Gitolite 构建 Git 服务器

作者: 北京群英汇信息技术有限公司
网址: http://www.ossxp.com/
版本: 0.1-1
日期: 2010-10-07 14:52:19

目录

1 SSH 协议
- 1.1 SSH 公钥认证
- 1.2 SSH 主机别名

2 Gitolite 服务架设
- 2.1 安装 Gitolite
  - 2.1.1 服务器端创建专用帐号
  - 2.1.2 Gitolite 的安装/升级
  - 2.1.3 关于 SSH 主机别名
  - 2.1.4 其他的安装方法
- 2.2 管理 Gitolite
  - 2.2.1 管理员克隆 gitolite-admin 管理库
  - 2.2.2 增加新用户
  - 2.2.3 更改授权
- 2.3 Gitolite 授权详解
  - 2.3.1 授权文件的基本语法
  - 2.3.2 定义用户组和版本库组
  - 2.3.3 版本库ACL
  - 2.3.4 Gitolite 授权机制
- 2.4 版本库授权案例
  - 2.4.1 对整个版本库进行授权
  - 2.4.2 通配符版本库的授权
  - 2.4.3 用户自己的版本库空间
  - 2.4.4 对引用的授权：传统模式
  - 2.4.5 对引用的授权：扩展模式
  - 2.4.6 对引用的授权：禁用规则的使用
  - 2.4.7 用户分支
  - 2.4.8 对路径的写授权
- 2.5 创建新版本库
  - 2.5.1 在配置文件中出现的版本库，即时生成
  - 2.5.2 通配符版本库，管理员通过push创建
  - 2.5.3 直接在服务器端创建
- 2.6 对 Gitolite 的改进
- 2.7 Gitolite 功能拓展
  - 2.7.1 版本库镜像
  - 2.7.2 Gitweb 和 Gitdaemon 支持
  - 2.7.3 其他功能拓展和参考

如果不是要和他人协同开发，Git 根本就不需要架设服务器。Git 在本地可以直接使用本地版本库的路径完成 git 版本库间的操作。

但是如果需要和他人分享版本库、协作开发，就需要能够通过特定的网络协议操作 Git 库。

Git 支持的协议很丰富，架设服务器的选择也很多，不同的方案有着各自的优缺点。

	HTTP	GIT-DAEMON	SSH	GITOSIS, GITOLITE
服务架设难易度	简单	中等	简单	复杂
匿名读取	支持	支持	否*	否*
身份认证	支持	否	支持	支持
版本库写操作	支持	否	支持	支持
企业级授权支持	否	否	否	支持
是否支持远程建库	否	否	否	支持

注：

- SSH 协议和基于 SSH 的 Gitolite 等可以通过空口令帐号实现匿名访问。

1 SSH 协议

SSH 协议用于为 Git 提供远程读写操作，是远程写操作的标准服务，在智能HTTP协议出现之前，甚至是写操作的唯一标准服务。

对于拥有 SHELL 权限的 SSH 登录帐号，可以直接用下面的 git 命令访问，例如：

```
$ git clone <username>@<server>:/path/to/repo.git
```

说明：

- <username> 是服务器 <server> 上的用户帐号。
- /path/to/repo.git 是服务器中版本库的绝对路径。若用相对路径则相对于 username 用户的主目录而言。
- 如果采用口令认证，不能像 HTTPS 协议那样可以在 URL 中同时给出登录名和口令，必须每次连接时输入。
- 如果采用公钥认证，则无须输入口令。

SSH 协议来实现 Git 服务，有如下方式：

- 其一是用标准的 ssh 帐号访问版本库。即用户帐号可以直接登录到服务器，获得 shell。
- 另外的方式是，所有用户都使用同一个专用的 SSH 帐号访问版本库。各个用户通过公钥认证的方式用此专用 SSH 帐号访问版本库。而用户在连接时使用的不同的公钥可以用于区分不同的用户身份。

  Gitosis 和 Gitolite 就是实现该方式的两个服务器软件。

标准SSH帐号和专用SSH帐号的区别在于：

	标准SSH	GITOSIS/GITOLITE
帐号	每个用户一个帐号	所有用户共用同一个帐号
认证方式	口令或公钥认证	公钥认证
用户是否能直接登录 shell	是	否
安全性	差	好
管理员是否需要 shell	是	否
版本库路径	相对路径或绝对路径	相对路径
授权方式	操作系统中用户组和目录权限	通过配置文件授权
对分支进行写授权	否	Gitolite
对路径进行写授权	否	Gitolite

架设难易度	简单	复杂

实际上，标准SSH，也可以用公钥认证的方式实现所有用户共用同一个帐号。不过这类似于把一个公共帐号的登录口令同时告诉给多个人。

- 在服务器端(server)创建一个公共帐号，例如 *anonymous* 。
- 管理员收集需要访问 git 服务的用户公钥。如: *user1.pub*, *user2.pub* 。
- 使用 *ssh-copy-id* 命令远程将各个 git 用户的公钥加入服务器(server)的公钥认证列表中。

```
$ ssh-copy-id -i user1.pub anonymous@server
$ ssh-copy-id -i user2.pub anonymous@server
```

如果直接在服务器上操作，则直接将文件追加到 *authorized_keys* 文件中。

```
$ cat /path/to/user1.pub >> ~anonymous/.ssh/authorized_keys
$ cat /path/to/user2.pub >> ~anonymous/.ssh/authorized_keys
```

- 在服务器端的 *anonymous* 用户主目录下建立 git 库，就可以实现多个用户利用同一个系统帐号(git) 访问 Git 服务了。

这样做除了免除了逐一设置帐号，以及用户无需口令认证之外，标准SSH部署 Git 服务的缺点一个也不少，而且因为用户之间无法区分，更无法进行针对用户授权。

下面重点介绍一下 SSH 公钥认证，因为它们是后面介绍的 Gitosis 和 Gitolite 服务器软件的基础。

1.1 SSH 公钥认证

关于公钥认证的原理，维基百科上的这个条目是一个很好的起点： http://en.wikipedia.org/wiki/Public-key_cryptography 。

如果你的主目录下不存在 .ssh 目录，说明你的 SSH 公钥/私钥对尚未创建。可以用这个命令创建：

```
$ ssh-keygen
```

该命令会在用户主目录下创建 .ssh 目录，并在其中创建两个文件：

- id_rsa

  私钥文件。是基于 RSA 算法创建。该私钥文件要妥善保管，不要泄漏。

- id_rsa.pub

  公钥文件。和 *id_rsa* 文件是一对儿，该文件作为公钥文件，可以公开。

创建了自己的公钥/私钥对后，就可以使用下面的命令，实现无口令登录远程服务器，即用公钥认证取代口令认证。

```
$ ssh-copy-id -i .ssh/id_rsa.pub user@server
```

说明:

- 该命令会提示输入用户 user 在 server 上的SSH登录口令。
- 当此命令执行成功后，再以 user 用户登录 server 远程主机时，不必输入口令直接登录。
- 该命令实际上将 *.ssh/id_rsa.pub* 公钥文件追加到远程主机 server 的 user 主目录下的 *.ssh/authorized_keys* 文件中。

检查公钥认证是否生效，运行SSH到远程主机，正常的话应该直接登录成功。如果要求输入口令则表明公钥认证配置存在问题。如果SSH服务存在问题，可以通过查看服务器端的 */var/log/auth.log* 进行诊断。

1.2 SSH 主机别名

在实际应用中，有时需要使用多套公钥/私钥对，例如：

- 使用缺省的公钥访问 git 帐号，获取 shell，进行管理员维护工作。
- 使用单独创建的公钥访问 git 帐号，执行 git 命令。
- 访问 github（免费的Git服务托管商）采用其他公钥。

如何创建指定名称的公钥/私钥对呢？还是用 *ssh-keygen* 命令，如下：

```
$ ssh-keygen -f ~/.ssh/<filename>
```

注：

- 将 <filename> 替换为有意义的名称。
- 会在 ~/.ssh 目录下创建指定的公钥/私钥对。 文件 <filename> 是私钥，文件 <filename>.pub 是公钥。

将新生成的公钥添加到远程主机的 .ssh/authorized_keys 文件中，建立新的公钥认证。例如:

```
$ ssh-copy-id -i .ssh/<filename>.pub user@server
```

这样，就有两个公钥用于登录主机 server，那么当执行下面的 ssh 登录指令，用到的是那个公钥呢？

```
$ ssh user@server
```

当然是缺省公钥 ~/.ssh/id_rsa.pub 。那么如何用新建的公钥连接 server 呢？

SSH 的客户端配置文件 ~/.ssh/config 可以通过创建主机别名，在连接主机时，使用特定的公钥。例如 ~/.ssh/config 文件中的下列配置：

```
host bj
  user git
  hostname bj.ossxp.com
  port 22
  identityfile ~/.ssh/jiangxin
```

当执行

```
$ ssh bj
```

或者执行

```
$ git clone bj:path/to/repo.git
```

含义为：

- 登录的 SSH 主机为 *bj.ossxp.com* 。
- 登录时使用的用户名为 git 。
- 认证时使用的公钥文件为 *~/.ssh/jiangxin.pub* 。

2 Gitolite 服务架设

Gitolite 是一款 Perl 语言开发的 Git 服务管理工具，通过公钥对用户进行认证，并能够通过配置文件对写操作进行基于分支和路径的的精细授权。Gitolite 采用的是 SSH 协议并且使用 SSH 公钥认证，因此需要您对 SSH 非常熟悉，无论是管理员还是普通用户。因此在开始之前，请确认已经通读过之前的“SSH 协议”一章。

Gitolite 的官方网址是: http://github.com/sitaramc/gitolite 。从提交日志里可以看出作者是 Sitaram Chamarty，最早的提交开始于 2009年8月。作者是受到了 Gitosis 的启发，开发了这款功能更为强大和易于安装的软件。Gitolite 的命名，作者的原意是 Gitosis 和 lite 的组合，不过因为 Gitolite 的功能越来越强大，已经超越了 Gitosis，因此作者笑称 Gitolite 可以看作是 Github-lite —— 轻量级的 Github。

我是在2010年8月才发现 Gitolite 这个项目，并尝试将公司基于 Gitosis 的管理系统迁移至 Gitolite。在迁移和使用过程中，增加和改进了一些实现，如：通配符版本库的创建过程，对创建者的授权，版本库名称映射等。本文关于 Gitolite 的介绍也是基于我改进的 Gitosis 的版本。

- 原作者的版本库地址：

  http://github.com/sitaramc/gitolite

- 笔者改进后的 Gitolite 分支：

  http://github.com/ossxp-com/gitolite

Gitolite 的实现机制概括如下：

- Gitolite 安装在服务器(*server*) 某个帐号之下，例如 *git* 帐号。
- 管理员通过 git 命令检出名为 gitolite-admin 的版本库。

```
$ git clone git@server:gitolite-admin.git
```

- 管理员将 git 用户的公钥保存在 gitolite-admin 库的 keydir 目录下，并编辑 conf/gitolite.conf 文件为用户授权。
- 当管理员对 gitolite-admin 库的修改提交并 PUSH 到服务器之后，服务器上 gitolite-admin 版本库的钩子脚本将执行相应的设置工作。
  - 新用户公钥自动追加到服务器端安装帐号的 .ssh/authorized_keys 中，并设置该用户的 shell 为 gitolite 的一条命令 *gl-auth-command* 。

    ```
    command="/home/git/.gitolite/src/gl-auth-command jiangxin",no-port-forwar
    ```

  - 更新服务器端的授权文件 *~/.gitolite/conf/gitolite.conf* 。
  - 编译授权文件 *~/.gitolite/conf/gitolite.conf-compiled.pm* 。
- 用户可以用 git 命令访问授权的版本库。
- 当用户以 git 用户登录 ssh 服务时，因为公钥认证的相关设置，不再直接进入 shell 环境，而是打印服务器端 git 库授权信息后马上退出。

  即用户不会通过 git 用户进入服务器的 shell，也就不会对系统的安全造成威胁。
- 当管理员授权，用户可以远程在服务器上创建新版本库。

下面介绍 Gitolite 的部署和使用。在下面的示例中，约定：服务器的名称为 *server* ，Gitolite 的安装帐号为 *git* ，管理员的 ID 为 *admin* 。

2.1 安装 Gitolite

Gitolite 要求 git 的版本必须是 1.6.2 或以上的版本，并且服务器要提供 SSH 服务。下面是 Gitolite 的安装过程。

2.1.1 服务器端创建专用帐号

安装 Gitolite，首先要在服务器端创建专用帐号，所有用户都通过此帐号访问 Git 库。一般为方便易记，选择 git 作为专用帐号名称。

```
$ sudo adduser --system --shell /bin/bash --group git
```

创建用户 git，并设置用户的 shell 为可登录的 shell，如 /bin/bash，同时添加同名的用户组。

有的系统，只允许特定的用户组（如 ssh 用户组）的用户才可以通过 SSH 协议登录，这就需要将新建的 git 用户添加到 ssh 用户组中。

```
$ sudo adduser git ssh
```

为 git 用户设置口令。当整个 git 服务配置完成，运行正常后，建议取消 git 的口令，只允许公钥认证。

```
$ sudo passwd git
```

管理员在客户端使用下面的命令，建立无口令登录：

```
$ ssh-copy-id git@server
```

至此，我们已经完成了安装 git 服务的准备工作，可以开始安装 Gitolite 服务软件了。

2.1.2 Gitolite 的安装/升级

本节的名字称为安装/升级，是因为 Gitolite 的安装和升级可以采用下列同样的步骤。

Gitolite 安装可以在客户端执行，而不需要在服务器端操作，非常方便。安装 Gitolite 的前提是:

- 已经在服务器端创建了专有帐号，如 *git* 。
- 管理员能够以 git 用户身份通过公钥认证，无口令方式登录方式登录服务器。

安装和升级都可以按照下面的步骤进行：

- 使用 git 下载 Gitolite 的源代码。

  ```
  $ git clone git://github.com/ossxp-com/gitolite.git
  ```

- 进入 *gitolite/src* 目录，执行安装。

  ```
  $ cd gitolite/src
  $ ./gl-easy-install git server admin
  ```

  命令 *gl-easy-install* 的第一个参数 *git* 是服务器上创建的专用帐号ID，第二个参数 *server* 是服务器IP或者域名，第三个参数 *admin* 是管理员ID。

- 首先显示版本信息。

  ```
  ------------------------------------------------------------------------

  you are upgrading     (or installing first-time)     to v1.5.4-22-g4024621

  Note: getting '(unknown)' for the 'from' version should only happen once.
  Getting '(unknown)' for the 'to' version means you are probably installing
  from a tar file dump, not a real clone.  This is not an error but it's nice to
  have those version numbers in case you need support.  Try and install from a
  clone
  ```

- 自动创建名为 admin 的私钥/公钥对。

  gl-easy-install 命令行的最后一个参数即用于设定管理员ID，这里设置为 *admin* 。

  ```
  ------------------------------------------------------------------------

  the next command will create a new keypair for your gitolite access

  The pubkey will be /home/jiangxin/.ssh/admin.pub.  You will have to choose a
  passphrase or hit enter for none.  I recommend not having a passphrase for
  now, *especially* if you do not have a passphrase for the key which you are
  ```

```
already using to get server access!

Add one using 'ssh-keygen -p' after all the setup is done and you've
successfully cloned and pushed the gitolite-admin repo.  After that, install
'keychain' or something similar, and add the following command to your bashrc
(since this is a non-default key)

    ssh-add $HOME/.ssh/admin

This makes using passphrases very convenient.
```

如果公钥已经存在，会弹出警告。

```
------------------------------------------------------------------------

Hmmm... pubkey /home/jiangxin/.ssh/admin.pub exists; should I just (re-)use it?

IMPORTANT: once the install completes, *this* key can no longer be used to get
a command line on the server -- it will be used by gitolite, for git access
only.  If that is a problem, please ABORT now.

doc/6-ssh-troubleshooting.mkd will explain what is happening here, if you need
more info.
```

- 自动修改客户端的 .ssh/config 文件，增加别名主机。

  即当访问主机 gitolite 时，会自动用名为 admin.pub 的公钥，以 git 用户身份，连接服务器

```
------------------------------------------------------------------------

creating settings for your gitolite access in /home/jiangxin/.ssh/config;
these are the lines that will be appended to your ~/.ssh/config:

host gitolite
     user git
     hostname server
     port 22
     identityfile ~/.ssh/admin
```

- 上传脚本文件到服务器，完成服务器端软件的安装。

```
gl-dont-panic
gl-conf-convert
gl-setup-authkeys
...
gitolite-hooked
update


------------------------------------------------------------------------

the gitolite rc file needs to be edited by hand.  The defaults are sensible,
so if you wish, you can just exit the editor.

Otherwise, make any changes you wish and save it.  Read the comments to
understand what is what -- the rc file's documentation is inline.

Please remember this file will actually be copied to the server, and that all
```

```
the paths etc. represent paths on the server!
```

- 自动打开编辑器(vi)，编辑 .gitolite.rc 文件，编辑结束，上传到服务器。

  以下为缺省配置，一般无须改变：

  - $REPO_BASE="repositories";

    用于设置 Git 服务器的根目录，缺省是 Git 用户主目录下的 repositories 目录，可以使用绝对路径。所有 Git 库都将部署在该目录下。

  - $REPO_UMASK = 0007; # gets you 'rwxrwx---'

    版本库创建使用的掩码。即新建立版本库的权限为 'rwxrwx---'。

  - $GL_BIG_CONFIG = 0;

    如果授权文件非常复杂，更改此项配置为1，以免产生庞大的授权编译文件。

  - $GL_WILDREPOS = 1;

    缺省支持通配符版本库授权。

  该配置文件为 perl 语法，注意保持文件格式和语法。退出 vi 编辑器，输入 ":q" （不带引号）。

- 至此完成安装。

2.1.3 关于 SSH 主机别名

在安装过程中，gitolite 创建了名为 admin 的公钥/私钥对，以名为 *admin.pub* 的公钥连接服务器，由 gitolite 提供服务。但是如果直接连接服务器，使用的是缺省的公钥，会直接进入 shell。

那么如何能够根据需要选择不同的公钥来连接 git 服务器呢？

别忘了我们在前面介绍过的 SSH 主机别名。实际上刚刚在安装 gitolite 的时候，就已经自动为我们创建了一个主机别名。 打开 ~/.ssh/config 文件，可以看到类似内容，如果对主机别名不满意，可以修改。

```
host gitolite
     user git
     hostname server
     port 22
     identityfile ~/.ssh/admin
```

即：

- 像下面这样输入 SSH 命令，会直接进入 shell，因为使用的是缺省的公钥。

  ```
  $ ssh git@server
  ```

- 像下面这样输入 SSH 命令，则不会进入 shell。因为使用名为 admin.pub 的公钥，会显示 git 授权信息并马上退出。

  ```
  $ ssh gitolite
  ```

2.1.4 其他的安装方法

上面介绍的是在客户端远程安装 Gitolite，是最常用和推荐的方法。当然还可以直接在服务器上安装。

1. 首先也要在服务器端先创建一个专用的帐号，如：git 。

   ```
   $ sudo adduser --system --shell /bin/bash --group git
   ```

2. 将管理员公钥复制到服务器上。

管理员在客户端执行下面的命令：

```
$ scp ~/.ssh/id_rsa.pub server:/tmp/admin.pub
```

3. 服务器端安装 Gitolite。

推荐采用源码方式安装，因为如果以平台自带软件包模式安装 Gitolite，其中不包含我们对 Gitolite 的改进。

- 从源码安装。

  使用 git 下载 Gitolite 的源代码。

  ```
  $ git clone git://github.com/ossxp-com/gitolite.git
  ```

  创建目录。

  ```
  $ sudo mkdir -p /usr/local/share/gitolite/conf /usr/local/share/gitolite/
  ```

  进入 gitolite/src 目录，执行安装。

  ```
  $ cd gitolite/src
  $ sudo ./gl-system-install /usr/local/bin /usr/local/share/gitolite/conf
  ```

- 采用平台自带的软件包安装。

  例如在 Debian/Ubuntu 平台，执行下面命令：

  ```
  $ sudo aptitude install gitolite
  ```

  Redhat 则使用 yum 命令安装。

4. 在服务器端以专用帐号执行安装脚本。

例如服务器端的专用帐号为 git。

```
$ sudo su - git
$ gl-setup /tmp/admin.pub
```

5. 管理员在客户端，克隆 gitolite-admin 库

```
$ git clone git@server:gitolite-admin
```

升级 Gitolite:

- 只需要执行上面的第3个步骤即可完成升级。
- 如果修改或增加了新的了钩子脚本，还需要重新执行第4个步骤。
- Gitolite 升级有可能要求修改配置文件： *~/.gitolite.rc* 。

2.2 管理 Gitolite

2.2.1 管理员克隆 gitolite-admin 管理库

当 gitolite 安装完成后，在服务器端自动创建了一个用于 gitolite 自身管理的 git 库: *gitolite-admin.git* 。

克隆 *gitolite-admin.git* 库。别忘了使用SSH主机别名：

```
$ git clone gitolite:gitolite-admin.git

$ git clone gitolite:gitolite-admin.git
Initialized empty Git repository in /data/tmp/gitolite-admin/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.

$ cd gitolite-admin/

$ ls -F
conf/  keydir/

$ ls conf
gitolite.conf

$ ls keydir/
admin.pub
```

我们可以看出 *gitolite-admin* 目录下有两个目录 *conf/* 和 *keydir/* 。

- *keydir/admin.pub* 文件

  目录 *keydir* 下初始时只有一个用户公钥，即 amdin 用户的公钥。

- *conf/gitolite.conf* 文件

  该文件为授权文件。初始内容为：

  ```
  #gitolite conf
  # please see conf/example.conf for details on syntax and features

  repo gitolite-admin
      RW+                 = admin

  repo testing
      RW+                 = @all
  ```

  缺省授权文件中只设置了两个版本库的授权：

  - gitolite-admin

    即本版本库（gitolite管理版本库）只有 admin 用户有读写和强制更新的权限。

  - testing

    缺省设置的测试版本库，设置为任何人都可以读写以及强制更新。

2.2.2 增加新用户

增加新用户，就是允许新用户能够通过其公钥访问 Git 服务。只要将新用户的公钥添加到 gitolite-admin 版本库的 keydir 目录下，即完成新用户的添加。

- 管理员从用户获取公钥，并将公钥按照 username.pub 格式进行重命名。

  用户可以通过邮件或者其他方式将公钥传递给管理员，切记不要将私钥误传给管理员。如果发生私钥泄漏，马上重新生成新的公钥/私钥对，并将新的公钥传递给管理员，并申请将旧的公钥作废。

  用户从不同的客户端主机访问有着不同的公钥，如果希望使用同一个用户名进行授权，可以按照 *username@host.pub* 方式命名公钥文件，和名为 *username@pub* 的公钥指向同一个用户 *username* 。

  Gitolite 也支持邮件地址格式的公钥，即形如 *username@gmail.com.pub* 的公钥。Gitolite 能够很智能的区分是以邮件地址命名的公钥还是相同用户在不同主机上的公钥。如果是邮件地址命名的公钥，将以整个邮件地址作为用户名。

- 管理员进入 gitolite-admin 本地克隆版本库中，复制新用户公钥到 keydir 目录。

```
$ cp /path/to/dev1.pub keydir/
$ cp /path/to/dev2.pub keydir/
$ cp /path/to/jiangxin.pub keydir/
```

- 执行 git add 命令，将公钥添加入版本库。

```
$ git add keydir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   keydir/dev1.pub
#       new file:   keydir/dev2.pub
#       new file:   keydir/jiangxin.pub
#
```

- 执行 git commit，完成提交。

```
$ git commit -m "add user: jiangxin, dev1, dev2"
[master bd81884] add user: jiangxin, dev1, dev2
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 keydir/dev1.pub
 create mode 100644 keydir/dev2.pub
 create mode 100644 keydir/jiangxin.pub
```

- 执行 git push，同步到服务器，才真正完成新用户的添加。

```
$ git push
Counting objects: 8, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.38 KiB, done.
Total 6 (delta 0), reused 0 (delta 0)
remote: Already on 'master'
remote:
remote:                 ***** WARNING *****
remote:         the following users (pubkey files in parens) do not appear in the
remote: dev1(dev1.pub),dev2(dev2.pub),jiangxin(jiangxin.pub)
```

如果我们这时查看服务器端 *~git/.ssh/authorized_keys* 文件，会发现新增的用户公钥也附加其中：

```
# gitolite start
command="/home/git/.gitolite/src/gl-auth-command admin",no-port-forwarding,no-X11-forwardi
command="/home/git/.gitolite/src/gl-auth-command dev1",no-port-forwarding,no-X11-forwardin
command="/home/git/.gitolite/src/gl-auth-command dev2",no-port-forwarding,no-X11-forwardin
command="/home/git/.gitolite/src/gl-auth-command jiangxin",no-port-forwarding,no-X11-forwa
# gitolite end
```

在之前执行 git push 后的输出中，以 remote 标识的输出是服务器端执行 *post-update* 钩子脚本的输出。其中的警告是说新添加的三个用户在授权文件中没有被引用。接下来我们便看看如何修改授权文件，以及如何为用户添加授权。

2.2.3 更改授权

新用户添加完毕，可能需要重新进行授权。更改授权的方法也非常简单，即修改 conf/gitolite.conf 配置文件，提交并

push。

- 管理员进入 *gitolite-admin* 本地克隆版本库中，编辑 *conf/gitolite.conf* 。

```
$ vi conf/gitolite.conf
```

- 授权指令比较复杂，我们先通过建立新用户组尝试一下更改授权文件。

  考虑到之前我们增加了三个用户公钥之后，服务器端发出了用户尚未在授权文件中出现的警告。我们就在这个示例中解决这个问题。

  - 例如我们在其中加入用户组 @team1，将新添加的用户 jiangxin, dev1, dev2 都归属到这个组中。

    我们只需要在 conf/gitolite.conf 文件的文件头加入如下指令。用户之间用空格分隔。

    ```
    @team1 = dev1 dev2 jiangxin
    ```

  - 编辑完毕退出。我们可以用 *git diff* 命令查看改动：

    我们还修改了版本库 *testing* 的授权，将 *@all* 用户组改为我们新建立的 *@team1* 用户组。

    ```
    $ git diff
    diff --git a/conf/gitolite.conf b/conf/gitolite.conf
    index 6c5fdf8..f983a84 100644
    --- a/conf/gitolite.conf
    +++ b/conf/gitolite.conf
    @@ -1,10 +1,12 @@
     #gitolite conf
     # please see conf/example.conf for details on syntax and features

    +@team1 = dev1 dev2 jiangxin
    +
     repo gitolite-admin
         RW+                 = admin

     repo testing
    -    RW+                 = @all
    +    RW+                 = @team1
    ```

- 编辑结束，提交改动。

```
$ git add conf/gitolite.conf
$ git commit -q -m "new team @team1 auth for repo testing."
```

- 执行 *git push* ，同步到服务器，才真正完成授权文件的编辑。

  我们可以注意到，PUSH 后的输出中没有了警告。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 398 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
To gitadmin.bj:gitolite-admin.git
   bd81884..79b29e4  master -> master
```

2.3 Gitolite 授权详解

2.3.1 授权文件的基本语法

下面我们看一个不那么简单的授权文件:

```
1   @admin = jiangxin wangsheng
2
3   repo gitolite-admin
4       RW+                 = jiangxin
5
6   repo ossxp/.+
7       C                   = @admin
8       RW                  = @all
9
10  repo testing
11      RW+                         =   @admin
12      RW      master              =   junio
13      RW+     pu                  =   junio
14      RW      cogito$             =   pasky
15      RW      bw/                 =   linus
16      -                           =   somebody
17      RW      tmp/                =   @all
18      RW      refs/tags/v[0-9]    =   junio
```

在上面的示例中，我们演示了很多授权指令。

- 第1行，定义了用户组 @admin，包含两个用户 jiangxin 和 wangsheng。
- 第3-4行，定义了版本库 gitolite-admin。并指定只有用户 jiangxin 才能够访问，并拥有读(R)写(W)和强制更新(+)的权限。
- 第6行，通过正则表达式定义了一组版本库，即在 ossxp/ 目录下的所有版本库。
- 第7行，用户组 *@admin* 中的用户，可以在 *ossxp/* 目录下创建版本库。

  创建版本库的用户，具有对版本库操作的所有权限。
- 第8行，所有用户都可以读写 *ossxp* 目录下的版本库，但不能强制更新。
- 第9行开始，定义的 *testing* 版本库授权使用了引用授权语法。
- 第11行，用户组 *@admin* 对所有的分支和里程碑拥有读写、重置、添加和删除的授权。
- 第12行，用户 *junio* 可以读写 *master* 分支。（还包括名字以 master 开头的其他分支，如果有的话）。
- 第13行，用户 *junio* 可以读写、强制更新、创建以及删除 *pu* 开头的分支。
- 第14行，用户 *pasky* 可以读写 *cogito* 分支。 (仅此分支，精确匹配）。

2.3.2 定义用户组和版本库组

在 *conf/gitolite.conf* 授权文件中，可以定义用户组或者版本库组。组名称以 @ 字符开头，可以包含一个或多个成员。成员之间用空格分开。

- 例如定义管理员组:

  ```
  @admin = jiangxin wangsheng
  ```

- 组可以嵌套:

  ```
  @staff = @admin @engineers tester1
  ```

- 除了作为用户组外，同样语法也适用于版本库组。

  版本库组和用户组的定义没有任何区别，只是在版本库授权指令中处于不同的位置。即位于授权指令中的版本库位置则代表版本库组，位于授权指令中的用户位置则代表用户组。

2.3.3 版本库ACL

一个版本库可以包含多条授权指令，这些授权指令组成了一个版本库的权限控制列表（ACL）。

例如：

```
repo testing
    RW+                 = jiangxin @admin
    RW                  = @dev @test
    R                   = @all
```

每一个版本库授权都以一条 *repo* 指令开始。

- 指令 *repo* 后面是版本库列表，版本之间用空格分开，还可以包括版本库组。

  注意：版本库名称不要添加 *.git* 后缀。在版本库创建过程中会自动添加 *.git* 后缀。

  ```
  repo sandbox/test1 sandbox/test2 @test_repos
  ```

- repo 指令后面的版本库也可以用正则表达式定义的 *通配符版本库* 。

  正则表达式匹配时，会自动在 *通配符版本库* 的前后加上前缀 ^ 和后缀 *$* 。这一点和后面将介绍的正则引用（refex）大不一样。

  ```
  repo ossxp/.+
  ```

  不过有时候使用了过于简单的正则表达式如： "*myrepo.*"，有可能产生歧义，让 Gitolite 误认为是普通版本库名称，在服务器端自动创建名为 *myrepo..git* 的版本库。解决歧义的一个办法是：在正则表达式的前面插入 ^ 符号，或者在表达式后面添加 *$* 符号，形如："*^myrepo.$*"。

在 repo 指令之后，是缩进的一条或者多条授权指令。授权指令的语法：

```
<权限>  [零个或多个正则表达式匹配的引用] = <user> [<user> ...]
```

- 每条指令必须指定一个权限。权限可以用下面的任意一个权限关键字：

  C, R, RW, RW+, RWC, RW+C, RWD, RW+D, RWCD, RW+CD 。

- 权限后面包含一个可选的 refex（正则引用）列表。

  正则表达式格式的引用，简称正则引用（refex），对 Git 版本库的引用（分支，里程碑等）进行匹配。

  如果在授权指令中省略正则引用，意味着对全部的 Git 引用（分支，里程碑等）都有效。

  正则引用如果不以 *refs/* 开头，会自动添加 *refs/heads/* 作为前缀。

  正则引用如果不以 *$* 结尾，意味着后面可以匹配任意字符，相当于添加 *.*$* 作为后缀。

- 权限后面也可以包含一个以 *NAME/* 开头的路径列表，进行基于路径的授权。
- 授权指令以等号（=）为标记分为前后两段，等号后面的是用户列表。

  用户之间用空格分隔，并且可以使用用户组。

不同的授权关键字有不同的含义，有的授权关键字只用在 **特定** 的场合。

- C

  C 代表创建。仅在 *通配符版本库* 授权时可以使用。用于指定谁可以创建和通配符匹配的版本库。

- R, RW, 和 RW+

  R 为只读。RW 为读写权限。RW+ 含义为除了具有读写外，还可以对 rewind 的提交强制 PUSH。

- RWC, RW+C

只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等），才可以使用该授权指令。其中 C 的含义是允许创建和正则引用匹配的引用（分支或里程碑等）。

- RWD, RW+D

  只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等），才可以使用该授权指令。其中 D 的含义是允许删除和正则引用匹配的引用（分支或里程碑等）。

- RWCD, RW+CD

  只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等），才可以使用该授权指令。其中 C 的含义是允许创建和正则引用匹配的引用（分支或里程碑等），D 的含义是允许删除和正则引用匹配的引用（分支或里程碑等）。

2.3.4 Gitolite 授权机制

Gitolite 的授权实际分为两个阶段，第一个阶段称为前Git阶段，即在 Git 命令执行前，由 SSH 链接触发的 *gl-auth-command* 命令执行的授权检查。包括：

- 版本库的读。

  用户必须拥有版本库至少一个分支的下列权限之一： *R*, *RW*, 或 *RW+* ，则整个版本库包含所有分支对用户均可读。

  而版本库分支实际上在这个阶段获取不到，即版本库的读取不能按照分支授权，只能是版本库的整体授权。

- 版本库的写。

  版本库的写授权，则要在两个阶段分别进行检查。第一阶段的检查是看用户是否拥有下列权限之一： *RW*, *RW+* 或者 *C* 授权。

  第二个阶段检查分支以及是否拥有强制更新。具体见后面的描述。

- 版本库的创建。

  仅对正则表达式定义的通配符版本库有效。即拥有 *C* 授权的用户，可以创建和对应正则表达式匹配的版本库。同时该用户也拥有对版本库的读写权限。

对授权的第二个阶段的检查，实际上是通过 *update* 钩子脚本进行的。

- 因为版本库的读操作不执行 *update* 钩子，所以读操作只在授权的第一个阶段（前Git阶段）进行检查，授权的第二个阶段对版本库的读授权无任何影响。
- 钩子脚本 *update* 针对 PUSH 操作的各个分支进行逐一检查，因此第二个阶段可以进行针对分支写操作的精细授权。
- 在这个阶段也可以获取到要更新的新的和老的 ref 的 SHA 摘要，因此也可以进行是否有回滚（rewind）的发生，即是否允许强制更新，还可以对分支的创建和删除进行授权检测。
- 基于路径的写授权，也是在这个阶段进行的。

2.4 版本库授权案例

Gitolite 的授权非常强大也非常复杂，因此从版本库授权的实际案例来学习非常行之有效。

2.4.1 对整个版本库进行授权

授权文件如下：

```
1  @admin = jiangxin
2  @dev   = dev1 dev2 badboy jiangxin
3  @test  = test1 test2
4
5  repo testing
6      R = @test
7      - = badboy
8      RW = @dev test1
9      RW+ = @admin
```

说明：

- 用户 *test1* 对版本库具有写的权限。

  第6行定义了 *test1* 所属的用户组 *@test* 具有只读权限。第8行定义了 test1 用户具有读写权限。

Gitolite 的实现是读权限和写权限分别进行判断并汇总（并集），从而 *test1* 用户具有读写权限。

- 用户 *jiangxin* 对版本库具有写的权限，并能强制PUSH。

  第9行授权指令中的加号（+）含义是允许强制 PUSH。

- 禁用指令，让用户 *badboy* 对版本库只具有读操作的权限。

  第7行的指令以减号（-）开始，是一条禁用指令。禁用指令只在授权的第二阶段起作用，即只对写操作起作用，不会对 *badboy* 用户的读权限施加影响。

  在第8行的指令中， *badboy* 所在的 *@dev* 组拥有读取权限。但禁用规则会对写操作起作用，导致 *badboy* 只有读操作权限，而没有写操作。

2.4.2 通配符版本库的授权

授权文件如下：

```
1  @administrators = jiangxin admin
2  @dev    = dev1 dev2 badboy
3  @test   = test1 test2
4
5  repo sandbox/.+$
6      C = @administrators
7      R = @test
8      - = badboy
9      RW = @dev test1
```

这个授权文件中的版本库名称中使用了正则表达式，匹配在 sandbox 下的任意版本库。

Tip

正则表达式末尾的 *$* 有着特殊的含义，代表匹配字符串的结尾，明确告诉 Gitolite 这个版本库是通配符版本库。

因为加号 + 既可以作为普通字符出现在版本库的命名中，又可以作为正则表达式中特殊含义的字符，如果 Gitolite 将授权文件中的通配符版本库误判为普通版本库，就会自动在服务器端创建该版本库，这是可能管理员不希望发生的。

在版本库结尾添加一个 *$* 字符，就明确表示该版本库为正则表达式定义的通配符版本库。

我修改了 Gitolite 的代码，能正确判断部分正则表达式，但是最好还是对简单的正则表达式添加 ^ 作为前缀，或者添加 *$* 作为后缀，避免误判。

正则表达式定义的通配符版本库不会自动创建。需要管理员手动创建。

Gitolite 原来对通配符版本库的实现是克隆即创建，但是这样很容易因为录入错误导致错误的版本库意外被创建。群英汇改进的 Gitolite 需要通过 PUSH 创建版本库。

以 *admin* 用户的身份创建版本库 *sandbox/repos1.git* 。

```
$ git push git-admin-server:sandbox/repos1.git master
```

创建完毕后，我们对各个用户的权限进行测试，会发现：

- 用户 *admin* 对版本库具有写的权限。

  这并不是因为第6行的授权指令为 *@administrators* 授予了 C 的权限。而是因为该版本库是由 *admin* 用户创建的，创建者具有对版本库完全的读写权限。

  服务器端该版本库目录自动生成的 *gl-creator* 文件记录了创建者 ID 为 *admin* 。

- 用户 *jiangxin* 对版本库没有读写权限。

  虽然用户 *jiangxin* 和用户 *admin* 一样都可以在 *sandbox/* 下创建版本库，但是由于 *sandbox/repos1.git* 已经存在并且不是 *jiangxin* 用户创建的，所以 *jiangxin* 用户没有任何权限，不能读写。

- 和之前的例子相同的是：
  - 用户 *test1* 对版本库具有写的权限。
  - 禁用指令，让用户 *badboy* 对版本库只具有读操作的权限。

- 版本库的创建者还可以使用 setperms 命令为版本库添加授权。具体用法参见下面的示例。

2.4.3 用户自己的版本库空间

授权文件如下：

```
1  @administrators = jiangxin admin
2
3  repo users/CREATOR/.+$
4      C = @all
5      R = @administrators
```

说明：

- 用户可以在自己的名字空间（ */usrs/<userid>/* ）下，自己创建版本库。

  ```
  $ git push dev1@server:users/dev1/repos1.git master
  ```

- 设置管理员组对任何用户在 *users/* 目录下创建的版本库都有只读权限。
- 用户可以使用 setperms 为自己的版本库进行二次授权

  ```
  $ ssh dev1@server setperms users/dev1/repos1.git
  R = dev2
  RW = jiangxin
  ^D
  ```

  即在输入 setperms 命令后，进入一个编辑界面，输入 ^D（Ctrl+D）结束编辑。

  也可以使用输入重定向，先将授权写入文件，再用 setperms 命令加载。

  ```
  $ cat > perms << EOF
  R = dev2
  RW = jiangxin
  EOF

  $ ssh dev1@server setperms < perms
  ```

- 用户可以使用 getperms 查看对自己版本库的授权

  ```
  $ ssh dev1@server getperms users/dev1/repos1.git
  R = dev2
  RW = jiangxin
  ```

2.4.4 对引用的授权：传统模式

传统的引用授权，指的是授权指令中不包含 *RWC*, *RWD*, *RWCD*, *RW+C*, *RW+D*, *RW+CD* 授权关键字，只采用 *RW*, *RW+* 的传统授权关键字。

在只使用传统的授权关键字的情况下，有如下注意事项：

- rewind 必须拥有 + 的授权。
- 创建引用必须拥有 *W* 的授权。
- 删除引用必须拥有 + 的授权。
- 如果没有在授权指令中提供引用相关的参数，相当于提供 *refs/.** 作为引用的参数，意味着对所有引用均有效。

授权文件：

```
1  @administrators = jiangxin admin
2  @dev    = dev1 dev2 badboy
```

```
3
4  repo test/repo1
5      RW+ = @administrators
6      RW master refs/heads/feature/ = @dev
7      R   = @test
```

说明：

- 第5行，版本库 *test/repo1* ，管理员组用户 *jiangxin* 和 *admin* 可以任意创建和删除引用，并且可以强制 PUSH。
- 第6行的规则看似只对 master 和 *refs/heads/feature/** 的引用授权，实际上 *@dev* 可以读取所有名字空间的引用。这是因为读取操作无法获得 ref 相关内容。

  即用户组 *@dev* 的用户只能对 master 分支，以及以 *feature/* 开头的分支进行写操作，但不能强制 PUSH 和删除。至于其他分支和里程碑，则只能读不能写。
- 至于用户组 *@test* 的用户，因为使用了 R 授权指令，所以不涉及到分支的写授权。

2.4.5 对引用的授权：扩展模式

扩展模式的引用授权，指的是该版本库的授权指令出现了下列授权关键字中的一个或多个： *RWC*, *RWD*, *RWCD*, *RW+C*, *RW+D*, *RW+CD* 。

- rewind 必须拥有 + 的授权。
- 创建引用必须拥有 *C* 的授权。
- 删除引用必须拥有 *D* 的授权。

授权文件：

```
repo test/repo2
    RW+C = @administrators
    RW+  = @dev
    RW   = @test

repo test/repo3
    RW+CD = @administrators
    RW+C  = @dev
    RW    = @test
```

说明：

对于版本库 *test/repo2.git* ：

- 用户组 *@administrators* 中的用户，具有创建和删除引用的权限，并且能强制 PUSH。
- 用户组 *@dev* 中的用户，不能创建引用，但可以删除引用，以及可以强制 PUSH。
- 用户组 *@test* 中的用户，可以 PUSH 到任何引用，但是不能创建引用，不能删除引用，也不能强制 PUSH。

对于版本库 *test/repo3.git* ：

- 用户组 *@administrators* 中的用户，具有创建和删除引用的权限，并且能强制 PUSH。
- 用户组 *@dev* 中的用户，可以创建引用，并能够强制 PUSH，但不能删除引用，
- 用户组 *@test* 中的用户，可以 PUSH 到任何引用，但是不能创建引用，不能删除引用，也不能强制 PUSH。

2.4.6 对引用的授权：禁用规则的使用

授权文件：

```
1  repo testing

       ...

12     RW      refs/tags/v[0-9]        =   jiangxin
13     -       refs/tags/v[0-9]        =   dev1 dev2 @others
14     RW      refs/tags/              =   jiangxin dev1 dev2 @others
```

说明：

- 用户 jiangxin 可以写任何里程碑，包括以 v 加上数字开头的里程碑。

- 用户 dev1, dev2 和 @others 组，只能写除了以 v 加上数字开头之外的其他里程碑。
- 其中以 - 开头的授权指令建立禁用规则。禁用规则只在授权的第二阶段有效，因此不能对用户的读取进行限制！

2.4.7 用户分支

和创建用户空间（使用了 *CREATOR* 关键字）的版本库类似，还可以在一个版本库内，允许管理自己名字空间（ *USER* 关键字）下的分支。在正则引用的参数中出现的 *USER* 关键字会被替换为用户的 ID。

授权文件：

```
repo test/repo4
    RW+CD = @administrators
    RW+CD refs/personal/USER/  = @all
    RW+    master = @dev
```

说明：

- 用户组 *@administrators* 中的用户，对所有引用具有创建和删除引用的权限，并且能强制 PUSH。
- 所有用户都可以在 *refs/personal/<userid>/* （自己的名字空间）下创建、删除引用。但是不能修改其他人的引用。
- 用户组 *@dev* 中的用户，对 master 分支具有读写和强制更新的权限，但是不能删除。

2.4.8 对路径的写授权

Gitolite 也实现了对路径的写操作的精细授权，并且非常巧妙的是：在实现上增加的代码可以忽略不计。这是因为 Gitolite 把对路径当作是特殊格式的引用的授权。

在授权文件中，如果一个版本库的授权指令中的正则引用字段出现了以 *NAME/* 开头的引用，则表明该授权指令是针对路径进行的写授权，并且该版本库要进行基于路径的写授权判断。

示例：

```
1  repo foo
2      RW                  =   @junior_devs @senior_devs
3
4      RW  NAME/           =   @senior_devs
5      -   NAME/Makefile   =   @junior_devs
6      RW  NAME/           =   @junior_devs
```

说明：

- 第2行，初级程序员 *@junior_devs* 和高级程序员 *@senior_devs* 可以对版本库 *foo* 进行读写操作。
- 第4行，设定高级程序员 *@senior_devs* 对所有文件（ *NAME/* ）进行写操作。
- 第5行和第6行，设定初级程序员 *@junior_devs* 对除了根目录的 *Makefile* 文件外的其他文件 ，具有写权限。

2.5 创建新版本库

Gitolite 维护的版本库位于安装用户主目录下的 repositories 目录中，即如果安装用户为 *git* ，则版本库都创建在 /home/git/repositories 目录之下。可以通过配置文件 .gitolite.rc 修改缺省的版本库的根路径。

```
$REPO_BASE="repositories";
```

有多种创建版本库的方式。一种是在授权文件中用 repo 指令设置版本库（未使用正则表达式的版本库）的授权，当对 gitolite-admin 版本库执行 git push 操作，自动在服务端创建新的版本库。另外一种方式是在授权文件中用正则表达式定义的版本库，不会即时创建，而是被授权的用户在远程创建后PUSH到服务器上完成创建。

注意，在授权文件中创建的版本库名称不要带 .git 后缀，在创建版本库过程中会自动在版本库后面追加 .git 后缀。

2.5.1 在配置文件中出现的版本库，即时生成

我们尝试在授权文件 *conf/gitolite.conf* 中加入一段新的版本库授权指令，而这个版本库尚不存在。新添加到授权文件中的内容：

```
repo testing2
    RW+                 = @all
```

然后将授权文件的修改提交并 PUSH 到服务器，我们会看到授权文件中添加新授权的版本库 testing2 被自动创建。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 375 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
remote: creating testing2...
remote: Initialized empty Git repository in /home/git/repositories/testing2.git/
To gitadmin.bj:gitolite-admin.git
   278e54b..b6f05c1  master -> master
```

注意其中带 remote 标识的输出，我们看到版本库 testing2.git 被自动初始化了。

此外使用版本库组的语法（即用 @ 创建的组，用作版本库），也会被自动创建。例如下面的授权文件片段设定了一个包含两个版本库的组 *@testing* ，当将新配置文件 PUSH 到服务器上的时候，会自动创建 *testing3.git* 和 *testing4.git* 。

```
@testing = testing3 testing4

repo @testing
    RW+                 = @all
```

还有一种版本库语法，是用正则表达式定义的版本库，这类版本库因为所指的版本库并不确定，因此不会自动创建。

2.5.2 通配符版本库，管理员通过push创建

通配符版本库是用正则表达式语法定义的版本库，所指的非某一个版本库而是和名称相符的一组版本库。首先要想使用通配符版本库，需要在服务器端安装用户（如 *git* ）用户的主目录下的配置文件 *.gitolite.rc* 中，包含如下配置：

```
$GL_WILDREPOS = 1;
```

使用通配符版本库，可以对一组版本库进行授权，非常有效。但是版本库的创建则不像前面介绍的那样，不会在授权文件 PUSH 到服务器时创建，而是拥有版本库创建授权（C）的用户手工进行创建。

对于用通配符设置的版本库，用 C 指令指定能够创建此版本库的管理员（拥有创建版本库的授权）。例如：

```
repo ossxp/.+
    C                   = jiangxin
    RW                  = dev1 dev2
```

管理员 jinagxin 可以创建路径符合正则表达式 "*ossxp/.+*" 的版本库，用户 dev1 和 dev2 对版本库具有读写（但是没有强制更新）权限。

使用该方法创建版本库后，创建者的 uid 将被记录在版本库目录下的 gl-creator 文件中。该帐号具有对该版本库最高的权限。该通配符版本库的授权指令中如果出现 *CREATOR* 将被创建者的 uid 替换。

- 本地建库

  ```
  $ mkdir somerepo
  $ cd somerepo
  $ git init
  $ git commit --allow-empty
  ```

- 使用 git remote 指令添加远程的源

  ```
  $ git remote add origin jiangxin@server:ossxp/somerepo.git
  ```

- 运行 git push 完成在服务器端版本库的创建

```
$ git push origin master
```

克隆即创建，还是PUSH即创建？

Gitolite 的原始实现是通配符版本库的管理员在对不存在的版本库执行 clone 操作时，自动创建。但是我认为这不是一个好的实践，会经常因为 clone 的 URL 写错，导致在服务器端创建垃圾版本库。因此我重新改造了 gitolite 通配符版本库创建的实现，改为在对版本库进行 PUSH 的时候进行创建，而 clone 一个不存在的版本库，会报错退出。

2.5.3 直接在服务器端创建

当版本库的数量很多的时候，在服务器端直接通过 git 命令创建或者通过复制创建可能会更方便。但是要注意，在服务器端手工创建的版本库和 Gitolite 创建的版本库最大的不同在于钩子脚本。如果不能为手工创建的版本库正确设定版本库的钩子，会导致失去一些 Gitolite 特有的功能。例如：失去分支授权的功能。

一个由 Gitolite 创建的版本库，hooks 目录下有三个钩子脚本实际上链接到 gitolite 安装目录下的相应的脚本文件中：

```
gitolite-hooked -> /home/git/.gitolite/hooks/common/gitolite-hooked
post-receive.mirrorpush -> /home/git/.gitolite/hooks/common/post-receive.mirrorpush
update -> /home/git/.gitolite/hooks/common/update
```

那么手工在服务器上创建的版本库，有没有自动更新钩子脚本的方法呢？

有，就是重新执行一遍 gitolite 的安装，会自动更新版本库的钩子脚本。安装过程一路按回车即可。

```
$ cd gitolite/src
$ ./gl-easy-install git server admin
```

除了钩子脚本要注意以外，还要确保服务器端版本库目录的权限和属主。

2.6 对 Gitolite 的改进

笔者对 Gitolite 进行扩展和改进，涉及到的内容主要包括：

- 通配符版本库的创建方式和授权。

  原来的实现是克隆即创建（克隆者需要被授予 C 的权限）。同时还要通过另外的授权语句为用户设置 RW 权限，否则创建者没有读和写权限。

  新的实现是通过 PUSH 创建版本库（PUSH 者需要被授予 C 权限）。不必再为创建者赋予 RW 等权限，创建者自动具有对版本库最高的授权。

- 避免通配符版本库误判。

  通配符版本库误判，会导致在服务器端创建错误的版本库。新的设计还可以在通配符版本库的正则表达式前或后添加 ^ 或 $ 字符，而不会造成授权文件编辑错误。

- 改变缺省配置。

  缺省安装即支持通配符版本库。

- 版本库重定向。

  Gitosis 的一个很重要的功能：版本库名称重定向，没有在 Gitolite 中实现。我们为 Gitolite 增加了这个功能。

  在Git服务器架设的开始，版本库的命名可能非常随意，例如 redmine 的版本库直接放在根下，例如： *redmine-0.9.x.git*, *redmine-1.0.x.git*, ... 当 *redmine* 项目越来越复杂，可能就需要将其放在子目录下进行管理，例如放到 *ossxp/redmine/* 目录下。

  只需要在 Gitolite 的授权文件中添加下面一行 map 语句，就可以实现版本库名称重定向。使用旧的地址的用户不必重新检出，可以继续使用。

```
map (redmine.*) = ossxp/redmine/$1
```

2.7 Gitolite 功能拓展

2.7.1 版本库镜像

Git 版本库控制系统往往并不需要设计特别的容灾备份，因为每一个Git用户就是一个备份。但是下面的情况，就很有必要考虑容灾了。

- Git 版本库的使用者很少（每个库可能只有一个用户）。
- 版本库检出只限制在办公区并且服务器也在办公区内（所有鸡蛋在一个篮子里）。
- Git 版本库采用集中式的应用模型，需要建立双机热备（以便在故障出现时，实现快速的服务器切换）。

Gitolite 提供了服务器间版本库同步的设置。原理是：

- 主服务器通过配置文件 *~/.gitolite.rc* 中的变量 *$ENV{GL_SLAVES}* 设置镜像服务器的地址。
- 从服务器通过配置文件 *~/.gitolite.rc* 中的变量 *$GL_SLAVE_MODE* 设置从服务器模式。
- 从主服务器端运行脚本 *gl-mirror-sync* 可以实现批量的版本库镜像。
- 主服务器的每一个版本库都配置 *post-receive* 钩子，一旦有提交，即时同步到镜像版本库。

在多个服务器之间设置 Git 库镜像的方法是：

- 每个服务器都要安装 Gitolite 软件，而且要启用 *post-receive* 钩子。

  缺省的钩子在源代码的 *hooks/common* 目录下，名称为 *post-receive.mirrorpush* ，要将其改名为 *post-receive* 。否则版本库的 *post-receive* 脚本不能生效。

- 主服务器配置到从服务器的公钥认证，并且配置使用特殊的 SHELL： *gl-mirror-shell* 。

  这是因为主服务器在向从服务器同步版本库的时候，如果从服务器版本库没有创建，直接通过 SSH 登录到从服务器，执行创建命令。因此需要通过一个特殊的SHELL，能够同时支持 Gitolite 的授权访问以及 SHELL 环境。这个特殊的 SHELL 就是 *gl-mirror-shell* 。而且这个 SHELL，通过特殊的环境变量绕过服务器的权限检查，避免因为授权问题导致同步失败。

  实际应用中，不光主服务器，每个服务器都进行类似设置，目的是主从服务器可能相互切换。

  在 Gitolite 不同的安装模式下， *gl-mirror-shell* 的安装位置可能不同。下面的命令用于在服务器端设置其他服务器访问时使用这个特殊的 SHELL。

  假设在服务器 foo 上，安装来自服务器 bar 和 baz 的公钥认证。公钥分别是 bar.pub 和 baz.pub。

  - 对于在客户端安装方式部署的 Gitolite：

```
# 在服务器 foo 上执行:
$ export GL_ADMINDIR=` cd $HOME;perl -e 'do ".gitolite.rc"; print $GL_ADM
$ cat bar.pub baz.pub |
    sed -e 's,^,command="'$GL_ADMINDIR'/src/gl-mirror-shell" ,' >> ~/.ssh
```

  - 对于在服务器端安装方式部署的 Gitolite， *gl-mirror-shell* 直接就可以在路径中找到。

```
# 在服务器 foo 上执行:
$ cat bar.pub baz.pub |
    sed -e 's,^,command="'$(which gl-mirror-shell)'" ,' >> ~/.ssh/authori
```

  在 foo 服务器上设置完毕，可以从服务器 bar 或者 baz 上远程执行：

  - 执行命令后退出

```
$ ssh git@foo pwd
```

  - 进入 shell

```
$ ssh git@foo bash -i
```

- 在从服务器上设置配置文件 *~/.gitolite.rc* 。

  进行如下设置后，将不允许用户直接 PUSH 到从服务器。但是主服务器仍然可以 PUSH 到从服务器，是因为主服务器版本库在 PUSH 到从服务器时，使用了特殊的环境变量，能够跳过从服务器版本库的 *update* 脚本。

  ```
  $GL_SLAVE_MODE = 1
  ```

- 在主服务器上设置配置文件 *~/.gitolite.rc* 。

  需要配置到从服务器的 SSH 连接，可以设置多个，用空格分隔。注意使用单引号，避免 @ 字符被 Perl 当作数组解析。

  ```
  $ENV{GL_SLAVES} = 'gitolite@bar gitolite@baz';
  ```

- 在主服务器端执行 *gl-mirror-sync* 进行一次完整的数据同步。

  需要以 Gitolite 安装用户身份（如git）执行。例如在服务器 foo 上执行到从服务器 bar 的同步。

  ```
  $ gl-mirror-sync gitolite@bar
  ```

- 之后，每当用户向主版本库同步，都会通过版本库的 *post-receive* 钩子即时同步到从版本库。
- 主从版本库的切换。

  切换非常简单，就是修改 *~/.gitolite.rc* 配置文件，修改 *$GL_SLAVE_MODE* 设置：主服务器设置为 0，从服务器设置为1。

2.7.2 Gitweb 和 Gitdaemon 支持

Gitolite 和 git-daemon 的整合很简单，就是在版本库目录中创建一个空文件 *git-daemon-export-ok* 。

Gitolite 和 Gitweb 的整合，则提供了两个方面的内容。一个是可以设置版本库的描述信息，用于在 gitweb 的项目列表页面显示。另外一个是自动生成项目的列表文件供 Gitweb 参卡，避免 Gitweb 使用效率低的目录递归搜索查找 Git 版本库列表。

可以在授权文件中设定版本库的描述信息，并在 gitolite-admin 管理库更新时创建到版本库的 description 文件中。

```
reponame = "one line of description"
reponame "owner name" = "one line of description"
```

- 第1行，为名为 *reponame* 的版本库设定描述。
- 第1行，同时设定版本库的属主名称，和一行版本库描述。

对于通配符版本库，使用这种方法则很不现实。Gitolite 提供了 SSH 子命令，供版本库的创建者使用。

```
$ ssh git@server setdesc description...
$ ssh git@server getdesc
```

- 第一条指令用于设置版本库的描述信息。
- 第二条指令显示版本库的描述信息。

至于生成 Gitweb 所用的项目列表文件，缺省创建在用户主目录下的 *projects.list* 文件中。对于所有启用 Gitweb 的 [repo] 小节设定的版本库，或者通过版本库描述隐式声明的版本库加入到版本库列表中。

2.7.3 其他功能拓展和参考

Gitolite 源码的 doc 目录包含用 *markdown* 标记语言编写的手册，可以直接在 *Github* 上查看。也可以使用 *markdown* 的文档编辑工具将 *.mkd* 文档转换为 html 文档。转换工具很多，有：rdiscount, Bluefeather, Maruku,

BlueCloth2 等等。

在这些参考文档中，你可以发现 Gitolite 包含的更多的小功能或者秘籍，包括：

- 版本库设置。

  在授权文件通过 git config 指令为版本库进行附加的设置。例如：

  ```
  repo gitolite
      config hooks.mailinglist = gitolite-commits@example.tld
      config hooks.emailprefix = "[gitolite] "
      config foo.bar = ""
      config foo.baz =
  ```

- 多级管理员授权。

  可以为不同版本库设定管理员，操作 gitolite-admin 库的部分授权文件。参见： *doc/5-delegation.mkd* 。

- 自定义钩子脚本。

  因为 Gitolite 占用了几个钩子脚本，如果需要对同名钩子进行扩展，Gitolite 提供了级联的钩子脚本，将定制放在级联的钩子脚本里。

  例如：通过自定义 gitolite-admin 的 post-update.secondary 脚本，以实现无需登录服务器，更改 *.gitolite.rc* 文件。参见： *doc/shell-games.mkd* 。

  关于钩子脚本的创建和维护，参见： *doc/hook-propagation.mkd* 。

- 管理员自定义命令。

  通过设置配置文件中的 *$GL_ADC_PATH* 变量，在远程执行该目录下的可执行脚本，如: *rmrepo* 。

  具体参考： *doc/admin-defined-commands.mkd* 。

- 创建匿名 SSH 认证。

  允许匿名用户访问 Gitolite 提供的 git 服务。即建立一个和 gitolite 服务器端帐号同 id 和主目录的用户，并设置其的特定 shell，并且允许口令为空。

  具体参考： *doc/mob-branches.mkd* 。

- 可以通过名为 @all 的版本库进行全局的授权。

  但是不能在 @all 版本库中对 @all 用户组进行授权。

- 版本库非常或者用户非常之多（几千个）的时候，需要使用 **大配置文件** 模式。

  因为 Gitolite 的授权文件要先编译才能生效，而编译文件的大小是和用户以及版本库数量的乘积成正比的。选择大配置文件模式，则不对用户组和版本库组进行扩展。

  参见： *doc/big-config.mkd* 。

- 授权文件支持包含语句，可以将授权文件分成多个独立的单元。
- 执行外部命令，如 rsync。
- Subversion 版本库支持。

  如果在同一个服务器上以 svn+ssh 方式运行 Subversion 服务器，可以使用同一套公钥，同时为用户提供 Git 和 Subversion 服务。

- HTTP 口令文件维护。通过 htpasswd SSH 子命令实现。
