NAME

githooks − Hooks used by git

SYNOPSIS

$GIT_DIR/hooks/*

DESCRIPTION

Hooks are little scripts you can place in $GIT_DIR/hooks directory to trigger action at certain points. When *git init* is run, a handful of example hooks are copied into the hooks directory of the new repository, but by default they are all disabled. To enable a hook, rename it by removing its .sample suffix.

Note

It is also a requirement for a given hook to be executable. However − in a freshly initialized repository − the .sample files are executable by default.

This document describes the currently defined hooks.

HOOKS

applypatch−msg

This hook is invoked by *git am* script. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with non−zero status causes *git am* to abort before applying the patch.

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format (if the project has one). It can also be used to refuse the commit after inspecting the message file.

The default *applypatch−msg* hook, when enabled, runs the *commit−msg* hook, if the latter is enabled.

pre−applypatch

This hook is invoked by *git am*. It takes no parameter, and is invoked after the patch is applied, but before a commit is made.

If it exits with non−zero status, then the working tree will not be committed after applying the patch.

It can be used to inspect the current working tree and refuse to make a commit if it does not pass certain test.

The default *pre−applypatch* hook, when enabled, runs the *pre−commit* hook, if the latter is enabled.

post−applypatch

This hook is invoked by *git am*. It takes no parameter, and is invoked after the patch is applied and a commit is made.

This hook is meant primarily for notification, and cannot affect the outcome of *git am*.

pre−commit

This hook is invoked by *git commit*, and can be bypassed with −−no−verify option. It takes no parameter, and is invoked before obtaining the proposed commit log message and making a commit. Exiting with non−zero status from this script causes the *git commit* to abort.

The default *pre−commit* hook, when enabled, catches introduction of lines with trailing whitespaces and aborts the commit when such a line is found.

All the *git commit* hooks are invoked with the environment variable GIT_EDITOR=: if the command will not bring up an editor to modify the commit message.

prepare−commit−msg
This hook is invoked by *git commit* right after preparing the default log message, and before the editor is started.

It takes one to three parameters. The first is the name of the file that contains the commit log message. The second is the source of the commit message, and can be: message (if a −m or −F option was given); template (if a −t option was given or the configuration option commit.template is set); merge (if the commit is a merge or a .git/MERGE_MSG file exists); squash (if a .git/SQUASH_MSG file exists); or commit, followed by a commit SHA1 (if a −c, −C or −−amend option was given).

If the exit status is non−zero, *git commit* will abort.

The purpose of the hook is to edit the message file in place, and it is not suppressed by the −−no−verify option. A non−zero exit means a failure of the hook and aborts the commit. It should not be used as replacement for pre−commit hook.

The sample prepare−commit−msg hook that comes with git comments out the Conflicts: part of a merge's commit message.

commit−msg
This hook is invoked by *git commit*, and can be bypassed with −−no−verify option. It takes a single parameter, the name of the file that holds the proposed commit log message. Exiting with non−zero status causes the *git commit* to abort.

The hook is allowed to edit the message file in place, and can be used to normalize the message into some project standard format (if the project has one). It can also be used to refuse the commit after inspecting the message file.

The default *commit−msg* hook, when enabled, detects duplicate "Signed−off−by" lines, and aborts the commit if one is found.

post−commit
This hook is invoked by *git commit*. It takes no parameter, and is invoked after a commit is made.

This hook is meant primarily for notification, and cannot affect the outcome of *git commit*.

pre−rebase
This hook is called by *git rebase* and can be used to prevent a branch from getting rebased.

post−checkout
This hook is invoked when a *git checkout* is run after having updated the worktree. The hook is given three parameters: the ref of the previous HEAD, the ref of the new HEAD (which may or may not have changed), and a flag indicating whether the checkout was a branch checkout (changing branches, flag=1) or a file checkout (retrieving a file from the index, flag=0). This hook cannot affect the outcome of *git checkout*.

It is also run after *git clone*, unless the −−no−checkout (−n) option is used. The first parameter given to the hook is the null−ref, the second the ref of the new HEAD and the flag is always 1.

This hook can be used to perform repository validity checks, auto−display differences from the previous HEAD if different, or set working dir metadata properties.

post−merge
This hook is invoked by *git merge*, which happens when a *git pull* is done on a local repository. The hook takes a single parameter, a status flag specifying whether or not the merge being done was a squash merge. This hook cannot affect the outcome of *git merge* and is not executed, if the merge failed due to conflicts.

This hook can be used in conjunction with a corresponding pre−commit hook to save and restore any form

of metadata associated with the working tree (eg: permissions/ownership, ACLS, etc). See contrib/hooks/setgitperms.perl for an example of how to do this.

pre−receive

This hook is invoked by *git−receive−pack* on the remote repository, which happens when a *git push* is done on a local repository. Just before starting to update refs on the remote repository, the pre−receive hook is invoked. Its exit status determines the success or failure of the update.

This hook executes once for the receive operation. It takes no arguments, but for each ref to be updated it receives on standard input a line of the format:

<old−value> SP <new−value> SP <ref−name> LF

where <old−value> is the old object name stored in the ref, <new−value> is the new object name to be stored in the ref and <ref−name> is the full name of the ref. When creating a new ref, <old−value> is 40 0.

If the hook exits with non−zero status, none of the refs will be updated. If the hook exits with zero, updating of individual refs can still be prevented by the *update* hook.

Both standard output and standard error output are forwarded to *git send−pack* on the other end, so you can simply echo messages for the user.

update

This hook is invoked by *git−receive−pack* on the remote repository, which happens when a *git push* is done on a local repository. Just before updating the ref on the remote repository, the update hook is invoked. Its exit status determines the success or failure of the ref update.

The hook executes once for each ref to be updated, and takes three parameters:

- the name of the ref being updated,
- the old object name stored in the ref,
- and the new objectname to be stored in the ref.

A zero exit from the update hook allows the ref to be updated. Exiting with a non−zero status prevents *git−receive−pack* from updating that ref.

This hook can be used to prevent *forced* update on certain refs by making sure that the object name is a commit object that is a descendant of the commit object named by the old object name. That is, to enforce a "fast−forward only" policy.

It could also be used to log the old..new status. However, it does not know the entire set of branches, so it would end up firing one e−mail per ref when used naively, though. The *post−receive* hook is more suited to that.

Another use suggested on the mailing list is to use this hook to implement access control which is finer grained than the one based on filesystem group.

Both standard output and standard error output are forwarded to *git send−pack* on the other end, so you can simply echo messages for the user.

The default *update* hook, when enabled—and with hooks.allowunannotated config option unset or set to false—prevents unannotated tags to be pushed.

post−receive

This hook is invoked by *git−receive−pack* on the remote repository, which happens when a *git push* is done on a local repository. It executes on the remote repository once after all the refs have been updated.

This hook executes once for the receive operation. It takes no arguments, but gets the same information as the *pre−receive* hook does on its standard input.

This hook does not affect the outcome of *git−receive−pack*, as it is called after the real work is done.

This supersedes the *post−update* hook in that it gets both old and new values of all the refs in addition to their names.

Both standard output and standard error output are forwarded to *git send−pack* on the other end, so you can simply echo messages for the user.

The default *post−receive* hook is empty, but there is a sample script post−receive−email provided in the contrib/hooks directory in git distribution, which implements sending commit emails.

post−update

This hook is invoked by *git−receive−pack* on the remote repository, which happens when a *git push* is done on a local repository. It executes on the remote repository once after all the refs have been updated.

It takes a variable number of parameters, each of which is the name of ref that was actually updated.

This hook is meant primarily for notification, and cannot affect the outcome of *git−receive−pack*.

The *post−update* hook can tell what are the heads that were pushed, but it does not know what their original and updated values are, so it is a poor place to do log old..new. The *post−receive* hook does get both original and updated values of the refs. You might consider it instead if you need them.

When enabled, the default *post−update* hook runs *git update−server−info* to keep the information used by dumb transports (e.g., HTTP) up−to−date. If you are publishing a git repository that is accessible via HTTP, you should probably enable this hook.

Both standard output and standard error output are forwarded to *git send−pack* on the other end, so you can simply echo messages for the user.

pre−auto−gc

This hook is invoked by *git gc −−auto*. It takes no parameter, and exiting with non−zero status from this script causes the *git gc −−auto* to abort.

post−rewrite

This hook is invoked by commands that rewrite commits (git commit −−amend, *git−rebase*; currently *git−filter−branch* does *not* call it!). Its first argument denotes the command it was invoked by: currently one of amend or rebase. Further command−dependent arguments may be passed in the future.

The hook receives a list of the rewritten commits on stdin, in the format

<old−sha1> SP <new−sha1> [SP <extra−info>] LF

The *extra−info* is again command−dependent. If it is empty, the preceding SP is also omitted. Currently, no commands pass any *extra−info*.

The hook always runs after the automatic note copying (see "notes.rewrite.<command>" in linkgit:git−config.txt) has happened, and thus has access to these notes.

The following command−specific comments apply:

rebase

For the *squash* and *fixup* operation, all commits that were squashed are listed as being rewritten to the squashed commit. This means that there will be several lines sharing the same *new−sha1*.

The commits are guaranteed to be listed in the order that they were processed by rebase.

There is no default *post−rewrite* hook, but see the post−receive−copy−notes script in contrib/hooks for an example that copies your git−notes to the rewritten commits.

GIT

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