge strategy.

Overrides `merge.verbosity`. See `git-merge(1)`

`GIT_PAGER`

This environment variable overrides `$PAGER`. If it is set to an empty string or to the value "cat", Git will not launch a pager. See also the `core.pager` option in `git-config(1)`.

`GIT_PROGRESS_DELAY`

A number controlling how many seconds to delay before showing optional progress indicators. Defaults to 1.

`GIT_EDITOR`

This environment variable overrides `$EDITOR` and `$VISUAL`. It is used by several Git commands when, on interactive mode, an editor is to be launched. See also `git-var(1)` and the `core.editor` option in `git-config(1)`.

`GIT_SEQUENCE_EDITOR`

This environment variable overrides the configured Git editor when editing the todo list of an interactive rebase. See also `git-rebase(1)` and the `sequence.editor` option in `git-config(1)`.

`GIT_SSH`, `GIT_SSH_COMMAND`

If either of these environment variables is set then `git fetch` and `git push` will use the specified command instead of `ssh` when they need to connect to a remote system. The command-line parameters passed to the configured command are determined by the `ssh` variant. See `ssh.variant` option in `git-config(1)` for details.

`$GIT_SSH_COMMAND` takes precedence over `$GIT_SSH`, and is interpreted by the shell, which allows additional arguments to be included. `$GIT_SSH` on the other hand must be just the path to a program (which can be a wrapper shell script, if additional arguments are needed).

Usually it is easier to configure any desired options through your personal `.ssh/config` file. Please consult your ssh documentation for further details.

`GIT_SSH_VARIANT`

If this environment variable is set, it overrides Git's autodetection whether `GIT_SSH/GIT_SSH_COMMAND/core.sshCommand` refer to OpenSSH, plink or tortoiseplink. This variable overrides the config setting `ssh.variant` that serves the same purpose.

`GIT_SSL_NO_VERIFY`

Setting and exporting this environment variable to any value tells Git not to verify the SSL certificate when fetching or pushing over HTTPS.

`GIT_ATTR_SOURCE`

Sets the treeish that gitattributes will be read from.

`GIT_ASKPASS`

If this environment variable is set, then Git commands which need to acquire passwords or passphrases (e.g. for HTTP or IMAP authentication) will call this program with a suitable prompt as command-line argument and read the password from its STDOUT. See also the `core.askPass` option in `git-config(1)`.

`GIT_TERMINAL_PROMPT`

If this Boolean environment variable is set to false, git will not prompt on the terminal (e.g., when asking for HTTP authentication).

`GIT_CONFIG_GLOBAL`, `GIT_CONFIG_SYSTEM`

Take the configuration from the given files instead from global or system-level configuration files. If `GIT_CONFIG_SYSTEM` is set, the system config file defined at build time (usually `/etc/gitconfig`) will not be read. Likewise, if `GIT_CONFIG_GLOBAL` is set, neither `$HOME/.gitconfig` nor `$XDG_CONFIG_HOME/git/config` will be read. Can be set to `/dev/null` to skip reading configuration files of the respective level.

`GIT_CONFIG_NOSYSTEM`

Whether to skip reading settings from the system-wide `$(prefix)/etc/gitconfig` file. This Boolean environment variable can be used along with `$HOME` and `$XDG_CONFIG_HOME` to create a predictable environment for a picky script, or you can set it to true to temporarily avoid using a buggy `/etc/gitconfig` file while waiting for someone with sufficient permissions to fix it.

`GIT_FLUSH`

If this Boolean environment variable is set to true, then commands such as `git blame` (in

incremental mode), `git rev-list`, `git log`, `git check-attr` and `git check-ignore` will force a flush of the output stream after each record have been flushed. If this variable is set to `false`, the output of these commands will be done using completely buffered I/O. If this environment variable is not set, Git will choose buffered or record-oriented flushing based on whether `stdout` appears to be redirected to a file or not.

`GIT_TRACE`

Enables general trace messages, e.g. alias expansion, built-in command execution and external command execution.

If this variable is set to `"1"`, `"2"` or `"true"` (comparison is case insensitive), trace messages will be printed to `stderr`.

If the variable is set to an integer value greater than 2 and lower than 10 (strictly) then Git will interpret this value as an open file descriptor and will try to write the trace messages into this file descriptor.

Alternatively, if the variable is set to an absolute path (starting with a `/` character), Git will interpret this as a file path and will try to append the trace messages to it.

Unsetting the variable, or setting it to empty, `"0"` or `"false"` (case insensitive) disables trace messages.

`GIT_TRACE_FSMONITOR`

Enables trace messages for the filesystem monitor extension. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_PACK_ACCESS`

Enables trace messages for all accesses to any packs. For each access, the pack file name and an offset in the pack is recorded. This may be helpful for troubleshooting some pack-related performance problems. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_PACKET`

Enables trace messages for all packets coming in or out of a given program. This can help with debugging object negotiation or other protocol issues. Tracing is turned off at a packet starting with `"PACK"` (but see `GIT_TRACE_PACKFILE` below). See `GIT_TRACE` for available trace output options.

`GIT_TRACE_PACKFILE`

Enables tracing of packfiles sent or received by a given program. Unlike other trace output, this trace is verbatim: no headers, and no quoting of binary data. You almost certainly want to direct into a file (e.g., `GIT_TRACE_PACKFILE=/tmp/my.pack`) rather than

displaying it on the terminal or mixing it with other trace output.

Note that this is currently only implemented for the client side of clones and fetches.

`GIT_TRACE_PERFORMANCE`

Enables performance related trace messages, e.g. total execution time of each Git command. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_REFS`

Enables trace messages for operations on the ref database. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_SETUP`

Enables trace messages printing the `.git`, working tree and current working directory after Git has completed its setup phase. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_SHALLOW`

Enables trace messages that can help debugging fetching / cloning of shallow repositories. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_CURL`

Enables a curl full trace dump of all incoming and outgoing data, including descriptive information, of the git transport protocol. This is similar to doing `curl --trace-ascii` on the command line. See `GIT_TRACE` for available trace output options.

`GIT_TRACE_CURL_NO_DATA`

When a curl trace is enabled (see `GIT_TRACE_CURL` above), do not dump data (that is, only dump info lines and headers).

`GIT_TRACE2`

Enables more detailed trace messages from the "trace2" library. Output from `GIT_TRACE2` is a simple text-based format for human readability.

If this variable is set to "1", "2" or "true" (comparison is case insensitive), trace messages will be printed to stderr.

If the variable is set to an integer value greater than 2 and lower than 10 (strictly) then Git will interpret this value as an open file descriptor and will try to write the trace messages into this file descriptor.

Alternatively, if the variable is set to an absolute path (starting with a / character), Git will interpret this as a file path and will try to append the trace messages to it.

If the path already exists and is a directory, the trace messages will be written to

files (one per process) in that directory, named according to the last component of the SID and an optional counter (to avoid filename collisions).

In addition, if the variable is set to `af_unix:[<socket-type>:]<absolute-pathname>`, Git will try to open the path as a Unix Domain Socket. The socket type can be either stream or dgram.

Unsetting the variable, or setting it to empty, "0" or "false" (case insensitive) disables trace messages.

See [Trace2 documentation\[2\]](#) for full details.

`GIT_TRACE2_EVENT`

This setting writes a JSON-based format that is suited for machine interpretation. See `GIT_TRACE2` for available trace output options and [Trace2 documentation\[2\]](#) for full details.

`GIT_TRACE2_PERF`

In addition to the text-based messages available in `GIT_TRACE2`, this setting writes a column-based format for understanding nesting regions. See `GIT_TRACE2` for available trace output options and [Trace2 documentation\[2\]](#) for full details.

`GIT_TRACE_REDACT`

By default, when tracing is activated, Git redacts the values of cookies, the "Authorization:" header, the "Proxy-Authorization:" header and packfile URIs. Set this Boolean environment variable to false to prevent this redaction.

`GIT_NO_REPLACE_OBJECTS`

Setting and exporting this environment variable tells Git to ignore replacement refs and do not replace Git objects.

`GIT_LITERAL_PATHSPECS`

Setting this Boolean environment variable to true will cause Git to treat all pathspecs literally, rather than as glob patterns. For example, running `GIT_LITERAL_PATHSPECS=1 git log -- '*.c'` will search for commits that touch the path *.c, not any paths that the glob *.c matches. You might want this if you are feeding literal paths to Git (e.g., paths previously given to you by `git ls-tree`, `--raw` diff output, etc).

`GIT_GLOB_PATHSPECS`

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as glob patterns (aka "glob" magic).

`GIT_NOGLOB_PATHSPECS`

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as literal (aka "literal" magic).

`GIT_ICASE_PATHSPECS`

Setting this Boolean environment variable to true will cause Git to treat all pathspecs as case-insensitive.

`GIT_NO_LAZY_FETCH`

Setting this Boolean environment variable to true tells Git not to lazily fetch missing objects from the promisor remote on demand.

`GIT_REFLOG_ACTION`

When a ref is updated, reflog entries are created to keep track of the reason why the ref was updated (which is typically the name of the high-level command that updated the ref), in addition to the old and new values of the ref. A scripted Porcelain command can use `set_reflog_action` helper function in `git-sh-setup` to set its name to this variable when it is invoked as the top level command by the end user, to be recorded in the body of the reflog.

`GIT_REF_PARANOIA`

If this Boolean environment variable is set to false, ignore broken or badly named refs when iterating over lists of refs. Normally Git will try to include any such refs, which may cause some operations to fail. This is usually preferable, as potentially destructive operations (e.g., `git-prune(1)`) are better off aborting rather than ignoring broken refs (and thus considering the history they point to as not worth saving). The default value is 1 (i.e., be paranoid about detecting and aborting all operations). You should not normally need to set this to 0, but it may be useful when trying to salvage data from a corrupted repository.

`GIT_COMMIT_GRAPH_PARANOIA`

When loading a commit object from the commit-graph, Git performs an existence check on the object in the object database. This is done to avoid issues with stale commit-graphs that contain references to already-deleted commits, but comes with a performance penalty.

The default is "false", which disables the aforementioned behavior. Setting this to "true" enables the existence check so that stale commits will never be returned from the commit-graph at the cost of performance.

`GIT_ALLOW_PROTOCOL`

If set to a colon-separated list of protocols, behave as if `protocol.allow` is set to `never`, and each of the listed protocols has `protocol.<name>.allow` set to `always` (overriding any existing configuration). See the description of `protocol.allow` in `git-config(1)` for more details.

GIT_PROTOCOL_FROM_USER

Set this Boolean environment variable to `false` to prevent protocols used by `fetch/push/clone` which are configured to the user state. This is useful to restrict recursive submodule initialization from an untrusted repository or for programs which feed potentially-untrusted URLs to `git` commands. See `git-config(1)` for more details.

GIT_PROTOCOL

For internal use only. Used in handshaking the wire protocol. Contains a colon : separated list of keys with optional values `<key>[=<value>]`. Presence of unknown keys and values must be ignored.

Note that servers may need to be configured to allow this variable to pass over some transports. It will be propagated automatically when accessing local repositories (i.e., `file://` or a filesystem path), as well as over the `git://` protocol. For `git-over-http`, it should work automatically in most configurations, but see the discussion in `git-http-backend(1)`. For `git-over-ssh`, the `ssh` server may need to be configured to allow clients to pass this variable (e.g., by using `AcceptEnv GIT_PROTOCOL` with `OpenSSH`).

This configuration is optional. If the variable is not propagated, then clients will fall back to the original "v0" protocol (but may miss out on some performance improvements or features). This variable currently only affects clones and fetches; it is not yet used for pushes (but may be in the future).

GIT_OPTIONAL_LOCKS

If this Boolean environment variable is set to `false`, `Git` will complete any requested operation without performing any optional sub-operations that require taking a lock. For example, this will prevent `git status` from refreshing the index as a side effect. This is useful for processes running in the background which do not want to cause lock contention with other operations on the repository. Defaults to `1`.

GIT_REDIRECT_STDIN, GIT_REDIRECT_STDOUT, GIT_REDIRECT_STDERR

Windows-only: allow redirecting the standard input/output/error handles to paths specified by the environment variables. This is particularly useful in multi-threaded applications where the canonical way to pass standard handles via `CreateProcess()` is not

an option because it would require the handles to be marked inheritable (and consequently every spawned process would inherit them, possibly blocking regular Git operations). The primary intended use case is to use named pipes for communication (e.g. `\\.\pipe\my-git-stdin-123`).

Two special values are supported: `off` will simply close the corresponding standard handle, and if `GIT_REDIRECT_STDERR` is `2>&1`, standard error will be redirected to the same handle as standard output.

`GIT_PRINT_SHA1_ELLIPSIS` (deprecated)

If set to `yes`, print an ellipsis following an (abbreviated) SHA-1 value. This affects indications of detached HEADs (`git-checkout(1)`) and the raw diff output (`git-diff(1)`).

Printing an ellipsis in the cases mentioned is no longer considered adequate and support for it is likely to be removed in the foreseeable future (along with the variable).

`GIT_ADVICE`

If set to `0`, then disable all advice messages. These messages are intended to provide hints to human users that may help them get out of problematic situations or take advantage of new features. Users can disable individual messages using the `advice.*` config keys. These messages may be disruptive to tools that execute Git processes, so this variable is available to disable the messages. (The `--no-advice` global option is also available, but old Git versions may fail when this option is not understood. The environment variable will be ignored by Git versions that do not understand it.)

DISCUSSION

More detail on the following is available from the Git concepts chapter of the [user-manual\[3\]](#) and [gitcore-tutorial\(7\)](#).

A Git project normally consists of a working directory with a `.git` subdirectory at the top level. The `.git` directory contains, among other things, a compressed object database representing the complete history of the project, an `"index"` file which links that history to the current contents of the working tree, and named pointers into that history such as tags and branch heads.

The object database contains objects of three main types: blobs, which hold file data; trees, which point to blobs and other trees to build up directory hierarchies; and commits, which each reference a single tree and some number of parent commits.

The commit, equivalent to what other systems call a "changeset" or "version", represents a step in the project's history, and each parent represents an immediately preceding step.

Commits with more than one parent represent merges of independent lines of development. All objects are named by the SHA-1 hash of their contents, normally written as a string of 40 hex digits. Such names are globally unique. The entire history leading up to a commit can be vouched for by signing just that commit. A fourth object type, the tag, is provided for this purpose.

When first created, objects are stored in individual files, but for efficiency may later be compressed together into "pack files".

Named pointers called refs mark interesting points in history. A ref may contain the SHA-1 name of an object or the name of another ref (the latter is called a "symbolic ref"). Refs with names beginning refs/head/ contain the SHA-1 name of the most recent commit (or "head") of a branch under development. SHA-1 names of tags of interest are stored under refs/tags/.

A symbolic ref named HEAD contains the name of the currently checked-out branch.

The index file is initialized with a list of all paths and, for each path, a blob object and a set of attributes. The blob object represents the contents of the file as of the head of the current branch. The attributes (last modified time, size, etc.) are taken from the corresponding file in the working tree. Subsequent changes to the working tree can be found by comparing these attributes. The index may be updated with new content, and new commits may be created from the content stored in the index.

The index is also capable of storing multiple entries (called "stages") for a given pathname. These stages are used to hold the various unmerged version of a file when a merge is in progress.

SECURITY

Some configuration options and hook files may cause Git to run arbitrary shell commands.

Because configuration and hooks are not copied using git clone, it is generally safe to clone remote repositories with untrusted content, inspect them with git log, and so on.

However, it is not safe to run Git commands in a .git directory (or the working tree that surrounds it) when that .git directory itself comes from an untrusted source. The commands in its config and hooks are executed in the usual way.

By default, Git will refuse to run when the repository is owned by someone other than the user running the command. See the entry for safe.directory in git-config(1). While this can help protect you in a multi-user environment, note that you can also acquire untrusted repositories that are owned by you (for example, if you extract a zip file or tarball from an untrusted source). In such cases, you'd need to "sanitize" the untrusted repository

first.

If you have an untrusted .git directory, you should first clone it with `git clone --no-local` to obtain a clean copy. Git does restrict the set of options and hooks that will be run by `upload-pack`, which handles the server side of a clone or fetch, but beware that the surface area for attack against `upload-pack` is large, so this does carry some risk. The safest thing is to serve the repository as an unprivileged user (either via `git-daemon(1)`, `ssh`, or using other tools to change user ids). See the discussion in the SECURITY section of `git-upload-pack(1)`.

FURTHER DOCUMENTATION

See the references in the "description" section to get started using Git. The following is probably more detail than necessary for a first-time user.

The Git concepts chapter of the `user-manual[3]` and `gitcore-tutorial(7)` both provide introductions to the underlying Git architecture.

See `gitworkflows(7)` for an overview of recommended workflows.

See also the `howto[4]` documents for some useful examples.

The internals are documented in the Git API documentation[5].

Users migrating from CVS may also want to read `gitcv-migration(7)`.

AUTHORS

Git was started by Linus Torvalds, and is currently maintained by Junio C Hamano. Numerous contributions have come from the Git mailing list `<git@vger.kernel.org[6]>`.

<https://openhub.net/p/git/contributors/summary> gives you a more complete list of contributors.

If you have a clone of `git.git` itself, the output of `git-shortlog(1)` and `git-blame(1)` can show you the authors for specific parts of the project.

REPORTING BUGS

Report bugs to the Git mailing list `<git@vger.kernel.org[6]>` where the development and maintenance is primarily done. You do not have to be subscribed to the list to send a message there. See the list archive at <https://lore.kernel.org/git> for previous bug reports and other discussions.

Issues which are security relevant should be disclosed privately to the Git Security mailing list `<git-security@googlegroups.com[7]>`.

SEE ALSO

`gittutorial(7)`, `gittutorial-2(7)`, `giteveryday(7)`, `gitcv-migration(7)`, `gitglossary(7)`,

gitcore-tutorial(7), gitcli(7), The Git User's Manual[1], gitworkflows(7)

GIT

Part of the git(1) suite

NOTES

1. Git User's Manual

<file:///usr/share/doc/git/html/user-manual.html>

2. Trace2 documentation

<file:///usr/share/doc/git/html/technical/api-trace2.html>

3. Git concepts chapter of the user-manual

<file:///usr/share/doc/git/html/user-manual.html#git-concepts>

4. howto

<file:///usr/share/doc/git/html/howto-index.html>

5. Git API documentation

<file:///usr/share/doc/git/html/technical/api-index.html>

6. git@vger.kernel.org

<mailto:git@vger.kernel.org>

7. git-security@googlegroups.com

<mailto:git-security@googlegroups.com>

Git 2.53.0

03/02/2026

GIT(1)