如何使用Git 钩子来自动化开发和部署任务

Git是，近年来已经看到了巨大的收养一个非常强大和成熟的版本控制系统。其中一个Git的强大功能是它能够使用“挂钩”时，一定混帐事件发生时调用任意脚本的能力。在本指南中，我们将讨论混帐挂钩背后的总体思路，谈谈如何解决一些矛盾，并表现出一些实现，你可以在你的开发过程中使用。

 2014-08-12 17:04:54

## 介绍

版本控制已成为现代软件开发的中心要求。 它允许项目安全地跟踪更改，启用撤销，完整性检查和协作等好处。 在git的版本控制系统，特别是，已由于其分散式架构，并在它可以使和转让方之间变化的速度看到广泛采用在最近几年。 虽然git的工具套件提供了许多良好的实现的功能，最有用的特点之一就是它的灵活性。通过使用“hooks”系统，git允许开发人员和管理员通过指定git将根据不同的事件和操作调用的脚本来扩展功能。 在本指南中，我们将探讨git钩子的概念，并演示如何实现可以帮助您在自己独特的环境中自动执行任务的代码。我们将在本指南中使用Ubuntu 14.04服务器，但是任何可以运行git的系统都应该以类似的方式工作。

先决条件

在开始之前，你必须有git服务器上安装。 如果您在Ubuntu 14.04下面一起，你可以看看我们的指南[如何在Ubuntu 14.04安装的git](https://www.howtoing.com/how-to-install-git-on-ubuntu-14-04/)在这里。 你应该熟悉如何在一般意义上使用git。如果你需要一个介绍，该系列安装的一部分，所谓的[介绍Git的方法：安装，使用和分支](https://www.digitalocean.com/community/tutorial_series/introduction-to-git-installation-usage-and-branches) ，是一个良好的开端。 完成上述要求后，继续。

基本想法与Git钩

Git钩子是一个相当简单的概念，被实现来解决一个需要。在共享项目上开发软件，维护风格指南标准或部署软件（都是git常常涉及的情况）时，每次执行操作时都会执行重复性任务。 Git钩子是基于事件的。当您运行某些Git命令，该软件会检查hooks的Git仓库内的目录，看看是否有相关的脚本来运行。 一些脚本在执行操作之前运行，可用于确保代码符合标准，正确性检查或设置环境。其他脚本在事件之后运行，以便部署代码，重新建立正确的权限（Git不能很好地跟踪），等等。 使用这些功能，可以实施策略，确保一致性并控制您的环境，甚至可以处理部署任务。 书[临的Git](http://git-scm.com/book)斯科特查孔尝试不同类型的钩子分为两类。他将它们归类为：

* 客户端挂钩：在提交者的计算机上调用并执行的挂钩。这些又分为几个单独的类别：
  + 提交 - 工作流挂钩：提交挂钩用于规定在提交时应该采取的行动。它们用于运行完整性检查，预填充提交消息以及验证消息详细信息。您也可以使用此提交时提供通知。
  + 电子邮件工作流挂接：此类挂接包括在使用电子邮件修补程序时执行的操作。像Linux内核这样的项目使用电子邮件方法提交和审查补丁。这些类似于提交钩子，但可以由负责应用提交的代码的维护者使用。
  + 其他：其他客户端挂钩包括在合并，签出代码，重建基准，重写和清除回收时执行的挂钩。
* 服务器端钩子：这些钩子在用于接收推送的服务器上执行。一般来说，这将是一个项目的主要Git仓库。再次，Chacon将这些分为以下几类：
  + 预接收和后接收：这些在服务器上执行，接收推送以执行检查项目一致性并在推送后部署。
  + 更新：这类似于预接收，但是在逐个分支的基础上操作以在每个分支被接受之前执行代码。

这些分类有助于大致了解可以选择设置挂钩的事件。但是要真正了解这些项目是如何工作的，最好尝试并找出您要实现的解决方案。 某些钩子也带有参数。这意味着当git为钩子调用脚本时，它将传递一些相关数据，然后脚本可以使用这些数据来完成任务。完全，可用的钩子有： 

| **钩名称** | **被调用者** | **描述** | **参数（数字和说明）** |
| --- | --- | --- | --- |
| applypatch-msg | git am | 可以编辑提交消息文件，并且通常用于验证或主动将补丁的消息格式化为项目的标准。非零退出状态将中止提交。 | （1）包含建议提交消息的文件的名称 |
| 预应用补丁 | git am | 在应用补丁*之后*这实际上是调用，但更改提交*之前* 。以非零状态退出将使更改处于未提交状态。可以用于在实际提交更改之前检查树的状态。 | （没有） |
| 后应用补丁 | git am | 此挂接在应用和提交补丁后运行。因此，它不能中止进程，并且主要用于创建通知。 | （没有） |
| 预提交  pre-commit | git commit | 在获得建议的提交消息之前调用此挂钩。使用非零值退出将中止提交。它用于检查提交本身（而不是消息）。 | （没有） |
| 准备提交信息 prepare-commit-msg | git commit | 在接收到默认提交消息之后，在提交消息编辑器启动之前调用。非零退出将中止提交。这用于以不能抑制的方式编辑消息。 | （1〜3）与提交信息的文件名，提交信息的来源（ message ， template ， merge ， squash ，或commit ），并提交SHA-1（在现有承诺工作时）。 |
| 提交信息 commit-msg | git commit | 可以用于在消息被编辑之后调整消息，以确保符合标准或根据任何标准拒绝。如果它以非零值退出，它可以中止提交。 | （1）保存建议消息的文件。 |
| 提交后 post-commit | git commit | 在实际提交后调用。正因为如此，它不能中断提交。它主要用于允许通知。 | （没有） |
| 预变基 pre-rebase | git rebase | 在重新分支分支时调用。主要用于停止rebase，如果不可取。 | （1或2）从其分叉的上游，分支被重新基址（当重新基址电流时不设置） |
| 切换后 post-checkout | git checkout和git clone | 当检出更新worktree后或称为运行git clone 。它主要用于验证条件，显示差异，并在必要时配置环境。 | （3）指示其是分支检出（1）还是文件检出（0）的新HEAD的ref的前一HEAD的ref， |
| 后合并 | git merge或git pull | 在合并后调用。因此，它不能中止合并。可以用于保存或应用权限或Git不处理的其他类型的数据。 | （1）指示合并是否是壁球的标志。 |
| 预推 | git push | 在推送到远程之前调用。除了参数之外，由空格分隔的附加信息以“<local ref> <local sha1> <remote ref> <remote sha1>”的形式通过stdin传入。解析输入可以为您提供可用于检查的其他信息。例如，如果本地sha1是40个0长，推是一个删除，如果远程sha1是40个0，它是一个新的分支。这可以用于做许多比较推送ref到目前在那里。非零退出状态将中止推送。 | （2）目的地远程的名称，目的地远程的位置 |
| 预接收 pre-receive | git-receive-pack上的远程回购 | 这在更新推送的引用之前在远程仓库上被调用。非零状态将中止该过程。虽然它不接收参数，但是对于每个ref，以“<old-value> <new-value> <ref-name>”的形式通过stdin传递一个字符串。 | （没有） |
| 更新 update | git-receive-pack上的远程回购 | 这在远程仓库上运行一次，每次推送，而不是每次推送一次。非零状态将中止该过程。例如，这可以用于确保所有提交仅仅是快进。 | （3）要更新的引用的名称，旧对象名称，新对象名称 |
| 接收后 post-receive | git-receive-pack上的远程回购 | 这是在所有引用更新后在推送时在远程运行。它不接受参数，但通过stdin以“<old-value> <new-value> <ref-name>”的形式接收信息。因为它在更新后被调用，所以它不能中止进程。 | （没有） |
| 更新后 | git-receive-pack上的远程回购 | 这只有在所有的ref被推送之后才运行一次。在这方面它类似于后接收钩子，但不接收旧的或新的值。它主要用于实现推送引用的通知。 | （？）每个包含其名称的推送引用的参数 |
| 预先自动化 | git gc --auto | 用于在自动清理repos之前进行一些检查。 | （没有） |
| 后重写 | git commit --amend ， git-rebase | 当git命令重写已经提交的数据时，将调用此函数。除了参数，它以“<old-sha1> <new-sha1>”的形式接收stdin中的字符串。 | （1）调用它的命令的名称（ amend或rebase ） |

现在您已经掌握了所有这些一般信息，我们可以演示如何在几种情况下实现这些。

设置存储库

要开始，我们将在我们的主目录中创建一个新的，空的存储库。我们将调用这个proj 。

mkdir ~/proj

cd ~/proj

git init

Initialized empty Git repository in /home/demo/proj/.git/

现在，我们在一个git控制的目录的空工作目录。之前，我们做任何事情，让我们跳进存储在名为隐藏文件的库.git这个目录中：

cd .git

ls -F

branches/ config description HEAD hooks/ info/ objects/ refs/

我们可以看到很多文件和目录。一个我们感兴趣的是hooks目录：

cd hooks

ls -l

total 40

-rwxrwxr-x 1 demo demo 452 Aug 8 16:50 applypatch-msg.sample

-rwxrwxr-x 1 demo demo 896 Aug 8 16:50 commit-msg.sample

-rwxrwxr-x 1 demo demo 189 Aug 8 16:50 post-update.sample

-rwxrwxr-x 1 demo demo 398 Aug 8 16:50 pre-applypatch.sample

-rwxrwxr-x 1 demo demo 1642 Aug 8 16:50 pre-commit.sample

-rwxrwxr-x 1 demo demo 1239 Aug 8 16:50 prepare-commit-msg.sample

-rwxrwxr-x 1 demo demo 1352 Aug 8 16:50 pre-push.sample

-rwxrwxr-x 1 demo demo 4898 Aug 8 16:50 pre-rebase.sample

-rwxrwxr-x 1 demo demo 3611 Aug 8 16:50 update.sample

我们可以在这里看到一些东西。首先，我们可以看到，每个这些文件都标记为可执行。由于这些脚本只是名称叫，它们必须是可执行文件和他们的第一行必须是一个[家当神奇的数字](http://en.wikipedia.org/wiki/Shebang_(Unix)#Magic_number)参考调用正确的脚本解释器。最常见的，这些是脚本语言，如bash，perl，python等。 您可能会注意到的第二件事是所有文件的结尾.sample 。 这是因为git只是看看文件名，当试图找到要执行的钩子文件。 偏离脚本的名称git正在寻找基本上禁用脚本。 为了使任何此目录中的脚本，我们将不得不去掉.sampleStapling。 让我们回到我们的工作目录：

cd ../..

第一个示例：使用提交后挂钩部署到本地Web服务器

我们的第一个例子将使用post-commit钩子向您展示如何部署到本地Web服务器，只要提交而成。这不是你将用于生产环境的钩子，但它允许我们演示一些重要的，几乎没有文档的项目，你应该知道什么时候使用钩子。 首先，我们将安装Apache Web服务器来演示：

sudo apt-get update

sudo apt-get install apache2

为了让我们的脚本来修改在Web根目录/var/www/html （这是在Ubuntu 14.04的文档根目录。根据需要修改），我们需要有写权限。让我们给这个目录的正常用户所有权。您可以输入以下命令：

sudo chown -R `whoami`:`id -gn` /var/www/html

现在，在我们的项目目录，让我们创建一个index.html文件：

cd ~/proj

nano index.html

在里面，我们可以添加一点点HTML来演示这个想法。它不必复杂：

<h1>Here is a title!</h1>

<p>Please deploy me!</p>

添加新文件以告诉git跟踪文件：

git add .

现在， *在*你提交*之前* ，我们要建立我们post-commit挂钩库。 创建在这个文件中.git/hooks项目目录：

vim .git/hooks/post-commit

在我们讨论在这个文件中放置什么之前，我们需要了解一下当运行钩子时git如何设置环境。

除了环境变量与Git钩

在我们开始我们的脚本之前，我们需要了解一下当调用hook时环境变量git设置的位置。为了让我们的脚本功能，我们最终需要取消设置呼叫时的git套环境变量post-commit钩子。 这是一个非常重要的内部化，如果你希望写Git钩子，以可靠的方式运行。 Git根据调用的钩子设置不同的环境变量。这意味着git从中提取信息的环境将根据钩子而有所不同。 第一个问题是，如果你不知道正在自动设置的变量，它可能使你的脚本环境非常不可预测。第二个问题是，设置的变量在git自己的文档中几乎完全不存在。 幸运的是，马克Longair开发[用于测试每个组的git变量的方法](http://longair.net/blog/2011/04/09/missing-git-hooks-documentation/)运行这些挂钩时。它涉及到以下内容在各种git钩子脚本：

#!/bin/bash

echo Running $BASH\_SOURCE

set | egrep GIT

echo PWD is $PWD

他的网站上的信息是从2011年使用git版本1.7.1，所以有一些改变。在撰写本文时，在2014年8月，当前版本的git在Ubuntu 14.04是1.9.1。 这个版本的git的测试结果如下（包括运行每个钩子时由git看到的工作目录）。用于测试的本地工作目录是/home/demo/test\_hooks和裸远程（在必要时）为/home/demo/origin/test\_hooks.git ：

* **钩** ： applypatch-msg ， pre-applypatch ， post-applypatch
  + **环境变量** ：
  + GIT\_AUTHOR\_DATE='Mon, 11 Aug 2014 11:25:16 -0400'
  + GIT\_AUTHOR\_EMAIL=demo@example.com
  + GIT\_AUTHOR\_NAME='Demo User'
  + GIT\_INTERNAL\_GETTEXT\_SH\_SCHEME=gnu
  + GIT\_REFLOG\_ACTION=am
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： pre-commit ， prepare-commit-msg ， commit-msg ， post-commit
  + **环境变量** ：
  + GIT\_AUTHOR\_DATE='@1407774159 -0400'
  + GIT\_AUTHOR\_EMAIL=demo@example.com
  + GIT\_AUTHOR\_NAME='Demo User'
  + GIT\_DIR=.git
  + GIT\_EDITOR=:
  + GIT\_INDEX\_FILE=.git/index
  + GIT\_PREFIX=
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： pre-rebase
  + **环境变量** ：
  + GIT\_INTERNAL\_GETTEXT\_SH\_SCHEME=gnu
  + GIT\_REFLOG\_ACTION=rebase
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： post-checkout
  + **环境变量** ：
  + GIT\_DIR=.git
  + GIT\_PREFIX=
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： post-merge
  + **环境变量** ：
  + GITHEAD\_4b407c...
  + GIT\_DIR=.git
  + GIT\_INTERNAL\_GETTEXT\_SH\_SCHEME=gnu
  + GIT\_PREFIX=
  + GIT\_REFLOG\_ACTION='pull other master'
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： pre-push
  + **环境变量** ：
  + GIT\_PREFIX=
  + **工作目录** ： /home/demo/test\_hooks
* **钩** ： pre-receive ， update ， post-receive ， post-update
  + **环境变量** ：
  + GIT\_DIR=.
  + **工作目录** ： /home/demo/origin/test\_hooks.git
* **钩** ： pre-auto-gc
  + （未知，因为这很难触发可靠）
* **钩** ： post-rewrite
  + **环境变量** ：
  + GIT\_AUTHOR\_DATE='@1407773551 -0400'
  + GIT\_AUTHOR\_EMAIL=demo@example.com
  + GIT\_AUTHOR\_NAME='Demo User'
  + GIT\_DIR=.git
  + GIT\_PREFIX=
  + **工作目录** ： /home/demo/test\_hooks

这些变量暗示了git如何看待它的环境。我们将使用上面有关变量的信息，以确保我们的脚本正确地考虑其环境。

回到脚本

现在，您对环境，到位的类型的想法（查看为设置的变量post-commit钩子），我们就可以开始我们的脚本。 因为git钩子是标准脚本，我们需要告诉git使用什么解释器：

#!/bin/bash

之后，我们只需要使用git本身，在提交之后将最新版本的存储库解压缩到我们的web目录中。为此，我们应将工作目录设置为Apache的文档根目录。我们还应该将我们的git目录设置为repo。 我们将要强制这个事务，以确保这是成功的每次，即使当前在工作目录之间有冲突。它应该看起来像这样：

#!/bin/bash

git --work-tree=/var/www/html --git-dir=/home/demo/proj/.git checkout -f

在这一点上，我们差不多完成了。但是，我们需要寻找额外收于那些每个时间设置环境变量post-commit钩子被调用。 特别是， GIT\_INDEX\_FILE被设定为.git/index。 此路径是相对于工作目录，在这种情况下是/var/www/html 。 由于git索引在此位置不存在，如果我们保持原样，脚本将失败。 为了避免这种情况，我们可以手动*取消设置*变量，这将导致GIT中有关搜索到回购目录，因为它通常不会。 我们需要添加此结帐线**以上** ：

#!/bin/bash

unset GIT\_INDEX\_FILE

git --work-tree=/var/www/html --git-dir=/home/demo/proj/.git checkout -f

这些类型的冲突是为什么git钩子问题有时很难诊断。你必须知道git如何构建它正在工作的环境。 完成这些更改后，保存并关闭文件。 因为这是一个常规的脚本文件，我们需要使其可执行：

chmod +x .git/hooks/post-commit

现在，我们终于准备好提交我们在git repo中所做的更改。确保您回到正确的目录，然后提交更改：

cd ~/proj

git commit -m "here we go..."

现在，如果你在你的浏览器访问您的服务器的域名或IP地址，你应该看到index.html你创建的文件：

http://server\_domain\_or\_IP

  
如您所见，我们最近的更改在提交后已自动推送到我们的Web服务器的文档根目录。我们可以进行一些额外的更改，以显示它适用于每个提交：

echo "<p>Here is a change.</p>" >> index.html

git add .

git commit -m "First change"

当您刷新浏览器时，应该立即看到您应用的新更改：   
  
如你所见，这种类型的设置可以使本地测试更容易。但是，你几乎不想在生产环境中提交时进行发布。在测试你的代码并确保它已经准备好后，推送更安全。

使用Git Hooks部署到单独的生产服务器

在下一个示例中，我们将演示更好地更新生产服务器的方法。我们可以通过使用push-to-deploy模型来更新我们的web服务器，每当我们推送到一个裸git仓库。 我们可以使用我们已经设置的相同的服务器作为我们的开发机器。这是我们将在那里做我们的工作。我们将能够在每次提交后看到我们的更改。 在我们的生产机器上，我们将设置另一个Web服务器，一个裸git存储库，我们将推送更改，以及一个git钩子，每当接收到推送时执行。使用sudo权限作为普通用户完成以下步骤。

设置生产服务器后接收挂接

在生产服务器上，通过安装Web服务器开始：

sudo apt-get update

sudo apt-get install apache2

同样，我们应该将文档根目录的所有权授予我们正在操作的用户：

sudo chown -R `whoami`:`id -gn` /var/www/html

我们需要记住在这台机器上安装git：

sudo apt-get install git

现在，我们可以在用户的主目录中创建一个目录来保存存储库。然后，我们可以进入该目录并初始化一个裸存储库。裸存储库没有工作目录，并且更适合您不会直接使用的服务器：

mkdir ~/proj

cd ~/proj

git init --bare

由于这是一个纯仓库，没有工作目录和所有位于文件.git在常规设置在主目录本身。 我们需要创建另一个git钩子。这一次，我们感兴趣的是post-receive挂机，这是服务器接收上运行git push 。在编辑器中打开此文件：

nano hooks/post-receive

再次，我们需要从识别我们正在写的脚本类型开始。在这之后，我们就可以打出来，我们在我们使用相同的checkout命令post-commit文件，修改为使用这台机器上的路径：

#!/bin/bash

git --work-tree=/var/www/html --git-dir=/home/demo/proj checkout -f

由于这是一个纯仓库中， --git-dir应该指向回购的顶级目录。其余的是相当类似。 但是，我们需要为此脚本添加一些额外的逻辑。如果我们不小心推test-feature跳转到本服务器，我们不希望进行部署。 我们要确保我们只将要部署的master分支。 对于post-receive挂钩，你可能已经在表中早就注意到了Git通过了旧版本的提交哈希值，新修订的提交哈希值，以及正在推进的标准输入到脚本参考。我们可以使用它来检查引用是否是主分支。 首先，我们需要读取标准输入。对于每个被推送的ref，将三个信息（旧rev，新rev，ref）馈送到脚本，用空格分隔，作为标准输入。我们可以用阅读while循环包围git命令：

#!/bin/bash

while read oldrev newrev ref

do

git --work-tree=/var/www/html --git-dir=/home/demo/proj checkout -f

done

所以现在，我们将根据正在推送的内容设置三个变量。对于主分支推动下， ref对象将包含的东西，看起来像refs/heads/master 。 我们可以检查服务器是否接受裁判通过使用具有这种格式if结构：

#!/bin/bash

while read oldrev newrev ref

do

if [[ $ref =~ .\*/master$ ]];

then

git --work-tree=/var/www/html --git-dir=/home/demo/proj checkout -f

fi

done

对于服务器端钩子，git实际上可以将消息传递回客户端。发送到标准输出的任何内容都将重定向到客户端。这使我们有机会明确通知用户已做出了哪些决定。 我们应该添加一些文本来描述检测到的情况，以及采取了什么行动。我们应该添加一个else块来通知用户时被成功接收非主分支，即使行动不会触发部署：

#!/bin/bash

while read oldrev newrev ref

do

if [[ $ref =~ .\*/master$ ]];

then

echo "Master ref received. Deploying master branch to production..."

git --work-tree=/var/www/html --git-dir=/home/demo/proj checkout -f

else

echo "Ref $ref successfully received. Doing nothing: only the master branch may be deployed on this server."

fi

done

完成后，保存并关闭文件。 记住，我们必须使脚本可执行以使挂钩工作：

chmod +x hooks/post-receive

现在，我们可以在我们的客户端上设置对这个远程服务器的访问。

在客户端计算机上配置远程服务器

回到你的客户端（开发）机器，回到你的项目的工作目录：

cd ~/proj

中，添加远程服务器作为一个远程叫做production 。您需要知道您在生产服务器上使用的用户名及其IP地址或域名。您还需要知道您设置的与用户主目录相关的裸存储库的位置。 您键入的命令应如下所示：

git remote add production demo@server\_domain\_or\_IP:proj

让我们将当前的master分支推送到我们的生产服务器：

git push production master

如果未配置SSH密钥，则可能需要输入生产服务器用户的密码。你应该看到这样的东西：

Counting objects: 8, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

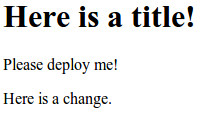
Writing objects: 100% (4/4), 473 bytes | 0 bytes/s, done.

Total 4 (delta 0), reused 0 (delta 0)

remote: Master ref received. Deploying master branch...

To demo@107.170.14.32:proj

009183f..f1b9027 master -> master

正如你所看到的，从我们的文本post-receive钩是命令的输出。如果我们在我们的网络浏览器中访问我们的生产服务器的域名或IP地址，我们应该看到我们的项目的当前版本：   
  
看起来钩子已经成功地将我们的代码推送到生产一旦它收到信息。 现在，让我们测试一些新的代码。回到开发机器上，我们将创建一个新的分支来保存我们的更改。这样，我们可以确保一切准备就绪，然后我们部署到生产。 建立一个新的分支称为test\_feature和检查通过键入新的分支出来：

git checkout -b test\_feature

我们现在是在工作test\_feature分支。 让我们，我们*可能*要移动到生产的转变。我们将把它交给这个分支：

echo "<h2>New Feature Here</h2>" >> index.html

git add .

git commit -m "Trying out new feature"

此时，如果您转到开发机器的IP地址或域名，您应该会看到您的更改显示：   
  
这是因为我们的开发机器仍在每次提交时重新部署。这个工作流程非常适合在将变更移动到生产之前测试变更。 我们可以把我们的test\_feature分行我们的远程生产服务器：

git push production test\_feature

你应该看到我们的其他邮件post-receive钩的输出：

Counting objects: 5, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 301 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

remote: Ref refs/heads/test\_feature successfully received. Doing nothing: only the master branch may be deployed on this server

To demo@107.170.14.32:proj

83e9dc4..5617b50 test\_feature -> test\_feature

如果你再次在浏览器中检出生产服务器，你应该看到没有什么改变。这是我们所期望的，因为我们推动的变化不在主分支。 现在我们已经在开发机器上测试了我们的变化，我们确信我们希望将这个特性合并到我们的主分支中。我们可以检出我们的master分支，我们合并test\_feature我们的开发机器上的分支：

git checkout master

git merge test\_feature

现在，您已将新功能合并到主分支中。推送到生产服务器将部署我们的更改：

git push production master

如果我们检查我们的生产服务器的域名或IP地址，我们将看到我们的更改：   
  
使用这个工作流，我们可以有一个开发机器，将立即显示任何已提交的更改。每当我们推动主分支时，生产机器将被更新。

结论

如果你遵循了这么远，你应该能够看到git钩子可以帮助自动化你的一些任务的不同的方式。他们可以帮助您部署代码，或通过拒绝不合规的更改或提交消息来帮助您保持质量标准。 虽然git钩子的效用很难说，实际的实现可能很难掌握和沮丧的故障排除。练习实现各种配置，尝试解析参数和标准输入，以及跟踪Git如何构造钩子环境将在教你如何编写有效的钩子方面有很大的帮助。从长远来看，时间投资通常是值得的，因为它可以轻松地在项目生命的过程中为您和您的团队加载手工工作。