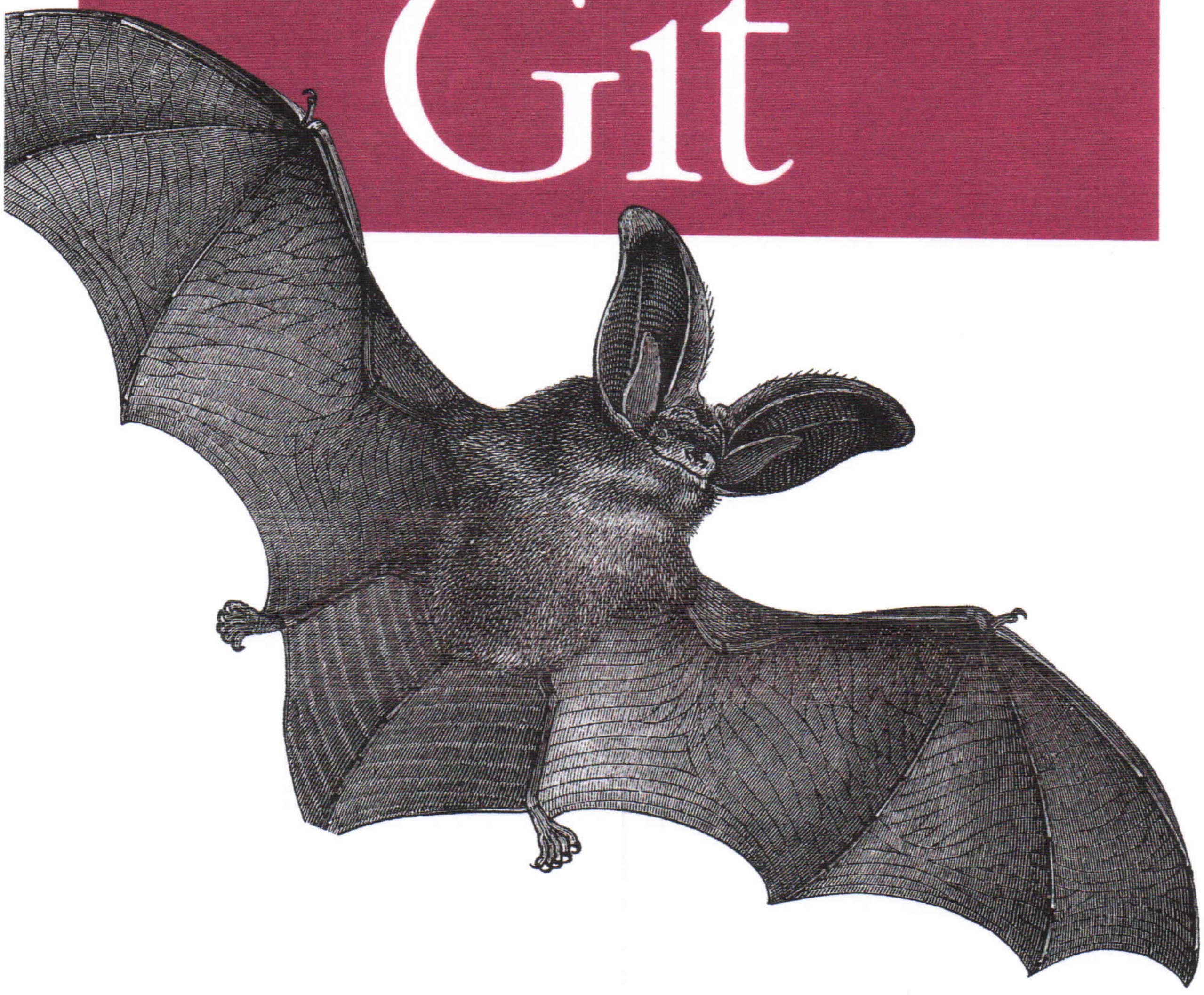
Powerful Tools and Techniques for Collaborative Software Development

2nd Edition
Covers GitHub

Version Control with Git



O'REILLY®

Jon Loeliger & Matthew McCullough

Version Control with Git, Second Edition
by Jon Loeliger and Matthew McCullough

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Andy Oram
Production Editor: Iris Febres
Copyeditor: Absolute Service, Inc.
Proofreader: Absolute Service, Inc.
Indexer: Nancy Guenther on behalf of Potomac Indexing, LLC
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Rebecca Demarest

May 2009:	First Edition.
August 2012:	Second Edition.

Revision History for the Second Edition:

2012-08-03	First release
2013-06-21	Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449316389* for release details.

ISBN: 978-1-449-31638-9

[LSI]

1371751874

Table of Contents

CHAPTER 15

Hooks

You can use a Git *hook* to run one or more arbitrary scripts whenever a particular event, such as a commit or a patch, occurs in your repository. Typically, an event is broken into several prescribed steps, and you can tie a custom script to each step. When the Git event occurs, the appropriate script is called at the outset of each step.

Hooks belong to and affect a specific repository and are not copied during a clone operation. In other words, hooks you set up in your private repository are not propagated to and do not alter the behavior of the new clone. If for some reason your development process mandates hooks in each coder's personal development repository, arrange to copy the directory *.git/hooks* through some other (nonclone) method.

A hook runs either in the context of your current, local repository or in the context of the remote repository. For example, fetching data into your repository from a remote repository and making a local commit can cause local hooks to run; pushing changes to a remote repository may cause hooks in the remote repository to run.

Most Git hooks fall into one of two categories:

- A *"pre" hook* runs before an action completes. You can use this kind of hook to approve, reject, or adjust a change before it's applied.
- A *"post" hook* runs after an action completes and can be used to trigger notifications (such as email) or to launch additional processing, such as running a build or closing a bug.

As a general rule, if a *pre*-action hook exits with a nonzero status (the convention to indicate failure), the Git action is aborted. In contrast, the exit status of a *post*-action hook is generally ignored because the hook can no longer affect the outcome or completion of the action.

In general, the Git developers advocate using hooks with caution. A hook, they say, should be a method of last resort, to be used only when you can't accomplish the same result in some other way. For example, if you want to specify a particular option each time you make a commit, check out a file, or create a branch, a hook is unnecessary.

You can accomplish the same task with a Git alias (see "Configuring an Alias" on page 30 in Chapter 3) or with shell scripts to augment `git commit`, `git checkout`, and `git branch`, respectively.[1]

At first blush, a hook may seem an appealing and straightforward solution. However, there are several implications of its use.

- A hook changes the behavior of Git. If a hook performs an unusual operation, other developers familiar with Git may run into surprises when using your repository.
- A hook can slow operations that are otherwise fast. For example, developers are often enticed to hook Git to run unit tests before anyone makes a commit, but this makes committing slow. In Git, a commit is supposed to be a fast operation, thus encouraging frequent commits to prevent the loss of data. Making a commit run slowly makes Git less enjoyable.
- A hook script that is buggy can interfere with your work and productivity. The only way to work around a hook is to disable it. In contrast, if you use an alias or shell script instead of a hook, then you can always fall back on the normal Git command wherever that makes sense.
- A repository's collection of hooks is not automatically replicated. Hence, if you install a commit hook in your repository, it won't reliably affect another developer's commits. This is partly for security reasons—a malicious script could easily be smuggled into an otherwise innocuous-looking repository—and partly because Git simply has no mechanism to replicate anything other than blobs, trees, and commits.

Junio's Overview of Hooks

Junio Hamano wrote the following about Git hooks on the Git mailing list (paraphrased from the original).

> There are five valid reasons to hook a Git command/operation:
>
> 1. To countermand the decision made by the underlying command. The `update` hook and the `pre-commit` hook are two hooks used for this purpose.
> 2. To manipulate data generated after a command starts to run. Modifying the commit log message in the `commit-msg` hook is an example.
> 3. To operate on the remote end of a connection, that you access only via the Git protocol. A `post-update` hook that runs `git update-server-info` does this very task.
> 4. To acquire a lock for mutual exclusion. This is rarely a requirement, but sufficient hooks are available to achieve it.

1. As it happens, running a hook at commit time is such a common requirement that a precommit hook exists for that, even though it isn't strictly necessary.

5. To run one of several possible operations, depending on the outcome of the command. The `post-checkout` hook is a notable example.

Each of these five requirements requires at least one hook. You cannot realize a similar result from outside the Git command.

On the other hand, if you always want some action to occur before or after running a Git operation locally, you don't need a hook. For instance, if your postprocessing depends on the effects of a command (item 5 in the list) but the results of the command are plainly observable, then you don't need a hook.

With those "warnings" behind us, we can state that hooks exist for very good reasons and that their use can be incredibly advantageous.

Installing Hooks

Each hook is a script, and the collection of hooks for a particular repository can be found in the *.git/hooks* directory. As already mentioned, Git doesn't replicate hooks between repositories; if you `git clone` or `git fetch` from another repository, you won't inherit that repository's hooks. You have to copy the hook scripts by hand.

Each hook script is named after the event with which it is associated. For example, the hook that runs immediately before a `git commit` operation is named *.git/hooks/pre-commit*.

A hook script must follow the normal rules for Unix scripts: it must be executable (`chmod a+x .git/hooks/pre-commit`) and must start with a line indicating the language in which the script is written (for example, `#!/bin/bash` or `#!/usr/bin/perl`).

If a particular hook script exists and has the correct name and file permissions, Git uses it automatically.

Example Hooks

Depending on your exact version of Git, you may find some hooks in your repository at the time it's created. Hooks are copied automatically from your Git template directory when you create a new repository. On Debian and Ubuntu, for example, the hooks are copied from */usr/share/git-core/templates/hooks*. Most Git versions include some example hooks that you can use, and these are preinstalled for you in the templates directory.

Here's what you need to know about the example hooks:

- The template hooks probably don't do exactly what you want. You can read them, edit them, and learn from them, but you rarely want to use them as is.
- Even though the hooks are created by default, all the hooks are initially disabled. Depending on your version of Git and your operating system, the hooks are

disabled either by removing the execute bit or by appending *.sample* to the hook file name. Modern versions of Git have executable hooks named with a *.sample* suffix.

- To enable an example hook, you must remove the *.sample* suffix from its filename (`mv .git/hooks/pre-commit.sample .git/hooks/pre-commit`) and set its execute bit, as is apropos (`chmod a+x .git/hooks/pre-commit`).

Originally, each example hook was simply copied into the *.git/hooks/* directory from the template directory with its execute permission removed. You could then enable the hook by setting its execute bit.

That worked fine on systems like Unix and Linux, but didn't work well on Windows. In Windows, file permissions work differently and, unfortunately, files are executable by default. This meant the example hooks were executable by default, causing great confusion among new Git users because all the hooks ran when none should have.

Because of this problem with Windows, newer versions of Git suffix each hook file name with *.sample* so it won't run even if it's executable. To enable the example hooks, you'll have to rename the appropriate scripts yourself.

If you aren't interested in the example hooks, it is perfectly safe to remove them from your repository: `rm .git/hooks/*`. You can always get them back by copying them from their home in the *templates* directory.

In addition to the template examples, there are more example hooks in Git's *contrib* directory, a portion of the Git source code. The supplemental files may also be installed along with Git on your system. On Debian and Ubuntu, for example, the contributed hooks are installed in */usr/share/doc/git-core/contrib/hooks*.

Creating Your First Hook

To explore how a hook works, let's create a new repository and install a simple hook. First, we create the repository and populate it with a few files:

```
$ mkdir hooktest

$ cd hooktest

$ git init
Initialized empty Git repository in .git/

$ touch a b c

$ git add a b c

$ git commit -m 'added a, b, and c'
Created initial commit 97e9cf8: added a, b, and c
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a
 create mode 100644 b
 create mode 100644 c
```

Next, let's create a `pre-commit` hook to prevent checking in changes that contain the word "broken." Using your favorite text editor, put the following in a file called *.git/hooks/pre-commit*:

```
#!/bin/bash
echo "Hello, I'm a pre-commit script!" >&2
if git diff --cached | grep '^\+' | grep -q 'broken'; then
        echo "ERROR: Can't commit the word 'broken'" >&2
        exit 1  # reject
fi
exit 0  # accept
```

The script generates a list of all differences about to be checked in, extracts the lines to be *added* (that is, those lines that begin with a + character), and scans those lines for the word "broken."

There are many ways to test for the word "broken," but most of the obvious ones result in subtle problems. I'm not talking about how to "test for the word 'broken'" but rather about how to find the text to be scanned for the word "broken."

For example, you might have tried the test:

```
if git ls-files | xargs grep -q 'broken'; then
```

or, in other words, search for the word "broken," in all files in the repository. But this approach has two problems. If someone else had already committed a file containing the word "broken," then this script would prevent all future commits (until you fix it), even if those commits are totally unrelated. Moreover, the Unix `grep` command has no way of knowing which files will actually be committed; if you add "broken" to file `b`, make an unrelated change to `a`, and then run `git commit a`, there's nothing wrong with your commit because you're not trying to commit `b`. However, a script with this test would reject it anyway.

If you write a `pre-commit` script that restricts what you're allowed to check in, it's almost certain that you'll need to bypass it someday. You can bypass the `pre-commit` hook either by using the `--no-verify` option to `git commit` or by temporarily disabling your hook.

Now that you've created the `pre-commit` hook, make sure it's executable:

```
$ chmod a+x .git/hooks/pre-commit
```

And now you can test that it works as expected:

```
$ echo "perfectly fine" >a

$ echo "broken" >b
```

```
# Try to commit all files, even a 'broken' one.
$ git commit -m "test commit -a" -a
Hello, I'm a pre-commit script!
ERROR: Can't commit the word 'broken'

# Selectively committing un-broken files works.
$ git commit -m "test only file a" a
Hello, I'm a pre-commit script!
Created commit 4542056: test
1 files changed, 1 insertions(+), 0 deletions(-)

# And committing 'broken' files won't work.
$ git commit -m "test only file b" b
Hello, I'm a pre-commit script!
ERROR: Can't commit the word 'broken'
```

Observe that even when a commit works, the `pre-commit` script still emits "Hello." This would be annoying in a real script, so you should use such messages only while debugging the script. Notice also that, when the commit is rejected, `git commit` doesn't print an error message; the only message is the one produced by the script. To avoid confusing the user, be careful always to print an error message from a "pre" script if it's going to return a nonzero ("reject") exit code.

Given those basics, let's talk about the different hooks you can create.

Available Hooks

As Git evolves, new hooks become available. To discover what hooks are available in your version of Git, run `git help hooks`. Also refer to the Git documentation to find all the command-line parameters as well as the input and output of each hook.

Commit-Related Hooks

When you run `git commit`, Git executes a process like that shown in Figure 15-1.

None of the commit hooks run for anything other than `git commit`. For example, `git rebase`, `git merge`, and `git am` don't run your commit hooks by default. (Those commands may run other hooks, though.) *However,* `git commit --amend` *does* run your commit hooks.

Each hook has its own purpose as follows:

- The `pre-commit` hook gives you the chance to immediately abort a commit if something is wrong with the content being committed. The `pre-commit` hook runs before the user is allowed to edit the commit message, so the user won't enter a commit message only to discover the changes are rejected. You can also use this hook to automatically modify the content of the commit.

```
pre-commit hook (unless --no-verify)
      |
      V
(prepare default commit message)
      |
      V
prepare-commit-msg hook
      |
      V
(let the user edit the commit message)
      |
      V
commit-msg hook (unless --no-verify)
      |
      V
(actually do the commit)
      |
      V
post-commit hook
```

Figure 15-1. Commit hook processing

- `prepare-commit-msg` lets you modify Git's default message before it is shown to the user. For example, you can use this to change the default commit message template.
- The `commit-msg` hook can validate or modify the commit message after the user edits it. For example, you can leverage this hook to check for spelling mistakes or reject messages with lines that exceed a certain maximum length.
- `post-commit` runs after the commit operation has finished. At this point, you can update a log file, send email, or trigger an autobuilder, for instance. Some people use this hook to automatically mark bugs as fixed if, say, the bug number is mentioned in the commit message. In real life, however, the `post-commit` hook is rarely useful, because the repository that you `git commit` in is rarely the one that you share with other people. (The `update` hook is likely more suitable.)

Patch-Related Hooks

When you run `git am`, Git executes a process like that shown in Figure 15-2.

Despite what you might expect from the names of the hooks shown in Figure 15-2, `git apply` does not run the `applypatch` hooks, only `git am` does. This is because `git apply` doesn't actually commit anything, so there's no reason to run any hooks.

- `applypatch-msg` examines the commit message attached to the patch and determines whether or not it's acceptable. For example, you can choose to reject a patch if it has no `Signed-off-by:` header. You can also modify the commit message at this point if desired.

```
applypatch-msg hook
          |
          V
(apply the patch)
          |
          V
pre-applypatch hook
          |
          V
(actually do the commit)
          |
          V
post-applypatch hook
```

Figure 15-2. Patch hook processing

- The `pre-applypatch` hook is somewhat misnamed, because this script actually runs *after* the patch is applied but before committing the result. This makes it exactly analogous to the `pre-commit` script when doing `git commit`, even though its name implies otherwise. In fact, many people choose to create a `pre-applypatch` script that runs `pre-commit`.
- `post-applypatch` is analogous to the `post-commit` script.

Push-Related Hooks

When you run `git push`, the *receiving end* of Git executes a process like the one shown in Figure 15-3.

```
(receive all new objects)
          |
          V
pre-receive hook
          |
          V
for each updated ref:
                 |
                 V
            update hook
                 |
                 V
            update ref
          |
          |
          V
post-receive hook
          |
          V
post-update hook
```

Figure 15-3. Receive hook processing

All push-related hooks run on the receiver, not the sender. Thus, the hook scripts that run are in the *.git/hooks* directory of the receiving repository, not the sending one. Output produced by remote hooks is still shown to the user doing the `git push`.

As you can see in the diagram, the very first step of `git push` is to transfer all the missing objects (blobs, trees, and commits) from your local repository to the remote one. There is no need for a hook during this process because all Git objects are identified by their unique SHA1 hash; your hook cannot modify an object because it would change the hash. Furthermore, there's no reason to reject an object, because `git gc` cleans up anyway if the object turns out to be unneeded.

Instead of manipulating the objects themselves, push-related hooks are called when it's time to update the *refs* (branches and tags).

- `pre-receive` receives a list of all the refs that are to be updated, including their new and old object pointers. The only thing the prereceive hook can do is accept or reject all the changes at once, which is of limited use. You might consider it a feature, though, because it enforces transactional integrity across branches. Yet, it's not clear why you'd need such a thing; if you don't like that behavior, use the `update` hook instead.
- The `update` hook is called exactly once for each ref being updated. The update hook can choose to accept or reject updates to individual branches, without affecting whether other branches are updated or not. Also for each update you can trigger an action such as closing a bug or sending an email acknowledgment. It's usually better to handle such notifications here than in a `post-commit` hook, because a commit is not really considered "final" until it's been pushed to a shared repository.
- Like the prereceive hook, `post-receive` receives a list of all the refs that have just been updated. Anything that `post-receive` can do could also be done by the `update` hook, but sometimes `post-receive` is more convenient. For example, if you want to send an update notification email message, `post-receive` can send just a single notification about all updates instead of a separate email for each update.
- Don't use the `post-update` hook. It has been superseded by the newer `post-receive` hook. (`post-update` knows what branches have changed but not what their old values were; this limited its usefulness.)

Other Local Repository Hooks

Finally, there are a few miscellaneous hooks, and by the time you read this there may be even more. (Again, you can find the list of available hooks quickly with the command `git help hooks`.)

- The `pre-rebase` hook runs when you attempt to rebase a branch. This is useful because it can stop you from accidentally running `git rebase` on a branch that shouldn't be rebased because it's already been published.
- `post-checkout` runs after you check out a branch or an individual file. For example, you can use this to automatically create empty directories (Git doesn't know how to track empty directories) or to set file permissions or Access Control List (ACLs) on checked out files (Git doesn't track ACLs). You might think of using this to modify files after checking them out—for example, to do RCS-style variable substitution—but it's not such a good idea because Git will think the files have been locally modified. For such a task, use smudge/clean filters instead.
- `post-merge` runs after you perform a merge operation. This is rarely used. If your `pre-commit` hook does some sort of change to the repository, you might need to use a `post-merge` script to do something similar.
- `pre-auto-gc` helps `git gc --auto` decide whether or not it's time to clean up. You can make `git gc --auto` skip its `git gc` task by returning nonzero from this script. This will rarely be needed, however.

Programming

Version Control with Git

Get up to speed on Git for tracking, branching, merging, and managing code revisions. Through a series of step-by-step tutorials, this practical guide takes you quickly from Git fundamentals to advanced techniques, and provides friendly yet rigorous advice for navigating the many functions of this open source version control system.

This thoroughly revised edition also includes tips for manipulating trees, extended coverage of the reflog and stash, and a complete introduction to the GitHub repository. Git lets you manage code development in a virtually endless variety of ways, once you understand how to harness the system's flexibility. This book shows you how.

- Learn how to use Git for several real-world development scenarios
- Gain insight into Git's common-use cases, initial tasks, and basic functions
- Use the system for both centralized and distributed version control
- Learn how to manage merges, conflicts, patches, and diffs
- Apply advanced techniques such as rebasing, hooks, and ways to handle submodules
- Interact with Subversion (SVN) repositories—including SVN to Git conversions
- Navigate, use, and contribute to open source projects through GitHub

Jon Loeliger, a software engineer at Freescale Semiconductor, Inc., works on open source projects such as Git, Linux, and U-Boot. He's given Git tutorials at conferences such as Linux World, and has contributed articles to *Linux Magazine*.

Matthew McCullough, VP of Training for GitHub, is a 15-year veteran of enterprise software development and an open source educator. Matthew is the creator of O'Reilly's Git Master Class series.

US $36.99 CAN $38.99
ISBN: 978-1-449-31638-9

9 781449 316389 53699