NAME

git-commit - Record changes to the repository

SYNOPSIS

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
           [--dry-run] [(-c | -C | --squash) <commit> | --fixup [(amend|reword):]<co
mmit>)]
           [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
           [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
           [--date=<date>] [--cleanup=<mode>] [--[no-]status]
           [-i | -o] [--pathspec-from-file=<file> [--pathspec-file-nul]]
           [(--trailer <token>[(=|:)<value>])...] [-S[<keyid>]]
           [--] [<pathspec>...]
```

DESCRIPTION

Create a new commit containing the current contents of the index and the given log message describing the changes. The new commit is a direct child of HEAD, usually the tip of the current branch, and the branch is updated to point to it (unless no branch is associated with the working tree, in which case HEAD is "detached" as described in git-checkout(1)).

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

1. by using git-add(1) to incrementally "add" changes to the index before using the commit command (Note: even modified files must be "added");

2. by using git-rm(1) 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 (without --interactive or --patch switch), 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 in addition to contents in the index, 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 choose 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=[(amend|reword):]<commit>
Create a new commit which "fixes up" <commit> when applied with git rebase --autosquash. Plain --fixup=<commit> creates a "fixup!" commit which changes the content of <commit> but leaves its log message untouched. --fixup=amend:<commit> is similar but creates an "amend!" commit which also replaces the log message of <commit> with the log message of the "amend!" commit.
--fixup=reword:<commit> creates an "amend!" commit which replaces the log message of <commit> with its own log message but makes no changes to the content of <commit>.

The commit created by plain --fixup=<commit> has a subject composed of "fixup!" followed by the subject line from <commit>, and is recognized specially by git rebase --autosquash. The -m option may be used to supplement the log message of the created commit, but the additional commentary will be thrown away once the "fixup!" commit is squashed into <commit> by git rebase --autosquash.

The commit created by --fixup=amend:<commit> is similar but its subject is instead prefixed with "amend!". The log message of <commit> is copied into the log message of the "amend!" commit and opened in an editor so it can be refined. When git rebase --autosquash squashes the "amend!" commit into <commit>, the log message of <commit> is replaced by the refined log message from the "amend!" commit. It is an error for the "amend!" commit's log message to be empty unless --allow-empty-message is specified.

--fixup=reword:<commit> is shorthand for --fixup=amend:<commit> --only. It creates an "amend!" commit with only a log message (ignoring any changes staged in the index). When squashed by git rebase --autosquash, it replaces the log message of <commit> without making any other changes.

Neither "fixup!" nor "amend!" commits change authorship of <commit> when applied by git rebase --autosquash. 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 conflicting cherry-pick, declare that the authorship of the resulting commit now belongs to 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 the long-format. Implies --dry-run.

-z, --null
When showing short or porcelain status output, print the filename verbatim and terminate the entries with NUL, instead of LF. If no format is given, implies the --porcelain output format. Without the -z option, filenames with "unusual" characters are quoted as explained for the configuration variable core.quotePath (see git-config(1)).

-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.

The -m option is mutually exclusive with -c, -C, and -F.

-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, --no-signoff
Add a Signed-off-by trailer by the committer at the end of the commit log message. The meaning of a signoff depends on the project to which you're committing. For example, it may certify that the committer has the rights to submit the work under the project's license or agrees to some contributor representation, such as a Developer Certificate of Origin. (See http://developercertificate.org for the one used by the Linux kernel and Git projects.) Consult the documentation or leadership of the project to which you're contributing to understand how the signoffs are used in that project.

The --no-signoff option can be used to countermand an earlier --signoff option on the command line.

--trailer <token>[(=|:)<value>]
Specify a (<token>, <value>) pair that should be applied as a trailer. (e.g. git commit --trailer "Signed-off-by:C O Mitter \ <committer@example.com>" --trailer "Helped-by:C O Mitter \ <committer@example.com>" will add the "Signed-off-by" trailer and the "Helped-by" trailer to the commit message.) The trailer.* configuration variables (git-interpret-trailers(1)) can be used to define if a duplicated trailer is omitted, where in the run of trailers each trailer would appear, and other details.

-n, --[no-]verify

By default, the pre-commit and commit-msg hooks are run. When any of --no-verify or -n is given, these are bypassed. 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, 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 found below is truncated, if the message is to be edited. "#" can be customized with core.commentChar.

```
# ------------------------ >8 ------------------------
```

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 by taking the updated working tree contents of the paths specified on the command line, disregarding any contents that have been staged for other paths. 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. If used together with --allow-empty paths are also not required, and an empty commit will be created.

--pathspec-from-file=<file>
Pathspec is passed in <file> instead of commandline args. If <file> is exactly - then standard input is used. Pathspec elements are separated by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable core.quotePath (see git-config(1)). See also --pathspec-file-nul and global --literal-pathspecs.

--pathspec-file-nul
Only meaningful with --pathspec-from-file. Pathspec elements are separated with NUL character and all other characters are taken literally (including newlines and quotes).

-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 to help the user describe the commit by reminding what changes the commit has. Note that this diff output doesn't have its lines prefixed with #. This diff will not be a part of the commit message. See the

commit.verbose configuration variable in git-config(1).

If specified twice, show in addition the unified diff between what would be committed and the worktree files, i.e. the unstaged changes to tracked files.

-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>], --no-gpg-sign
GPG-sign commits. The keyid argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space. --no-gpg-sign is useful to countermand both commit.gpgSign configuration variable, and earlier --gpg-sign.

--
Do not interpret any more arguments as options.

<pathspec>...
When pathspec is given on the command line, commit the contents of the files that match the pathspec without recording the changes already added to the index. The contents of these files are also staged for the next commit on top of what have been staged before.

For more details, see the pathspec entry in gitglossary(7).

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 restore --staged <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).

COMMIT INFORMATION

Author and committer information is taken from the following environment variables, if set:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

(nb "<", ">" and "\n"s are stripped)

The author and committer names are by convention some form of a personal name (that is, the name by which other humans refer to you), although Git does not enforce or require any particular form. Arbitrary Unicode may be used, subject to the constraints listed above. This name has no effect on authentication; for that, see the credential.username variable in git-config(1).

In case (some of) these environment variables are not set, the information is taken from the configuration items user.name and user.email, or, if not present, the environment variable EMAIL, or, if that is not set, system user name and the hostname used for outgoing mail (taken from /etc/mailname and falling back to the fully qualified hostname when that file does not exist).

The author.name and committer.name and their corresponding email options override user.name and user.email if set and are overridden themselves by the environment variables.

The typical usage is to set just the user.name and user.email variables; the other options are provided for more complex use cases.

DATE FORMATS

The GIT_AUTHOR_DATE and GIT_COMMITTER_DATE environment variables 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 1 hour ahead of UTC) is +0100.

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. Fractional parts of a second will be ignored, for example 2005-04-07T22:13:13.019 will be treated as 2005-04-07T22:13:13.

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

In addition to recognizing all date formats above, the --date option will also try to make sense of other, more human-centric date formats, such as relative dates like "yesterday" or "last Friday at noon".

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.

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.

- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (.git/config (see git-config(1)), gitignore(5), gitattributes(5) and gitmodules(5)).

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that

use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but not UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

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, post-commit and post-rewrite 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