**如何搭建你自己的 Git 服务器**

|  |  |
| --- | --- |
| 导读 | 现在我们将要学习如何搭建 git 服务器，如何编写自定义的 Git 钩子来在特定的事件触发相应的动作（例如通知），或者是发布你的代码到一个站点。 |

直到现在，我们主要讨论的还是以一个使用者的身份与 Git 进行交互。这篇文章中我将讨论 Git 的管理，并且设计一个灵活的 Git 框架。你可能会觉得这听起来是 “高阶 Git 技术” 或者 “只有狂热粉才能阅读”的一句委婉的说法，但是事实是这里面的每个任务都不需要很深的知识或者其他特殊的训练，就能基本理解 Git 的工作原理，有可能需要一丁点关于 Linux 的知识。

共享 Git 服务器

创建你自己的共享 Git 服务器意外地简单，而且在很多情况下，遇到的这点麻烦是完全值得的。不仅仅是因为它保证你有权限查看自己的代码，它还可以通过扩展为 Git 的使用敞开了一扇大门，例如个人 Git 钩子、无限制的数据存储、和持续集成与分发（CI & CD）。

如果你知道如何使用 Git 和 SSH，那么你已经知道怎么创建一个 Git 服务器了。Git 的设计方式，就是让你在创建或者 clone 一个仓库的时候，就完成了一半服务器的搭建。然后允许用 SSH 访问仓库，而且任何有权限访问的人都可以使用你的仓库作为 clone 的新仓库的基础。

但是，这是一个小的点对点环境。按照一些方案你可以创建一些带有同样的功能的设计优良的 Git 服务器，同时有更好的拓展性。

首要之事：确认你的用户们，现在的用户以及之后的用户都要考虑。如果你是唯一的用户那么没有任何改动的必要。但是如果你试图邀请其他的代码贡献者使用，那么你应该允许一个专门的分享系统用户给你的开发者们。

假定你有一个可用的服务器（如果没有，这不成问题，Git 会帮忙解决，CentOS 的 树莓派 3 是个不错的开始），然后第一步就是只允许使用 SSH 密钥认证的 SSH 登录。这比使用密码登录安全得多，因为这可以免于暴力破解，也可以通过直接删除用户密钥而禁用用户。

一旦你启用了 SSH 密钥认证，创建 gituser 用户。这是给你的所有授权的用户们的公共用户：

$ su -c 'adduser gituser'

然后切换到刚创建的 gituser 用户，创建一个 ~/.ssh 的框架，并设置好合适的权限。这很重要，如果权限设置得太开放会使自己所保护的 SSH 没有意义。

$ su - gituser

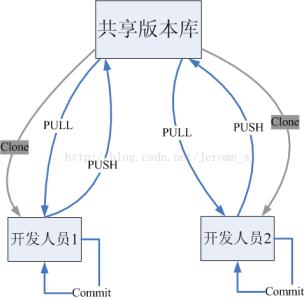
$ mkdir .ssh && chmod 700 .ssh

$ touch .ssh/authorized\_keys

$ chmod 600 .ssh/authorized\_keys

authorized\_keys 文件里包含所有你的开发者们的 SSH 公钥，你开放权限允许他们可以在你的 Git 项目上工作。他们必须创建他们自己的 SSH 密钥对然后把他们的公钥给你。复制公钥到 gituser 用户下的 authorized\_keys 文件中。例如，为一个叫 Bob 的开发者，执行以下命令：

$ cat ~/path/to/id\_rsa.bob.pub >> /home/gituser/.ssh/authorized\_keys

只要开发者 Bob 有私钥并且把相对应的公钥给你，Bob 就可以用 gituser 用户访问服务器。  


但是，你并不是想让你的开发者们能使用服务器，即使只是以 gituser 的身份访问。你只是想给他们访问 Git 仓库的权限。因为这个特殊的原因，Git 提供了一个限制的 shell，准确的说是git-shell 。以 root 身份执行以下命令，把 git-shell添加到你的系统中，然后设置成gituser 用户的默认 shell。

# grep git-shell /etc/shells || su -c "echo `which git-shell` >> /etc/shells"

# su -c 'usermod -s git-shell gituser'

现在 gituser 用户只能使用 SSH 来 push 或者 pull Git 仓库，并且无法使用任何一个可以登录的 shell。你应该把你自己添加到和 gituser 一样的组中，在我们的样例服务器中这个组的名字也是 gituser。

举个例子：

# usermod -a -G gituser seth

仅剩下的一步就是创建一个 Git 仓库。因为没有人能在服务器上直接与 Git 交互（也就是说，你之后不能 SSH 到服务器然后直接操作这个仓库），所以创建一个空的仓库 。如果你想使用这个放在服务器上的仓库来完成工作，你可以从它的所在处clone 下来，然后在你的 home 目录下进行工作。

严格地讲，你不是必须创建这个空的仓库；它和一个正常的仓库一样工作。但是，一个空的仓库没有工作分支（working tree） （也就是说，使用 checkout并没有任何分支显示）。这很重要，因为不允许远程使用者们push 到一个有效的分支上（如果你正在 dev 分支工作然后突然有人把一些变更push 到你的工作分支，你会有怎么样的感受？）。因为一个空的仓库可以没有有效的分支，所以这不会成为一个问题。

你可以把这个仓库放到任何你想放的地方，只要你想要放开权限给用户和用户组，让他们可以在仓库下工作。千万不要保存目录到比如说一个用户的 home 目录下，因为那里有严格的权限限制。保存到一个常规的共享地址，例如/opt 或者 /usr/local/share。

以 root 身份创建一个空的仓库：

# git init --bare /opt/jupiter.git

# chown -R gituser:gituser /opt/jupiter.git

# chmod -R 770 /opt/jupiter.git

现在任何一个用户，只要他被认证为gituser 或者在gituser 组中，就可以从 jupiter.git 库中读取或者写入。在本地机器尝试以下操作：

$ git clone gituser@example.com:/opt/jupiter.git jupiter.clone

Cloning into 'jupiter.clone'...

Warning: you appear to have cloned an empty repository.

谨记：开发者们一定要把他们的 SSH 公钥加入到gituser 用户下的 authorized\_keys文件里，或者说，如果他们有服务器上的用户（如果你给了他们用户），那么他们的用户必须属于 gituser 用户组。

Git 钩子

运行你自己的 Git 服务器最赞的一件事之一就是可以使用 Git 钩子。Git 托管服务有时提供一个钩子类的接口，但是他们并不会给你真正的 Git 钩子来让你访问文件系统。Git 钩子是一个脚本，它将在一个 Git 过程的某些点运行；钩子可以运行在当一个仓库即将接收一个commit 时、或者接受一个 commit之后，或者即将接收一次 push 时，或者一次 push之后等等。

这是一个简单的系统：任何放在.git/hooks 目录下的脚本、使用标准的命名体系，就可按设计好的时间运行。一个脚本是否应该被运行取决于它的名字；pre-push 脚本在push 之前运行，post-receive 脚本在接受 commit 之后运行等等。这或多或少的可以从名字上看出来。

脚本可以用任何语言写；如果在你的系统上有可以执行的脚本语言，例如输出 ‘hello world’ ，那么你就可以这个语言来写 Git 钩子脚本。Git 默认带了一些例子，但是并不有启用。

想要动手试一个？这很简单。如果你没有现成的 Git 仓库，首先创建一个 Git 仓库：

$ mkdir jupiter

$ cd jupiter

$ git init .

然后写一个 “hello world” 的 Git 钩子。因为我为了支持老旧系统而使用 tsch，所以我仍然用它作为我的脚本语言，你可以自由的使用自己喜欢的语言（Bash，Python，Ruby，Perl，Rust，Swift，Go）：

$ echo "#/!/bin/tcsh" > .git/hooks/post-commit

$ echo "echo 'POST-COMMIT SCRIPT TRIGGERED'" >> ~/jupiter/.git/hooks/post-commit

$ chmod +x ~/jupiter/.git/hooks/post-commit

现在测试它的输出：

$ echo "hello world" > foo.txt

$ git add foo.txt

$ git commit -m 'first commit'

! POST-COMMIT SCRIPT TRIGGERED

[master (root-commit) c8678e0] first commit

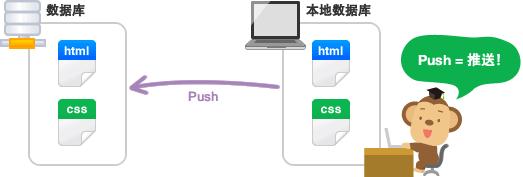
1 file changed, 1 insertion(+)

create mode 100644 foo.txt

现在你已经实现了：你的第一个有功能的 Git 钩子。

有名的 push-to-web 钩子

Git 钩子最流行的用法就是自动 push 更改的代码到一个正在使用中的产品级 Web 服务器目录下。这是摆脱 FTP 的很好的方式，对于正在使用的产品保留完整的版本控制，整合并自动化内容的发布。



如果操作正确，网站发布工作会像以前一样很好的完成，而且在某种程度上，很精准。Git 真的好棒。我不知道谁最初想到这个主意，但是我是从 Emacs 和 Git 方面的专家，IBM 的 Bill von Hagen 那里第一次听到它的。他的文章包含关于这个过程的权威介绍：Git 改变了分布式网页开发的游戏规则。

Git 变量

每一个 Git 钩子都有一系列不同的变量对应触发钩子的不同 Git 行为。你需不需要这些变量，主要取决于你写的程序。如果你只是需要一个当某人 push 代码时候的通用邮件通知，那么你就不需要什么特殊的东西，甚至也不需要编写额外的脚本，因为已经有现成的适合你的样例脚本。如果你想在邮件里查看commit 信息和 commit的作者，那么你的脚本就会变得相对麻烦些。

Git 钩子并不是被用户直接执行，所以要弄清楚如何收集可能会混淆的重要信息。事实上，Git 钩子脚本类似于其他的脚本，像 BASH、Python、C++ 等等一样从标准输入读取参数。不同的是，我们不会给它提供这个输入，所以，你在使用的时候，需要知道可能的输入参数。

在写 Git 钩子之前，看一下 Git 在你的项目目录下 .git/hooks 目录中提供的一些例子。举个例子，在这个 pre-push.sample文件里，注释部分说明了如下内容：

# $1 -- 即将 push 的远程仓库的名字

# $2 -- 即将 push 的远程仓库的 URL

# 如果 push 的时候，并没有一个命名的远程仓库，那么这两个参数将会一样。

#

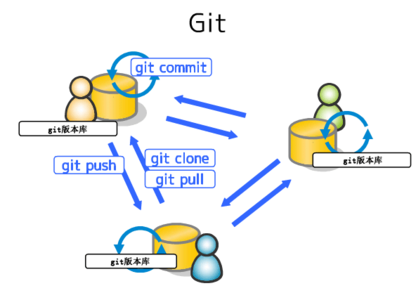
# 提交的信息将以下列形式按行发送给标准输入

# <local ref> <local sha1> <remote ref> <remote sha1>

并不是所有的例子都是这么清晰，而且关于钩子获取变量的文档依旧缺乏（除非你去读 Git 的源码）。但是，如果你有疑问，你可以从线上其他用户的尝试中学习，或者你只是写一些基本的脚本，比如 echo $1, $2, $3 等等。

分支检测示例

我发现，对于生产环境来说有一个共同的需求，就是需要一个只有在特定分支被修改之后，才会触发事件的钩子。以下就是如何跟踪分支的示例。

首先，Git 钩子本身是不受版本控制的。 Git 并不会跟踪它自己的钩子，因为对于钩子来说，它是 Git 的一部分，而不是你仓库的一部分。所以，Git 钩子可以监控你的 Git 服务器上的一个空仓库的 commit 记录和 push 记录，而不是你本地仓库的一部分。

我们来写一个 post-receive（也就是说，在 commit 被接受之后触发）钩子。第一步就是需要确定分支名：

#!/bin/tcsh

foreach arg ( $< )

set argv = ( $arg )

set refname = $1

end

这个 for 循环用来读入第一个参数 $1 ，然后循环用第二个参数$2 去覆盖它，然后用第三个参数 $3 再这样。在 Bash 中有一个更好的方法，使用 read命令，并且把值放入数组里。但是，这里是 tcsh，并且变量的顺序可以预测的，所以，这个方法也是可行的。

当我们有了 commit 记录的 refname，我们就能使用 Git 去找到这个分支的供人看的名字：

set branch = `git rev-parse --symbolic --abbrev-ref $refname`

echo $branch #DEBUG

然后把这个分支名和我们想要触发的事件的分支名关键字进行比较：

if ( "$branch" == "master" ) then

echo "Branch detected: master"

git /

--work-tree=/path/to/where/you/want/to/copy/stuff/to /

checkout -f $branch || echo "master fail"

else if ( "$branch" == "dev" ) then

echo "Branch detected: dev"

Git /

--work-tree=/path/to/where/you/want/to/copy/stuff/to /

checkout -f $branch || echo "dev fail"

else

echo "Your push was successful."

echo "Private branch detected. No action triggered."

endif

给这个脚本分配可执行权限：

$ chmod +x ~/jupiter/.git/hooks/post-receive

现在，当一个用户提交到服务器的 master 分支，那些代码就会被复制到一个生产环境的目录，提交到 dev 分支则会被复制到另外的地方，其他分支将不会触发这些操作。

同时，创造一个pre-commit 脚本也很简单。比如，判断一个用户是否在他们不该push 的分支上push 代码，或者对 commit 信息进行解析等等。

Git 钩子也可以变得复杂，而且它们因为 Git 的工作流的抽象层次不同而变得难以理解，但是它们确实是一个强大的系统，让你能够在你的 Git 基础设施上针对所有的行为进行对应的操作。如果你是一个 Git 重度用户，或者一个全职 Git 管理员，那么 Git 钩子是值得学习的，只有当你熟悉这个过程，你才能真正掌握它。

在我们这个系列下一篇也是最后一篇文章中，我们将会学习如何使用 Git 来管理非文本的二进制数据，比如音频和图片。