git-hook

git-hook(1) ==========

NAME ---- git-hook - Run Git hooks

SYNOPSIS --------

> 'git hook' run [--allow-unknown-hook-name] [--ignore-missing] [--to-stdin=<path>] [(-j|--jobs) <n>]
> <hook-name> [-- <hook-args>]
> 'git hook' list [--allow-unknown-hook-name] [-z] [--show-scope] <hook-name>

DESCRIPTION -----------

A command interface for running Git hooks (see linkgit:githooks[5]), for use by other scripted Git commands.

This command parses the default configuration files for sets of configs like so:

```
[hook "linter"]
  event = pre-commit
  command = ~/bin/linter --cpp20
```

In this example, `[hook` `"linter"]` represents one script - `~/bin/linter` `--cpp20` - which can be shared by many repos, and even by many hook events, if appropriate.

To add an unrelated hook which runs on a different event, for example a spell-checker for your commit messages, you would write a configuration like so:

```
[hook "linter"]
  event = pre-commit
  command = ~/bin/linter --cpp20
[hook "spellcheck"]
  event = commit-msg
  command = ~/bin/spellchecker
```

With this config, when you run 'git commit', first `~/bin/linter` `--cpp20` will have a chance to check your files to be committed (during the `pre-commit` hook event), and then `~/bin/spellchecker` will

have a chance to check your commit message (during the `commit-msg` hook event).

Commands are run in the order Git encounters their associated `hook.<friendly-name>.event` configs during the configuration parse (see linkgit:git-config[1]). Although multiple `hook.linter.event` configs can be added, only one `hook.linter.command` event is valid - Git uses "last-one-wins" to determine which command to run.

So if you wanted your linter to run when you commit as well as when you push, you would configure it like so:

```
[hook "linter"]
  event = pre-commit
  event = pre-push
  command = ~/bin/linter --cpp20
```

With this config, `~/bin/linter` `--cpp20` would be run by Git before a commit is generated (during `pre-commit`) as well as before a push is performed (during `pre-push`).

And if you wanted to run your linter as well as a secret-leak detector during only the "pre-commit" hook event, you would configure it instead like so:

```
[hook "linter"]
  event = pre-commit
  command = ~/bin/linter --cpp20
[hook "no-leaks"]
  event = pre-commit
  command = ~/bin/leak-detector
```

With this config, before a commit is generated (during `pre-commit`), Git would first start `~/bin/linter` `--cpp20` and second start `~/bin/leak-detector`. It would evaluate the output of each when deciding whether to proceed with the commit.

For a full list of hook events which you can set your `hook.<friendly-name>.event` to, and how hooks are invoked during those events, see linkgit:githooks[5].

Git will ignore any `hook.<friendly-name>.event` that specifies an event it doesn’t recognize. This is intended so that tools which wrap Git can use the hook infrastructure to run their own hooks; see "WRAPPERS" for more guidance.

In general, when instructions suggest adding a script to `.git/hooks/<hook-event>`, you can specify it in the config instead by running:

```
git config set hook.<some-name>.command <path-to-script>
git config set --append hook.<some-name>.event <hook-event>
```

This way you can share the script between multiple repos. That is, `cp ~/my-script.sh ~/project/.git/hooks/pre-commit` would become:

```
git config set hook.my-script.command ~/my-script.sh
git config set --append hook.my-script.event pre-commit
```

SUBCOMMANDS ----------

run
Runs hooks configured for `<hook-name>`, in the order they are discovered during the config parse. The default `<hook-name>` from the hookdir is run last. See linkgit:githooks[5] for supported hook names.

Any positional arguments to the hook should be passed after a mandatory `--` (or `--end-of-options`, see linkgit:gitcli[7]). See linkgit:githooks[5] for arguments hooks might expect (if any).

list [-z] [--show-scope]
Print a list of hooks which will be run on `<hook-name>` event. If no hooks are configured for that event, print a warning and return 1. Use `-z` to terminate output lines with NUL instead of newlines.

OPTIONS -------

--allow-unknown-hook-name
By default `git hook run` and `git hook list` will bail out when `<hook-name>` is not a hook event known to Git (see linkgit:githooks[5] for the list of known hooks). This is meant to help catch typos such as `prereceive` when `pre-receive` was intended. Pass this flag to allow unknown hook names.

--to-stdin
For "run"; specify a file which will be streamed into the hook’s stdin. The hook will receive the entire file from beginning to EOF.

--ignore-missing
Ignore any missing hook by quietly returning zero. Used for tools that want to do a blind one-shot run of a hook that may or may not be present.

-z
Terminate "list" output lines with NUL instead of newlines.

--show-scope
For "list"; prefix each configured hook’s friendly name with a tab-separated config scope (e.g. `local`, `global`, `system`), mirroring the output style of `git config --show-scope`. Traditional hooks from the hookdir are unaffected.

-j

--jobs
Only valid for `run`.

Specify how many hooks to run simultaneously. If this flag is not specified, the value of the `hook.jobs` config is used, see linkgit:git-config[1]. If neither is specified, defaults to 1 (serial execution).

When greater than 1, it overrides the per-hook `hook.<friendly-name>.parallel` setting, allowing all hooks for the event to run concurrently, even if they are not individually marked as parallel.

Some hooks always run sequentially regardless of this flag or the `hook.jobs` config, because Git knows they cannot safely run in parallel: `applypatch-msg`, `pre-commit`, `prepare-commit-msg`, `commit-msg`, `post-commit`, `post-checkout`, and `push-to-checkout`.

WRAPPERS --------

`git` `hook` `run` has been designed to make it easy for tools which wrap Git to configure and execute hooks using the Git hook infrastructure. It is possible to provide arguments and stdin via the command line, as well as specifying parallel or series execution if the user has provided multiple hooks.

Assuming your wrapper wants to support a hook named "mywrapper-start-tests", you can have your users specify their hooks like so:

```
[hook "setup-test-dashboard"]
  event = mywrapper-start-tests
  command = ~/mywrapper/setup-dashboard.py --tap
```

Then, in your ’mywrapper’ tool, you can invoke any users’ configured hooks by running:

```
git hook run --allow-unknown-hook-name mywrapper-start-tests \
  # providing something to stdin
  --stdin some-tempfile-123 \
  # execute multiple hooks in parallel
  --jobs 3 \
  # plus some arguments of your own...
  -- \
  --testname bar \
  baz
```

Take care to name your wrapper’s hook events in a way which is unlikely to overlap with Git’s native hooks (see linkgit:githooks[5]) - a hook event named `mywrappertool-validate-commit` is much less likely to be added to native Git than a hook event named `validate-commit`. If Git begins to use a hook event named the same thing as your wrapper hook, it may invoke your users’ hooks in

unintended and unsupported ways.

CONFIGURATION ---------- :git-hook: 1 include::config/hook.adoc[]

SEE ALSO -------- linkgit:githooks[5]

GIT --- Part of the linkgit:git[1] suite

□□□ #1
□ □□□ □□□ 2026-10-31 20:11:15 UTC
□ □□□ □□□ 2026-10-31 20:11:15 UTC