## NAME

git-commit - Record changes to the repository

## SYNOPSIS

*git commit* [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
[--dry-run] [(-c | -C | --fixup | --squash) <commit>]
[-F <file> | -m <msg>] [--reset-author] [--allow-empty]
[--allow-empty-message] [--no-verify] [-e] [--author=<author>]
[--date=<date>] [--cleanup=<mode>] [--[no-]status]
[-i | -o] [-S[<key-id>]] [--] [<file>...]

## DESCRIPTION

Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

The content to be added can be specified in several ways:

1. by using *git add* to incrementally add changes to the index before using the *commit* command (Note: even modified files must be added);
2. by using *git rm* to remove files from the working tree and the index, again before using the *commit* command;
3. by listing files as arguments to the *commit* command, in which case the commit will ignore changes staged in the index, and instead record the current content of the listed files (which must already be known to Git);
4. by using the -a switch with the *commit* command to automatically add changes from all known files (i.e. all files that are already listed in the index) and to automatically rm files in the index that have been removed from the working tree, and then perform the actual commit;
5. by using the --interactive or --patch switches with the *commit* command to decide one by one which files or hunks should be part of the commit, before finalizing the operation. See the "Interactive Mode" section of **git-add(1)** to learn how to operate these modes.

The --dry-run option can be used to obtain a summary of what is included by any of the above for the next commit by giving the same set of parameters (options and paths).

If you make a commit and then find a mistake immediately after that, you can recover from it with *git reset*.

## OPTIONS

-a, --all
: Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

-p, --patch
: Use the interactive patch selection interface to chose which changes to commit. See **git-add(1)** for details.

-C <commit>, --reuse-message=<commit>
: Take an existing commit object, and reuse the log message and the authorship information (including the timestamp) when creating the commit.

-c <commit>, --reedit-message=<commit>
: Like *-C*, but with *-c* the editor is invoked, so that the user can further edit the commit message.

--fixup=<commit>
: Construct a commit message for use with rebase --autosquash. The commit message will be the subject line from the specified commit with a prefix of fixup! . See **git-rebase(1)** for details.

--squash=<commit>
: Construct a commit message for use with rebase --autosquash. The commit message subject

line is taken from the specified commit with a prefix of squash! . Can be used with additional commit message options (-m/-c/-C/-F). See **git-rebase(1)** for details.

--reset-author
: When used with -C/-c/--amend options, or when committing after a a conflicting cherry-pick, declare that the authorship of the resulting commit now belongs of the committer. This also renews the author timestamp.

--short
: When doing a dry-run, give the output in the short-format. See **git-status(1)** for details. Implies --dry-run.

--branch
: Show the branch and tracking info even in short-format.

--porcelain
: When doing a dry-run, give the output in a porcelain-ready format. See **git-status(1)** for details. Implies --dry-run.

--long
: When doing a dry-run, give the output in a the long-format. Implies --dry-run.

-z, --null
: When showing short or porcelain status output, terminate entries in the status output with NUL, instead of LF. If no format is given, implies the --porcelain output format.

-F <file>, --file=<file>
: Take the commit message from the given file. Use - to read the message from the standard input.

--author=<author>
: Override the commit author. Specify an explicit author using the standard A U Thor <author@example.com> format. Otherwise <author> is assumed to be a pattern and is used to search for an existing commit by that author (i.e. rev-list --all -i --author=<author>); the commit author is then copied from the first such commit found.

--date=<date>
: Override the author date used in the commit.

-m <msg>, --message=<msg>
: Use the given <msg> as the commit message. If multiple -m options are given, their values are concatenated as separate paragraphs.

-t <file>, --template=<file>
: When editing the commit message, start the editor with the contents in the given file. The commit.template configuration variable is often used to give this option implicitly to the command. This mechanism can be used by projects that want to guide participants with some hints on what to write in the message in what order. If the user exits the editor without editing the message, the commit is aborted. This has no effect when a message is given by other means, e.g. with the -m or -F options.

-s, --signoff
: Add Signed-off-by line by the committer at the end of the commit log message.

-n, --no-verify
: This option bypasses the pre-commit and commit-msg hooks. See also **githooks(5)**.

--allow-empty
: Usually recording a commit that has the exact same tree as its sole parent commit is a mistake, and the command prevents you from making such a commit. This option bypasses the safety, and is primarily for use by foreign SCM interface scripts.

--allow-empty-message

Like --allow-empty this command is primarily for use by foreign SCM interface scripts. It allows you to create a commit with an empty commit message without using plumbing commands like **git-commit-tree(1)**.

--cleanup=<mode>
: This option determines how the supplied commit message should be cleaned up before committing. The <*mode*> can be strip, whitespace, verbatim, scissors or default.

  strip
  : Strip leading and trailing empty lines, trailing whitespace, and #commentary and collapse consecutive empty lines.

  whitespace
  : Same as strip except #commentary is not removed.

  verbatim
  : Do not change the message at all.

  scissors
  : Same as whitespace, except that everything from (and including) the line # ------------------------ >8 ------------------------ is truncated if the message is to be edited. # can be customized with core.commentChar.

  default
  : Same as strip if the message is to be edited. Otherwise whitespace.

  The default can be changed by the *commit.cleanup* configuration variable (see **git-config(1)**).

-e, --edit
: The message taken from file with -F, command line with -m, and from commit object with -C are usually used as the commit log message unmodified. This option lets you further edit the message taken from these sources.

--no-edit
: Use the selected commit message without launching an editor. For example, git commit --amend --no-edit amends a commit without changing its commit message.

--amend
: Replace the tip of the current branch by creating a new commit. The recorded tree is prepared as usual (including the effect of the -i and -o options and explicit pathspec), and the message from the original commit is used as the starting point, instead of an empty message, when no other message is specified from the command line via options such as -m, -F, -c, etc. The new commit has the same parents and author as the current one (the --reset-author option can countermand this).

  It is a rough equivalent for:

  $ git reset --soft HEAD^
  $ ... do something else to come up with the right tree ...
  $ git commit -c ORIG_HEAD

  but can be used to amend a merge commit.

  You should understand the implications of rewriting history if you amend a commit that has already been published. (See the RECOVERING FROM UPSTREAM REBASE section in **git-rebase(1)**.)

--no-post-rewrite
: Bypass the post-rewrite hook.

-i, --include
: Before making a commit out of staged contents so far, stage the contents of paths given on the command line as well. This is usually not what you want unless you are concluding a

conflicted merge.

-o, --only
Make a commit only from the paths specified on the command line, disregarding any contents that have been staged so far. This is the default mode of operation of *git commit* if any paths are given on the command line, in which case this option can be omitted. If this option is specified together with *--amend*, then no paths need to be specified, which can be used to amend the last commit without committing changes that have already been staged.

-u[<mode>], --untracked-files[=<mode>]
Show untracked files.

The mode parameter is optional (defaults to *all*), and is used to specify the handling of untracked files; when -u is not used, the default is *normal*, i.e. show untracked files and directories.

The possible options are:
- *no* - Show no untracked files
- *normal* - Shows untracked files and directories
- *all* - Also shows individual files in untracked directories.

The default can be changed using the status.showUntrackedFiles configuration variable documented in **git-config(1)**.

-v, --verbose
Show unified diff between the HEAD commit and what would be committed at the bottom of the commit message template. Note that this diff output doesn't have its lines prefixed with *#*.

-q, --quiet
Suppress commit summary message.

--dry-run
Do not create a commit, but show a list of paths that are to be committed, paths with local changes that will be left uncommitted and paths that are untracked.

--status
Include the output of **git-status(1)** in the commit message template when using an editor to prepare the commit message. Defaults to on, but can be used to override configuration variable commit.status.

--no-status
Do not include the output of **git-status(1)** in the commit message template when using an editor to prepare the default commit message.

-S[<keyid>], --gpg-sign[=<keyid>]
GPG-sign commit.

--no-gpg-sign
Countermand commit.gpgsign configuration variable that is set to force each and every commit to be signed.

--
Do not interpret any more arguments as options.

<file>...
When files are given on the command line, the command commits the contents of the named files, without recording the changes already staged. The contents of these files are also staged for the next commit on top of what have been staged before.

## DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables and the --date option support the following date formats:

Git internal format
: It is <unix timestamp> <time zone offset>, where <unix timestamp> is the number of seconds since the UNIX epoch. <time zone offset> is a positive or negative offset from UTC. For example CET (which is 2 hours ahead UTC) is +0200.

RFC 2822
: The standard email format as described by RFC 2822, for example Thu, 07 Apr 2005 22:13:13 +0200.

ISO 8601
: Time and date specified by the ISO 8601 standard, for example 2005-04-07T22:13:13. The parser accepts a space instead of the T character as well.

**Note**

In addition, the date part is accepted in the following formats: YYYY.MM.DD, MM/DD/YYYY and DD.MM.YYYY.

## EXAMPLES

When recording your own work, the contents of modified files in your working tree are temporarily stored to a staging area called the index with *git add*. A file can be reverted back, only in the index but not in the working tree, to that of the last commit with git reset HEAD -- <file>, which effectively reverts *git add* and prevents the changes to this file from participating in the next commit. After building the state to be committed incrementally with these commands, git commit (without any pathname parameter) is used to record what has been staged so far. This is the most basic form of the command. An example:

$ edit hello.c
$ git rm goodbye.c
$ git add hello.c
$ git commit

Instead of staging files after each individual change, you can tell git commit to notice the changes to the files whose contents are tracked in your working tree and do corresponding git add and git rm for you. That is, this example does the same as the earlier example if there is no other change in your working tree:

$ edit hello.c
$ rm goodbye.c
$ git commit -a

The command git commit -a first looks at your working tree, notices that you have modified hello.c and removed goodbye.c, and performs necessary git add and git rm for you.

After staging changes to many files, you can alter the order the changes are recorded in, by giving pathnames to git commit. When pathnames are given, the command makes a commit that only records the changes made to the named paths:

$ edit hello.c hello.h
$ git add hello.c hello.h
$ edit Makefile
$ git commit Makefile

This makes a commit that records the modification to Makefile. The changes staged for hello.c and hello.h are not included in the resulting commit. However, their changes are not lost — they are still staged and merely held back. After the above sequence, if you do:

$ git commit

this second commit would record the changes to hello.c and hello.h as expected.

After a merge (initiated by *git merge* or *git pull*) stops because of conflicts, cleanly merged paths are already staged to be committed for you, and paths that conflicted are left in unmerged state.

You would have to first check which paths are conflicting with *git status* and after fixing them manually in your working tree, you would stage the result as usual with *git add*:

$ git status | grep unmerged
unmerged: hello.c
$ edit hello.c
$ git add hello.c

After resolving conflicts and staging the result, git ls-files -u would stop mentioning the conflicted path. When you are done, run git commit to finally record the merge:

$ git commit

As with the case to record your own changes, you can use -a option to save typing. One difference is that during a merge resolution, you cannot use git commit with pathnames to alter the order the changes are committed, because the merge should be recorded as a single commit. In fact, the command refuses to run when given pathnames (but see -i option).

## DISCUSSION

Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, **git-format-patch(1)** turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

At the core level, Git is character encoding agnostic.

- The pathnames recorded in the index and in the tree objects are treated as uninterpreted sequences of non-NUL bytes. What readdir(2) returns are what are recorded and compared with the data Git keeps track of, which in turn are expected to be what lstat(2) and creat(2) accepts. There is no such thing as pathname encoding translation.
- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- The commit log messages are uninterpreted sequences of non-NUL bytes.

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in .git/config file, like this:

   [i18n]
   commitencoding = ISO-8859-1

   Commit objects created with the above setting record the value of i18n.commitencoding in its encoding header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.
2. *git log*, *git show*, *git blame* and friends look at the encoding header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with i18n.logoutputencoding in .git/config file, like this:

   [i18n]
   logoutputencoding = ISO-8859-1

   If you do not have this configuration variable, the value of i18n.commitencoding is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to

force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

## ENVIRONMENT AND CONFIGURATION VARIABLES

The editor used to edit the commit log message will be chosen from the GIT_EDITOR environment variable, the core.editor configuration variable, the VISUAL environment variable, or the EDITOR environment variable (in that order). See **git-var(1)** for details.

## HOOKS

This command can run commit-msg, prepare-commit-msg, pre-commit, and post-commit hooks. See **githooks(5)** for more information.

## FILES

$GIT_DIR/COMMIT_EDITMSG

This file contains the commit message of a commit in progress. If git commit exits due to an error before creating a commit, any commit message that has been provided by the user (e.g., in an editor session) will be available in this file, but will be overwritten by the next invocation of git commit.

## SEE ALSO

**git-add(1)**, **git-rm(1)**, **git-mv(1)**, **git-merge(1)**, **git-commit-tree(1)**

## GIT

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