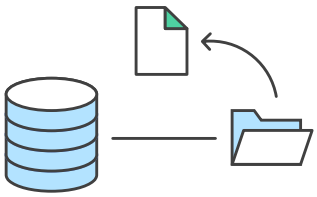
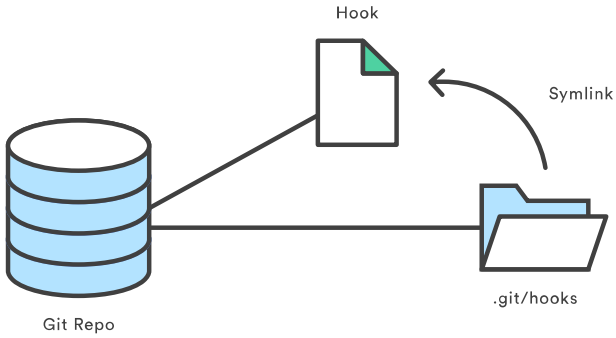


Git Hooks

Git hooks are scripts that run automatically every time a particular event occurs in a Git repository. They let you customize Git's internal behavior and trigger customizable actions at key points in the development life cycle.

Maintaining a hook using a symlink to version-controlled script



Common use cases for Git hooks include encouraging a commit policy, altering the project environment

hooks to automate or optimize virtually any aspect of your development workflow.

In this article, we'll start with a conceptual overview of how Git hooks work. Then, we'll survey some of the most popular hooks for use in both local and server-side repositories.

Conceptual Overview

All Git hooks are ordinary scripts that Git executes when certain events occur in the repository. This makes them very easy to install and configure.

Hooks can reside in either local or server-side repositories, and they are only executed in response to actions in that repository. We'll take a concrete look at categories of hooks later in this article. The configuration discussed in the rest of this section apply to both local and server-side hooks.

Installing Hooks

Hooks reside in the `.git/hooks` directory of every Git repository. Git automatically populates this directory with example scripts when you initialize a repository. If you take a look inside `.git/hooks`, you'll find the following files:

```
applypatch-msg.sample          pre-push.sample
commit-msg.sample              pre-rebase.sample
post-update.sample             prepare-commit-msg.sam
```

Tutorials

These represent most of the available hooks, but the `.sample` extension prevents them from executing by default. To "install" a hook, all you have to do is remove the `.sample` extension. Or, if you're writing a new script from scratch, you can simply add a new file matching one of the above filenames, minus the `.sample` extension.

As an example, try installing a simple `prepare-commit-msg` hook. Remove the `.sample` extension from this script, and add the following to the file:

```
#!/bin/sh

echo "# Please include a useful commit message!" :
```

Hooks need to be executable, so you may need to change the file permissions of the script if you're creating it from scratch. For example, to make sure that `prepare-commit-msg` is executable, you would run the following command:

```
chmod +x prepare-commit-msg
```

You should now see this message in place of the default commit message every time you run `git commit`. We'll take a closer look at how this actually works in the Prepare Commit Message section. For now, let's just revel in the fact that we can customize some of Git's internal functionality.

The built-in sample scripts are very useful references, as they document the parameters that are passed in to

Tutorials

Scripting Languages

The built-in scripts are mostly shell and PERL scripts, but you can use any scripting language you like as long as it can be run as an executable. The shebang line (`#!/bin/sh`) in each script defines how your file should be interpreted. So, to use a different language, all you have to do is change it to the path of your interpreter.

For instance, we can write an executable Python script in the `prepare-commit-msg` file instead of using shell commands. The following hook will do the same thing as the shell script in the previous section.

```
#!/usr/bin/env python

import sys, os

commit_msg_filepath = sys.argv[1]
with open(commit_msg_filepath, 'w') as f:
    f.write("# Please include a useful commit mess
```

Notice how the first line changed to point to the Python interpreter. And, instead of using `$1` to access the first argument passed to the script, we used `sys.argv[1]` (again, more on this in a moment).

This is a very powerful feature for Git hooks because it lets you work in whatever language you're most comfortable with.

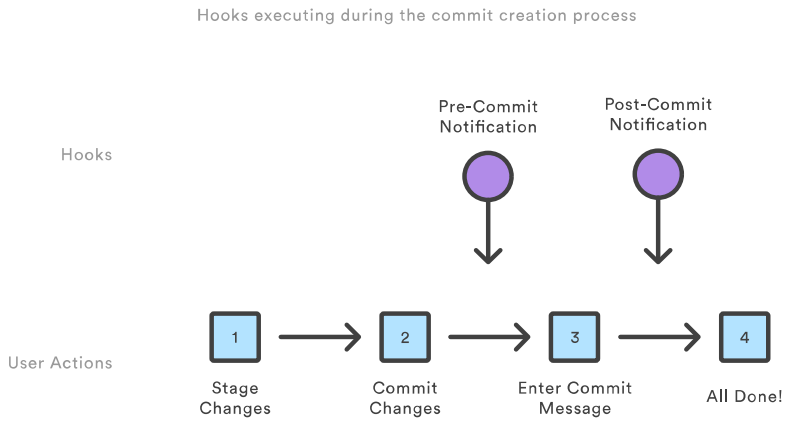
Scope of Hooks

Hooks are local to any given Git repository, and they are *not* copied over to the new repository when you run

Tutorials

This has an important impact when configuring hooks for a team of developers. First, you need to find a way to make sure hooks stay up-to-date amongst your team members. Second, you can't force developers to create commits that look a certain way—you can only encourage them to do so.

Maintaining hooks for a team of developers can be a little tricky because the `.git/hooks` directory isn't cloned with the rest of your project, nor is it under version control. A simple solution to both of these problems is to store your hooks in the actual project directory (above the `.git` directory). This lets you edit them like any other version-controlled file. To install the hook, you can either create a symlink to it in `.git/hooks`, or you can simply copy and paste it into the `.git/hooks` directory whenever the hook is updated.



As an alternative, Git also provides a Template Directory mechanism that makes it easier to install hooks automatically. All of the files and directories contained in this template directory are copied into the `.git` directory every time you use `git init` or `git clone`.

or not they actually use a hook. With this in mind, it's best to think of Git hooks as a convenient developer tool rather than a strictly enforced development policy.

That said, it is possible to reject commits that do not conform to some standard using server-side hooks. We'll talk more about this later in the article.

Local Hooks

Local hooks affect only the repository in which they reside. As you read through this section, remember that each developer can alter their own local hooks, so you can't use them as a way to enforce a commit policy. They can, however, make it much easier for developers to adhere to certain guidelines.

In this section, we'll be exploring 6 of the most useful local hooks:

- `pre-commit`
- `prepare-commit-msg`
- `commit-msg`
- `post-commit`
- `post-checkout`
- `pre-rebase`

The first 4 hooks let you plug into the entire commit life

Tutorials

All of the `pre-` hooks let you alter the action that's about to take place, while the `post-` hooks are used only for notifications.

We'll also see some useful techniques for parsing hook arguments and requesting information about the repository using lower-level Git commands.

Pre-Commit

The `pre-commit` script is executed every time you run `git commit` before Git asks the developer for a commit message or generates a commit object. You can use this hook to inspect the snapshot that is about to be committed. For example, you may want to run some automated tests that make sure the commit doesn't break any existing functionality.

No arguments are passed to the `pre-commit` script, and exiting with a non-zero status aborts the entire commit. Let's take a look at a simplified (and more verbose) version of the built-in `pre-commit` hook. This script aborts the commit if it finds any whitespace errors, as defined by the `git diff-index` command (trailing whitespace, lines with only whitespace, and a space followed by a tab inside the initial indent of a line are considered errors by default).

```
#!/bin/sh

# Check if this is the initial commit
if git rev-parse --verify HEAD >/dev/null 2>&1
then
```

Tutorials


```
fi

# Use git diff-index to check for whitespace erro
echo "pre-commit: Testing for whitespace errors..."
if ! git diff-index --check --cached $against
then
    echo "pre-commit: Aborting commit due to whit
    exit 1
else
    echo "pre-commit: No whitespace errors :)"
    exit 0
fi
```

In order to use `git diff-index`, we need to figure out which commit reference we're comparing the index to. Normally, this is `HEAD`; however, `HEAD` doesn't exist when creating the initial commit, so our first task is to account for this edge case. We do this with `git rev-parse --verify`, which simply checks whether or not the argument (`HEAD`) is a valid reference. The `>/dev/null 2>&1` portion silences any output from `git rev-parse`. Either `HEAD` or an empty commit object is stored in the `against` variable for use with `git diff-index`. The `4b825d...` hash is a magic commit ID that represents an empty commit.

The `git diff-index --cached` command compares a commit against the index. By passing the `--check` option, we're asking it to warn us if the changes introduces whitespace errors. If it does, we abort the commit by returning an exit status of `1`, otherwise we exit with `0` and the commit workflow continues as normal.

This is just one example of the `pre-commit` hook. It happens to use existing Git commands to run tests on the changes introduced by the proposed commit, but

Tutorials

Prepare Commit Message

The `prepare-commit-msg` hook is called after the `pre-commit` hook to populate the text editor with a commit message. This is a good place to alter the automatically generated commit messages for squashed or merged commits.

One to three arguments are passed to the `prepare-commit-msg` script:

1. The name of a temporary file that contains the message. You change the commit message by altering this file in-place.
2. The type of commit. This can be `message` (`-m` or `-F` option), `template` (`-t` option), `merge` (if the commit is a merge commit), or `squash` (if the commit is squashing other commits).
3. The SHA1 hash of the relevant commit. Only given if `-c`, `-C`, or `--amend` option was given.

As with `pre-commit`, exiting with a non-zero status aborts the commit.

We already saw a simple example that edited the commit message, but let's take a look at a more useful script. When using an issue tracker, a common convention is to address each issue in a separate branch. If you include the issue number in the branch name, you can write a `prepare-commit-msg` hook to automatically include it in each commit message on that

Tutorials


```
import sys, os, re
from subprocess import check_output

# Collect the parameters
commit_msg_filepath = sys.argv[1]
if len(sys.argv) > 2:
    commit_type = sys.argv[2]
else:
    commit_type = ''
if len(sys.argv) > 3:
    commit_hash = sys.argv[3]
else:
    commit_hash = ''

print "prepare-commit-msg: File: %s\nType: %s\nHas

# Figure out which branch we're on
branch = check_output(['git', 'symbolic-ref', '--sh
print "prepare-commit-msg: On branch '%s'" % branc

# Populate the commit message with the issue #, i
if branch.startswith('issue-'):
    print "prepare-commit-msg: Oh hey, it's an iss
    result = re.match('issue-(.*)', branch)
    issue_number = result.group(1)

    with open(commit_msg_filepath, 'r+') as f:
        content = f.read()
        f.seek(0, 0)
        f.write("ISSUE-%s %s" % (issue_number, con
```

First, the above `prepare-commit-msg` hook shows you how to collect all of the parameters that are passed to the script. Then, it calls `git symbolic-ref --short HEAD` to get the branch name that corresponds to `HEAD`. If this branch name starts with `issue-`, it re-writes the commit message file contents to include the issue number in the first line. So, if your branch name is `issue-224`, this will generate the following commit message.

```
ISSUE-224

# Please enter the commit message for your change
# with '#' will be ignored, and an empty message
# On branch issue-224
```

One thing to keep in mind when using `prepare-commit-msg` is that it runs even when the user passes in a message with the `-m` option of `git commit`. This means that the above script will automatically insert the `ISSUE-[#]` string without letting the user edit it. You can handle this case by seeing if the 2nd parameter (`commit_type`) is equal to `message`.

However, without the `-m` option, the `prepare-commit-msg` hook does allow the user to edit the message after its generated, so this is really more of a convenience script than a way to enforce a commit message policy. For that, you need the `commit-msg` hook discussed in the next section.

Commit Message

The `commit-msg` hook is much like the `prepare-commit-msg` hook, but it's called *after* the user enters a commit message. This is an appropriate place to warn developers that their message doesn't adhere to your team's standards.

The only argument passed to this hook is the name of the file that contains the message. If it doesn't like the message that the user entered, it can alter this file in-place (just like with `prepare-commit-msg`) or it can abort the commit entirely by exiting with a non-zero status.

For example, the following script checks to make sure that the user didn't delete the `ISSUE-[#]` string that was automatically generated by the `prepare-commit-msg`

Tutorials


```
import sys, os, re
from subprocess import check_output

# Collect the parameters
commit_msg_filepath = sys.argv[1]

# Figure out which branch we're on
branch = check_output(['git', 'symbolic-ref', '--sh
print "commit-msg: On branch '%s'" % branch

# Check the commit message if we're on an issue b
if branch.startswith('issue-'):
    print "commit-msg: Oh hey, it's an issue branc
    result = re.match('issue-(.*)', branch)
    issue_number = result.group(1)
    required_message = "ISSUE-%s" % issue_number

    with open(commit_msg_filepath, 'r') as f:
        content = f.read()
        if not content.startswith(required_message
            print "commit-msg: ERROR! The commit r
            sys.exit(1)
```

While this script is called every time the user creates a commit, you should avoid doing much outside of checking the commit message. If you need to notify other services that a snapshot was committed, you should use the `post-commit` hook instead.

Post-Commit

The `post-commit` hook is called immediately after the `commit-msg` hook. It can't change the outcome of the `git commit` operation, so it's used primarily for notification purposes.

The script takes no parameters and its exit status does not affect the commit in any way. For most `post-commit` scripts, you'll want access to the commit that was just created. You can use `git rev-parse HEAD` to get the

Tutorials

For example, if you want to email your boss every time you commit a snapshot (probably not the best idea for most workflows), you could add the following `post-commit` hook.

```
#!/usr/bin/env python

import smtplib
from email.mime.text import MIMEText
from subprocess import check_output

# Get the git log --stat entry of the new commit
log = check_output(['git', 'log', '-1', '--stat', '

# Create a plaintext email message
msg = MIMEText("Look, I'm actually doing some work

msg['Subject'] = 'Git post-commit hook notification
msg['From'] = 'mary@example.com'
msg['To'] = 'boss@example.com'

# Send the message
SMTP_SERVER = 'smtp.example.com'
SMTP_PORT = 587

session = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
session.ehlo()
session.starttls()
session.ehlo()
session.login(msg['From'], 'secretPassword')

session.sendmail(msg['From'], msg['To'], msg.as_str
session.quit()
```

It's possible to use `post-commit` to trigger a local continuous integration system, but most of the time you'll want to be doing this in the `post-receive` hook. This runs on the server instead of the user's local machine, and it also runs every time *any* developer pushes their code. This makes it a much more appropriate place to perform your continuous integration.

Tutorials

`post-commit` hook, but it's called whenever you successfully check out a reference with `git checkout`. This is nice for clearing out your working directory of generated files that would otherwise cause confusion.

This hook accepts three parameters, and its exit status has no affect on the `git checkout` command.

1. The ref of the previous HEAD
2. The ref of the new HEAD
3. A flag telling you if it was a branch checkout or a file checkout. The flag will be `1` and `0`, respectively.

A common problem with Python developers occurs when generated `.pyc` files stick around after switching branches. The interpreter sometimes uses these `.pyc` instead of the `.py` source file. To avoid any confusion, you can delete all `.pyc` files every time you check out a new branch using the following `post-checkout` script:

```
#!/usr/bin/env python

import sys, os, re
from subprocess import check_output

# Collect the parameters
previous_head = sys.argv[1]
new_head = sys.argv[2]
is_branch_checkout = sys.argv[3]

if is_branch_checkout == "0":
    print "post-checkout: This is a file checkout.
    sys.exit(0)

print "post-checkout: Deleting all '.pyc' files in
for root, dirs, files in os.walk('.'):
    for filename in files:
        ext = os.path.splitext(filename)[1]
```

Tutorials

Getting Started

Collaborating

Migrating to Git

Advanced Tips

Advanced Git Tutorials

Merging vs. Rebasing

Reset, Checkout, and Revert

Advanced Git log

Git Hooks

- Conceptual Overview
- Local Hooks
- Server-side Hooks
- Summary

Refs and the Reflog

The current working directory for hook scripts is always set to the root of the repository, so the `os.walk('.')` call iterates through every file in the repository. Then, we check its extension and delete it if it's a `.pyc` file.

You can also use the `post-checkout` hook to alter your working directory based on which branch you have checked out. For example, you might use a `plugins` branch to store all of your plugins outside of the core codebase. If these plugins require a lot of binaries that other branches do not, you can selectively build them only when you're on the `plugins` branch.

Pre-Rebase

The `pre-rebase` hook is called before `git rebase` changes anything, making it a good place to make sure something terrible isn't about to happen.

This hook takes 2 parameters: the upstream branch that the series was forked from, and the branch being rebased. The second parameter is empty when rebasing the current branch. To abort the rebase, exit with a non-zero status.

For example, if you want to completely disallow rebasing in your repository, you could use the following `pre-rebase` script:

```
#!/bin/sh

# Disallow all rebasing
echo "pre-rebase: Rebasing is dangerous. Don't do
exit 1
```

```
pre-rebase: Rebasing is dangerous. Don't do it.
The pre-rebase hook refused to rebase.
```

For a more in-depth example, take a look at the included `pre-rebase.sample` script. This script is a little more intelligent about when to disallow rebasing. It checks to see if the topic branch that you're trying to rebase has already been merged into the `next` branch (which is assumed to be the mainline branch). If it has, you're probably going to get into trouble by rebasing it, so the script aborts the rebase.

Server-side Hooks

Server-side hooks work just like local ones, except they reside in server-side repositories (e.g., a central repository, or a developer's public repository). When attached to the official repository, some of these can serve as a way to enforce policy by rejecting certain commits.

There are 3 server-side hooks that we'll be discussing in the rest of this article:

- `pre-receive`
- `update`
- `post-receive`

All of these hooks let you react to different stages of the `git push` process.

that these scripts don't return control of the terminal until they finish executing, so you should be careful about performing long-running operations.

Pre-Receive

The `pre-receive` hook is executed every time somebody uses `git push` to push commits to the repository. It should always reside in the *remote* repository that is the destination of the push, not in the originating repository.

The hook runs before any references are updated, so it's a good place to enforce any kind of development policy that you want. If you don't like who is doing the pushing, how the commit message is formatted, or the changes contained in the commit, you can simply reject it. While you can't stop developers from making malformed commits, you can prevent these commits from entering the official codebase by rejecting them with `pre-receive`.

The script takes no parameters, but each ref that is being pushed is passed to the script on a separate line on standard input in the following format:

```
<old-value> <new-value> <ref-name>
```

You can see how this hook works using a very basic `pre-receive` script that simply reads in the pushed refs and prints them out.

Tutorials


```
# Read in each ref that the user is trying to upd
for line in fileinput.input():
    print "pre-receive: Trying to push ref: %s" %

# Abort the push
# sys.exit(1)
```

Again, this is a little different than the other hooks because information is passed to the script via standard input instead of as command-line arguments. After placing the above script in the `.git/hooks` directory of a remote repository and pushing the `master` branch, you'll see something like the following in your console:

```
b6b36c697eb2d24302f89aa22d9170dfe609855b 85baa88c
```

You can use these SHA1 hashes, along with some lower-level Git commands, to inspect the changes that are going to be introduced. Some common use cases include:

- Rejecting changes that involve an upstream rebase
- Preventing non-fast-forward merges
- Checking that the user has the correct permissions to make the intended changes (mostly used for centralized Git workflows)

If multiple refs are pushed, returning a non-zero status from `pre-receive` aborts *all* of them. If you want to accept or reject branches on a case-by-case basis, you need to use the `update` hook instead.

Tutorials

works much the same way. It's still called before anything is actually updated, but it's called separately for each ref that was pushed. That means if the user tries to push 4 branches, `update` is executed 4 times. Unlike `pre-receive`, this hook doesn't need to read from standard input. Instead, it accepts the following 3 arguments:

1. The name of the ref being updated
2. The old object name stored in the ref
3. The new object name stored in the ref

This is the same information passed to `pre-receive`, but since `update` is invoked separately for each ref, you can reject some refs while allowing others.

```
#!/usr/bin/env python

import sys

branch = sys.argv[1]
old_commit = sys.argv[2]
new_commit = sys.argv[3]

print "Moving '%s' from %s to %s" % (branch, old_

# Abort pushing only this branch
# sys.exit(1)
```

The above `update` hook simply outputs the branch and the old/new commit hashes. When pushing more than one branch to the remote repository, you'll see the `print` statement execute for each branch.

push operation, making it a good place to perform notifications. For many workflows, this is a better place to trigger notifications than `post-commit` because the changes are available on a public server instead of residing only on the user's local machine. Emailing other developers and triggering a continuous integration system are common use cases for `post-receive`.

The script takes no parameters, but is sent the same information as `pre-receive` via standard input.

Summary

In this article, we learned how Git hooks can be used to alter internal behavior and receive notifications when certain events occur in a repository. Hooks are ordinary scripts that reside in the `.git/hooks` repository, which makes them very easy to install and customize.

We also looked at some of the most common local and server-side hooks. These let us plug in to the entire development life cycle. We now know how to perform customizable actions at every stage in the commit creation process, as well as the `git push` process. With a little bit of scripting knowledge, this lets you do virtually anything you can imagine with a Git repository.

Next up:

Refs and the Reflog

START NEXT TUTORIAL

Tutorials

Enter Your Email For Git News

Getting Started

Collaborating

Migrating to Git

Advanced Tips

Advanced Git Tutorials

Merging vs. Rebasing

Reset, Checkout, and Revert

Advanced Git log

Git Hooks

- Conceptual Overview
- Local Hooks
- Server-side Hooks
- Summary

Refs and the Reflog