git(1) Manual Page

NAME

git - the stupid content tracker

SYNOPSIS

git [--version] [--exec-path[=GIT_EXEC_PATH]] [--html-path]
[-p|--paginate|--no-pager] [--no-replace-objects]
[--bare] [--git-dir=GIT_DIR] [--work-tree=GIT_WORK_TREE]
[--help] 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 Everyday Git for a useful minimum set of commands, and "man git-commandname" for documentation of each command. CVS users may also want to read gitcvs-migration(7). See the Git User's Manual for a more in-depth introduction.

The COMMAND is either a name of a Git command (see below) or an alias as defined in the configuration file (see git-config(1)).

Formatted and hyperlinked version of the latest git documentation can be viewed at `http://www.kernel.org/pub/software/scm/git/docs/`.

Note

You are reading the documentation for the latest (possibly unreleased) version of git, that is available from *master* branch of the `git.git` repository. Documentation for older releases are available here:

- documentation for release 1.6.5.2
- release notes for 1.6.5.2, 1.6.5.1, 1.6.5.
- documentation for release 1.6.4.4
- release notes for 1.6.4.4, 1.6.4.3, 1.6.4.2, 1.6.4.1, 1.6.4.
- documentation for release 1.6.3.4
- release notes for 1.6.3.4, 1.6.3.3, 1.6.3.2, 1.6.3.1, 1.6.3.
- release notes for 1.6.2.5, 1.6.2.4, 1.6.2.3, 1.6.2.2, 1.6.2.1, 1.6.2.

- documentation for release 1.6.1.3
- release notes for 1.6.1.3, 1.6.1.2, 1.6.1.1, 1.6.1.
- documentation for release 1.6.0.6
- release notes for 1.6.0.6, 1.6.0.5, 1.6.0.4, 1.6.0.3, 1.6.0.2, 1.6.0.1, 1.6.0.
- documentation for release 1.5.6.6
- release notes for 1.5.6.6, 1.5.6.5, 1.5.6.4, 1.5.6.3, 1.5.6.2, 1.5.6.1, 1.5.6.
- documentation for release 1.5.5.6
- release notes for 1.5.5.6, 1.5.5.5, 1.5.5.4, 1.5.5.3, 1.5.5.2, 1.5.5.1, 1.5.5.
- documentation for release 1.5.4.7
- release notes for 1.5.4.7, 1.5.4.6, 1.5.4.5, 1.5.4.4, 1.5.4.3, 1.5.4.2, 1.5.4.1, 1.5.4.
- documentation for release 1.5.3.8
- release notes for 1.5.3.8, 1.5.3.7, 1.5.3.6, 1.5.3.5, 1.5.3.4, 1.5.3.3, 1.5.3.2, 1.5.3.1, 1.5.3.
- documentation for release 1.5.2.5
- release notes for 1.5.2.5, 1.5.2.4, 1.5.2.3, 1.5.2.2, 1.5.2.1, 1.5.2.
- documentation for release 1.5.1.6
- release notes for 1.5.1.6, 1.5.1.5, 1.5.1.4, 1.5.1.3, 1.5.1.2, 1.5.1.1, 1.5.1.
- documentation for release 1.5.0.7
- release notes for 1.5.0.7, 1.5.0.6, 1.5.0.5, 1.5.0.3, 1.5.0.2, 1.5.0.1, 1.5.0.
- documentation for release 1.4.4.4, 1.3.3, 1.2.6, 1.0.13.

OPTIONS

--version

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

--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 …`.

--exec-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 to wherever your git HTML documentation is installed and exit.

-p

--paginate
: Pipe all output into *less* (or if set, $PAGER).

--no-pager
: Do not pipe git output into a pager.

--git-dir=<path>
: Set the path to the repository. 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.

--work-tree=<path>
: Set the path to the working tree. The value will not be used in combination with repositories found automatically in a .git directory (i.e. $GIT_DIR is not set). This can also be controlled by setting the GIT_WORK_TREE environment variable and the core.worktree configuration variable. It can be an absolute path or relative path to the directory specified by --git-dir or GIT_DIR. Note: If --git-dir or GIT_DIR are specified but none of --work-tree, GIT_WORK_TREE and core.worktree is specified, the current working directory is regarded as the top directory of your working tree.

--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. See git-replace(1) for more information.

FURTHER DOCUMENTATION

See the references above 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 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 documents for some useful examples.

The internals are documented in the GIT API documentation.

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-bisect(1)
: Find by binary search the change that introduced a bug.

git-branch(1)
: List, create, or delete branches.

git-bundle(1)
: Move objects and refs by archive.

git-checkout(1)
: Checkout a branch or paths to the working tree.

git-cherry-pick(1)
: Apply the change introduced by an existing commit.

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)
: Show the most recent tag that is reachable from a commit.

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-merge(1)
: Join two or more development histories together.

git-mv(1)
: Move or rename a file, a directory, or a symlink.

git-pull(1)
: Fetch from and merge with another repository or a local branch.

git-push(1)
: Update remote refs along with associated objects.

git-rebase(1)
: Forward-port local commits to the updated upstream head.

git-reset(1)
: Reset current HEAD to the specified state.

git-revert(1)
: Revert an existing commit.

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-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-tag(1)
: Create, list, delete or verify a tag object signed with GPG.

gitk(1)
: The git repository browser.

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-lost-found(1)
: (deprecated) Recover lost refs that luckily have not yet been pruned.

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-relink(1)
: Hardlink common objects in local repositories.

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.

git-repo-config(1)
: (deprecated) Get and set repository or global options.

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-cherry(1)
: Find commits not merged upstream.

git-count-objects(1)
: Count unpacked number of objects and their disk consumption.

git-difftool(1)
: Show changes using common diff tools.

git-fsck(1)
: Verifies the connectivity and validity of the objects in the database.

git-get-tar-commit-id(1)
: Extract commit ID from an archive created using git-archive.

git-help(1)
: display help information about git.

git-instaweb(1)
: Instantly browse your working repository in gitweb.

git-merge-tree(1)
: Show three-way merge without touching index.

git-rerere(1)
: Reuse recorded resolution of conflicted merges.

git-rev-parse(1)
: Pick out and massage parameters.

git-show-branch(1)
: Show branches and their commits.

git-verify-tag(1)
: Check the GPG signature of tags.

git-whatchanged(1)
: Show logs with difference each commit introduces.

Interacting with Others

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

git-archimport(1)
: Import an 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-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.

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 on a git index file and/or a working tree.

git-checkout-index(1)
: Copy files from the index to the working tree.

git-commit-tree(1)
: Create a new commit object.

git-hash-object(1)
: Compute object ID and optionally creates a blob 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.

git-mktree(1)
: Build a tree-object from ls-tree formatted text.

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-symbolic-ref(1)
Read and modify 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 content or type and size information for repository objects.

git-diff-files(1)
Compares files in the working tree and the index.

git-diff-index(1)
Compares content and mode of blobs between the index and repository.

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-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-rev-list(1)
Lists commit objects in reverse chronological order.

git-show-index(1)
Show packed archive index.

git-show-ref(1)
: List references in a local repository.

git-tar-tree(1)
: (deprecated) Create a tar archive of the files in the named tree object.

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.

Synching repositories

git-daemon(1)
: A really simple server for git repositories.

git-fetch-pack(1)
: Receive missing objects from another repository.

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-parse-remote(1)
: Routines to help parsing remote repository access parameters.

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-ref-format(1)
: Ensures that a reference name is well formed.

git-fmt-merge-msg(1)
: Produce a merge commit message.

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-peek-remote(1)
: (deprecated) List the references in a remote repository.

git-sh-setup(1)
: Common git shell script setup code.

git-stripspace(1)
: Filter out empty lines.

Configuration Mechanism

Starting from 0.99.9 (actually mid 0.99.8.GIT), `.git/config` file is used to hold per-repository configuration options. It is a simple text file modeled after `.ini` format familiar to some people. Here is an example:

```
#
# A '#' or ';' character indicates a comment.
#

; core variables
[core]
        ; Don't trust file modes
        filemode = false

; user identity
[user]
        name = "Junio C Hamano"
        email = "junkio@twinsun.com"
```

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

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 (i.e. the contents of `$GIT_DIR/HEAD`).

<tag>
: a valid tag *name* (i.e. the contents of `$GIT_DIR/refs/tags/<tag>`).

<head>
: a valid head *name* (i.e. the contents of `$GIT_DIR/refs/heads/<head>`).

For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in git-rev-parse(1).

File/Directory Structure

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 use the following environment variables:

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 Cogito etc.

GIT_INDEX_FILE
: This environment allows the specification of an alternate index file. If not specified, the default of `$GIT_DIR/index` is used.

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.

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.

GIT_WORK_TREE
: Set the path to the working tree. The value will not be used in combination with repositories found automatically in a .git directory (i.e. $GIT_DIR is not set). This can also be controlled by the *--work-tree* command line option and the core.worktree configuration variable.

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. It will not exclude the current working directory or a GIT_DIR set on the command line or in the environment. (Useful for excluding slow-loading network directories.)

git Commits

GIT_AUTHOR_NAME

GIT_AUTHOR_EMAIL

GIT_AUTHOR_DATE

GIT_COMMITTER_NAME

GIT_COMMITTER_EMAIL

GIT_COMMITTER_DATE

EMAIL
see git-commit-tree(1)

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, instead of the diff invocation described above. 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 SHA1 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>.

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_SSH
: If this environment variable is set then *git-fetch* and *git-push* will use this command instead of *ssh* when they need to connect to a remote system. The *$GIT_SSH* command will be given exactly two arguments: the *username@host* (or just *host*) from the URL and the shell command to execute on that remote system.

  To pass options to the program that you want to list in GIT_SSH you will need to wrap the program and options into a shell script, then set GIT_SSH to refer to the shell script.

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

GIT_FLUSH
: If this environment variable is set to "1", then commands such as *git-blame* (in incremental mode), *git-rev-list*, *git-log*, and *git-whatchanged* will force a flush of the output stream after each commit-oriented record have been flushed. If this variable is set to "0", 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
: If this variable is set to "1", "2" or "true" (comparison is case insensitive), git will print `trace:` messages on stderr telling about alias expansion, built-in command execution and external command execution. If this variable is set to an integer value greater than 1 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 this variable is set to an absolute path (starting with a / character), git will interpret this as a file path and will try to write the trace messages into it.

Discussion

More detail on the following is available from the git concepts chapter of the user-manual 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 SHA1 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 SHA1 name of an object or the name of another ref. Refs with names beginning `ref/head/` contain the SHA1 name of the most recent commit (or "head") of a branch under development. SHA1 names of tags of interest are stored under `ref/tags/`. A special 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.

Authors

- git's founding father is Linus Torvalds <torvalds@osdl.org>.
- The current git nurse is Junio C Hamano <gitster@pobox.com>.
- The git potty was written by Andreas Ericsson <ae@op5.se>.
- General upbringing is handled by the git-list <git@vger.kernel.org>.

Documentation

The documentation for git suite was started by David Greaves <david@dgreaves.com>, and later enhanced greatly by the contributors on the git-list <git@vger.kernel.org>.

SEE ALSO

gittutorial(7), gittutorial-2(7), Everyday Git, gitcvs-migration(7), gitglossary(7), gitcore-tutorial(7), gitcli(7), The Git User's Manual, gitworkflows(7)

GIT

Part of the git(1) suite

Last updated 2009-11-10 22:13:25 UTC