Lutts' Diary

简单、务实、美

Lutts Wolf

lutts.cao@gmail.com

November 28, 2012

—— 生活有三原色就够了，又何必一定要求彩虹

Contents

List of Tables

Android Development Environment Setup

shows howto setup the Android development environment, the issue(errors) I meet, and how I solved it. This includes:

- source code download
- build and evaluate
- eclipse development environment setup

主要参照 android 的官网 (`http://source.android.com/source/index.html`)。

我的环境:

- ‘uname -a‘: Linux lutts-server 2.6.35-30-generic #61 lucid1-Ubuntu SMP Thu Oct 13 19:29:42 UTC 2011 x86_64 GNU/Linux
- ‘python --version‘: Python 2.6.5
- ‘java -version‘: java version ”1.6.0_26” Java(TM) SE Runtime Environment (build 1.6.0_26-b03) Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode)
- ‘git –version‘: git version 1.7.0.4
- 100G 的 Android 专属分区

下面是我的步骤：

1. 安装 **Java 1.6**

```
$ sudo add-apt-repository “deb http://archive.canonical.com/ lucid partner”
$ sudo add-apt-repository “deb-src http://archive.canonical.com/ubuntu lucid partner”
$ sudo apt-get update
$ sudo apt-get install sun-java6-jdk
```

要安装 Java 1.5 的话 (如果你想编译 froyo 早期及更早的 android 版本), 使用以下命令安装：

```
$ sudo add-apt-repository “deb http://archive.ubuntu.com/ubuntu dapper main multiverse”
$ sudo add-apt-repository “deb http://archive.ubuntu.com/ubuntu dapper-updates main multiverse”
$ sudo apt-get update
$ sudo apt-get install sun-java5-jdk
```

你可以随时切换 Java 的版本，相关命令如下

```
sudo update-alternatives –config java
```

在出现的选项中选择你需要的版本即可。

你也许还需要执行以下命令 (这些命令和编译 Android 无关，因此如果报错的话，无视之)

```
sudo update-java-alternatives -s java-1.5.0-sun
sudo update-java-alternatives -s java-6-sun
```

其中 -s 后机的参数是/usr/lib/jvm 目录下相应的目录名

2. 安装各种工具包

按官网所说的做，命令如下

```
sudo apt-get install git-core gnupg flex bison gperf build-essential \
zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs \
x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
libgl1-mesa-dev g++-multilib mingw32 tofrodos
```

3. 下载源代码

3.1. Installing Repo

- 先给 repo 找个家

```
$ mkdir ~/bin
$ export PATH=~/bin:$PATH
```

为了避免以后都要设置 PATH 变量，最好将以下代码加入.bashrc 中

```
export PATH=$HOME/bin:$PATH
```

- 下载 repo 并添加可执行权限

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

- The MD5 checksum for repo is bbf05a064c4d184550d71595a662e098

3.2. Initializing a Repo client, 为下载做准备

- 创建一个目录用于存放 android 的源代码，这里我新建一个 android-2.3 目录, 当然，名字可以随便取。

使用 root 用户执行以下命令：

```
$ mkdir android-2.3
$ cd android-2.3
```

- 初始化 repo

使用 root 用户执行以下命令：

```
$ repo init -u https://android.googlesource.com/platform/manifest –mirror
```

这里使用了 --mirror，本地的目录结构将按照源的版本库组织方式进行组织，否则会按照 manifest.xml 指定的方式重新组织并检出到本地。这就要体现在以下几个方面：

  - 使用 --mirror 得到的将是 xxx.git 形式目录，也就是原始的 git repository.
  - 在 manifest.xml 的 project 语法中，name 实际就是 git repository 在源的版本库中的路径 (需加上一个前缀)，不使用 mirror 时，会 check 到 path 指定到的目录

之所以要使用 mirror，主要是考虑到以下几点:

  - 我可能需要在多台机器上工作，如果每台都从官网 sync 就太费时了
  - 我是一个很小心的人，尽管有 git 帮忙管理代码，但人总会犯错呀，多留个备份总不会有错的，所以这里用 root sync 一个原始库，然后再用普通用户 clone 出来作为工作目录。
  - 把 android 代码 clone 下来之后，再自行用 git 管理，不也是一种方法么？但这种方法有其不方便之处。因为 android 很大，如果整个 android 作为一个 git repo 的话，不论是 git diff, git status 之类的命令都会很费时。怎么说呢，

Android 很像是一大堆可以各自独立维护的软件/库的综合体，因此各个软件包分开来独立维护会更为方便，我想这也是 repo 产生的原因吧。

- 开始下载源码，执行以下命令：

```
$ repo sync
```

这个是支持断点续传的，如果中间停了，再执行一次就行。(不知道为什么，我第一次执行时在最后时刻一直停在那，Ctrl+C 都不行，最后使用 kill -9 才搞定，然后重新 repo sync 就好了)
总计要下载 2.5G 多.
官网上什么 gpg 验证什么的就算了吧，不需要验证，一般不会有错的。

- 前面得到的只是原始库，不能直接用来编译，需执行以下命令:

```
repo init -u /source/android-2.3/platform/manifest \
--repo-url /source/android-2.3/tools/repo
```

4. 初体验

4.1. 编译之前执行以下命令初始化环境:

```
$ source build/envsetup.sh
```

4.2. Choose a Target, 执行以下命令：

```
$ lunch full-eng
```

直接执行 lunch 会提供一个选择单，但你会发现 full-eng 不在选择单里，事实上选择单里的只是典型的选项，lunch 命令的 target 可以是 buildname 和 buildtype 的任意组合。

4.3. 现在可以 make 了，我使用 make -j16，按官方说法，这里的 16 是硬线程 (hardware thread) 数的两倍，因为我的机器是 4 核，支持超线程 (每核两线程)，就是 4*2*2=16

tips:

- 我发现其实 < 硬线程数＋ 1> 似乎要比 <2* 线程数 > 要快一些^o^.
- 我的机器是 I7 2600K，8G 内存，编译 Android2.3.4 约需要 15 分钟，其间内存占用一度达到 7G, 编译 Android 4.0.1 用了 35 分钟！
- 编译成功完成后在 out/target/product/generic 目录下会有 system.img, userdata.img, ramdisk.img 三个文件出现
- 最好保存好编译 log，以后出了问题或是想知道都编译了什么可以有据可查

```
$ make -j9 | tee android-make.log
```

4.4. 用模拟器运行:

```
$ emulator &
```

模拟器运行需要四个文件，分别是 Linux Kernel 镜像 zImage 和 Android 镜像文件 system.img、userdata.img 和 ramdisk.img。执行 emulator 命令时，如果不带任何参数，则 Linux Kernel 镜像默认使用 prebuilt/android-arm/kernel 目录下的 kernel-qemu 文件，而 Android 镜像文件则默认使用 ANDROID_PRODUCT_OUT 目录下的 system.img、

userdata.img 和 ramdisk.img，也就是我们刚刚编译出来的镜像问题。当然，我们也可以以指定的镜像文件来运行模拟器，即运行 emulator 时，即：

```
$ emulator -kernel ./prebuilt/android-arm/kernel/kernel-qemu -sysdir ./out/target/product/
generic -system system.img -data userdata.img -ramdisk ramdisk.img &
```

启动速度可能会有点慢，耐心等待一会界面就出来了，如下：

Figure 1.1: Android Emulator

也可用使 ddms 来调试：

```
$ ddms &
```

Tips

按上述步骤编完 android 后，ddms 启动正常，但第二天却发现启动 ddms 的时候出现以下错误：

```
$ ddms &
SWT folder '/home/lutts/android/mydroid/out/host/linux-x86/framework/x86_64' does not
exist.
Please export ANDROID_SWT to point to the folder containing swt.jar for your platform.
$
```

遍寻 google 不果，仔细想想为什么第一次可以呢，呵呵，原来只要再执行以下命令就好了：

```
$ lunch full-eng
```

其中的 full-eng 是我当初编译 Android 时使用的配置。

5. 自己编译内核

5.1. 下载 kernel 源码，首先进入你的 work 目录 (我的是~/android/mydroid)，新建一个 kernel 目录，将 common check 下来.

```
$ cd ~/android/mydroid
$ mkdir kernel
$ cd kernel
$ git clone git://android.git.kernel.org/kernel/common.git
```

这里我选择了 common.git，因为我想基于这个进行一些开发学习。你也可以 check 其他的，具体有哪些请到`http://android.git.kernel.org/`查看, kernel 开头的 git tree 都是 kernel。

先查看一下有哪些 branch：

```
$ git branch -a
* (no branch)
android-2.6.36
remotes/origin/HEAD -> origin/android-2.6.36
remotes/origin/android-2.6.35
remotes/origin/android-2.6.36
remotes/origin/android-2.6.37
remotes/origin/android-2.6.38
remotes/origin/android-2.6.39
remotes/origin/android-3.0
remotes/origin/archive/android-2.6.25
remotes/origin/archive/android-2.6.27
remotes/origin/archive/android-2.6.29
remotes/origin/archive/android-2.6.32
remotes/origin/archive/android-gldfish-2.6.29
remotes/origin/archive/android-goldfish-2.6.27
remotes/origin/linux-bcm43xx-2.6.39
remotes/origin/linux-wl12xx-2.6.39
```

android-gldfish-2.6.29 这个版本可用于模拟器。

```
$ git checkout remotes/origin/archive/android-gldfish-2.6.29
```

5.2. 编译内核代码

- 导出交叉编译工具目录到 $PATH 环境变量中去

```
$ export PATH=$PATH:~/android/mydroid/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin
```

- 修改 common 目录下的 Makefile 文件的以下两行为：

```
# ARCH ?= (SUBARCH)
# CROSS_COMPILE ?=
ARCH ?= arm  # 体系结构为 arm
CROSS_COMPILE      ?= arm-eabi- # 交叉编译工具链前缀，参考 prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin 目录
```

- 开始编译

```
$ make goldfish_defconfig
$ make -j16
```

编译完成会生成 kernel/common/arch/arm/boot/zImage 文件

- 启动模拟器

```
$ emulator -kernel ./kernel/common/arch/arm/boot/zImage &
```

- 用 adb 工具连接模拟器，查看内核版本信息，看看模拟器上跑的内核是不是我们刚才编译出来的内核，在系统起来之后，运行以下命令

```
$ adb shell
```

这时候如果是第一次运行 adb shell 命令，会看到以下输出，不用管它，稍等一会，再运行一次 adb shell 命令就可以了，如果再不行，就再运行一次，直到行为止^o^

```
* daemon not running. start it now on port 5037 *
* daemon started successfully *
error: device offline
```

切换到 proc 目录：

```
$ adb shell
# cd proc
# cat version
Linux version 2.6.29-gb0d93fb-dirty (lutts@lutts-server) (gcc version 4.4.3 (GCC) ) #1 Tue Aug 23 22:21:35 CST 2011
#
```

从机器名 lutts@lutts-server 和日期 Tue Aug 23 22:21:35 CST 2011 可以看出，模拟器使用的内核即为刚刚编译出来的内核。

Android Development Environment Setup(Cont.)

这一部分主要涉及 eclipse 环境的搭建。
主要参考文档：

1. `http://source.android.com/source/using-eclipse.html`

步骤：

1. 拷贝 development/ide/eclipse/.classpath 到源代码根目录

```
$ cd /path/to/android/root
$ cp development/ide/eclipse/.classpath ./
$ chmod u+w .classpath
```

这样一来相当于将源码目录变成了一个 eclipse project。每个 eclipse project 下都有一个.classpath，当然还需要.project 等文件，这些稍后使用 eclipse 来创建

.classpath 需要稍作修改，否则 eclipse build 时会报错

```
Project 'gingerbread' is missing required library: 'out/target/common/obj/JAVA_-
LIBRARIES/google-common_intermediates/javalib.jar'
Project 'gingerbread' is missing required library: 'out/target/common/obj/JAVA_-
LIBRARIES/gsf-client_intermediates/javalib.jar'
```

解决办法：

删除掉以下两行 (位于文件的结尾处):

```
<classpathentry kind="lib" path="out/target/common/obj/JAVA_LIBRARIES/google-common_intermediates/javalib.jar"
<classpathentry kind="lib" path="out/target/common/obj/JAVA_LIBRARIES/gsf-client_intermediates/javalib.jar"/>
```

添加以下一行：

```
<classpathentry kind="lib" path="out/target/common/obj/JAVA_LIBRARIES/
android-common_intermediates/javalib.jar"/>
```

注：对于 android-4.0，提示找不到 Effect 和 EffectFactory, 需要加入下面的 lib:

```
<classpathentry kind="lib" path="out/target/common/obj/JAVA_LIBRARIES/
filterfw_intermediates/classes-jarjar.jar"/>
```

(注：为方便阅读，上面的一行分为了两行书写，实际使用时请写为一行)

2. eclipse 配置 (eclipse.ini)

我的 eclipse 版本信息:

Eclipse Java EE IDE for Web Developers.

Version: Indigo Release

Build id: 20110615-0604

在启动 eclipse 前，需要重新配置内存设置，否则会出现 “GC overhead limit exceeded” 和 “Out of Memory Error”:

以下为 eclipse 的默认值:

```
-XX:MaxPermSize=256m
-Xms40m
-Xmx256m
```

修改为:

```
-XX:MaxPermSize=512m
-Xms256m
-Xmx1024m
```

上面这些值要视你的总内存大小而定。

步骤:

(a) 新建一个 Java 工程

(File -> New -> Java Project, eclipse 首次使用时，是在 File -> New -> Project...，进去之后就有 Java Project 了)。因为我们需要从已有工程添加，因此去掉 Use Default Location，选择自己的 android 根目录。

然后按 Finish，会触发 Eclipse 开始 build，时间可能较长，耐心等待。

build 结果: 0 errors, 10,594 warnings, 0 others

这此在 Android 的根目录下会多出一个.project 文件，eclipse build 的输出在.classpath 文件的最后一行 output 指定的目录里。

(b) Eclipse formatting

You can import files in development/ide/eclipse to make Eclipse follow the Android style rules.

i. Select Window > Preferences > Java > Code Style.
ii. Use Formatter > Import to import android-formatting.xml.
iii. Organize Imports > Import to import android.importorder

(c) Debugging the emulator with Eclipse

You can also use eclipse to debug the emulator and step through code. First, start the emulator running:

```
cd /path/to/android/root
. build/envsetup.sh
lunch full-eng (NOTE: select whatever you configured, for example: full-eng)
make
emulator &
ddms &
```

if the emulator is running, you should see a picture of a phone.

the ‘ddms‘ command will start DDMS (the Dalvik debug manager), you should see a splufty debugging console.

Now, in eclipse, you can attach to the emulator:

i. Run > Debug Configurations...
ii. Double click “Remote Java Application” in the left panel.
iii. Pick a name, i.e. ”android-debug” or anything you like.
iv. Set the ”Project” to your project name.
v. Keep the Host set to ”localhost”, but change Port to 8700.

vi. Click the ”Debug” button and you should be all set.

Note that port 8700 is attached to whatever process is currently selected in the DDMS console, so you need to sure that DDMS has selected the process you want to debug.

You may need to open the Debug perspective (next to the ”Java” perspective icon in the upper-right, click the small ”Open Perspective” icon and select ”Debug”). Once you do, you should see a list of threads; if you select one and break it (by clicking the ”pause” icon), it should show the stack trace, source file, and line where execution is at. Breakpoints and whatnot should all work.

3. 编译并安装 ADT

如果还没能执行 envsetup.sh，需要在编译前执行以下命令:

```
$ . envsetup.sh
$ lunch full-eng
```

然后可以执行以下命令来生成 ADT 插件：

```
$ export ECLIPSE_HOME=<your eclipse root dir>
$ mkdir /home/lutts/android/adt
$ ./sdk/eclipse/scripts/build_server.sh /home/lutts/android/adt/
```

执行成功后在/home/lutts/android/adt 目录下会有一个 zip 文件。这就相当于从网上下了一个 ADT zip 包了。

注：事实证明，源码包编出来的 ADT 太旧了！配置 SDK 时报错:

Could not find /home/lutts/android/mydroid_sdk/android-sdk_eng.lutts_linux-x86/tools/adb!

不得已还是得从网上下载, `http://dl.google.com/android/ADT_9.0.0.zip`

安装 ADT 到 eclipse:

在此过程中要确保联网, 因为有些 dependency 可能需要从网上下载，安装过程中可能会卡在 “Calculating requirements and dependencies” 和 “Installing Software” 这两步很久，要视你的网速而定，请耐心等待。

以下为详细步骤：

(a) 启动 Eclipse，然后选择 Help > Install New Software。

(b) 在 Available Software 对话框中，单击 Add…。

(c) 在 Add Repository 对话框，点击 Archive... 按钮，选择之前生成的 zip 文件

注：如果安装失败了，再次选择 zip 文件时会提示 “Duplicate location”, 点取消，然后在 Available Software 界面点那个 Available Software Sites 进入到里面把你前面添加的 remove 掉

(d) 回到 Available Software 界面，应该可以看到列表中的 Developer Tools 选项。选择 Developer Tools 旁边的复选框 (或者点示 Select All 按钮)，会同时选中 Android DDMS 和 Android Development Tools。单击 Next。

(e) 在最后的 Install Details 对话框中，会列出 Android DDMS 和 Android Development Tools 等特性。阅读并接受许可协议，单击 Next，同时安装所有依赖组件，然后单击 Finish。会开始 Installing Software。

(f) 安装完后会提示重启 eclipse，重启之后在 Help –> About Eclipse 里确认一下插件是否安装成功。

这里需要先提个醒，一旦安装成功 ADT，以后调试时需要先启动 ddms，后启动 eclipse，否则会报错 “Could not open Selected VM debug port (8700)…”

4. 编译并配置 SDK

```
$ make sdk
```

这个过程不是很长。

编译结束打印以下信息:

```
Package SDK Stubs: out/target/common/obj/PACKAGING/android_jar_intermediates/android.jar
Package SDK: out/host/linux-x86/sdk/android-sdk_eng.lutts_linux-x86.zip
```

也就是说，我们的 SDK 生成在 out/host/linux-x86/sdk/目录下.

注意：当用 mmm 命令编译模块时，一样会把 SDK 的输出文件清除，因此，最好把 android-sdk_eng.xxx_linux-x86 移出来

```
cp -r mydroid/out/host/linux-x86/sdk/* /home/lutts/android/mydroid_sdk/
```

配置 eclipse: 打开 Window –> Preference，在左边选择 Android，然后在右边选择好你的 SDK Location，单击 OK 退出。

5. Hello World!

Git server setup using SSH and Gitolite on ubuntu

Contents

安装 git

```
$ sudo apt-get install git-core
```

Please note that gitolite needs at least version 1.6.2 of git. If your distribution contains n older version(this is the case of Ubuntu 8.04), use a backport packag, or install git via source.

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

*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。

SSH Basic

对于拥有 shell 权限的 SSH 登录账号, 可以使用下面的命令访问 git:

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

其中:

- <username> 是服务器 <server> 上的用户账号
- /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, 每次连接时都会提示输入;
- 如果采用公钥认证, 则无需每次都输入口令

SSH 协议来实现 Git 服务, 有如下方式:

- 其一是用标准的 ssh 账号访问版本库. 以用户账号可以直接登录到服务器, 获得 shell
- 另外的方式是, 所有用户都使用专用的 ssh 账号访问版本库. 各个用户通过公钥认证的方式用些专用 SSH 账号访问版本库. 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 账号和专用 SSH 账号的区别在于:

	标准 SSH	GITOSIS/GITOLITE
账号	每个用户一个账号	所有用户共用同一个账号
认证方式	口令或公钥认证	公钥认证
用户是否以直接登录 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账号。不过这类似于把一个公共账号的登录口令同时告诉给多个人。

- 在服务端创建一个公共账号，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
```

  如果在服务器上操作，则将文件追加到 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账号访问 git 服务。

这样做除了免除了靛一设置账号，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。

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
```

Just press enter. If asked for a passphrase, just press enter: for internal home connections, you will not need a passphrase, and it will break the procedure of having a seamless remote application.

该命令会在用户主目录下创建建.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 username@server
```

或

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

说明:

- 该命令会提示输入用户 user 在 server 上的 SSH 登录口令。不想要口令的话，直接回车即可。
- 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:

```
$ ssh username@server
```

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。

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>
```

说明：

- 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 dev-server
        user lutts
        hostname 192.168.100.200
        port 22
        identityfile ~/.ssh/id_rsa
host gitadmin
        user gitolite
        hostname 192.168.100.200
        port 22
        identityfile ~/.ssh/gitadmin
```

当执行 ssh gitadmin 或 git clone gitbox:path/to/repo.git 时，其实是使用 gitolite 用户登录 192.168.100.200，认证时使用的公钥文件为~/.ssh/gitadmin.pub

Gitolite 服务器架设 (on ubuntu)

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。

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 (注: 安装过程会自动检出到 HOME 下)。

```
$ 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 commit 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-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa < 公钥内容来自于 xxx.pub ...>
  - 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。

First you will need to setup the git user account on the remote machine, under which gitolite will run, so login.

```
$ ssh dev-server
```

dev-server is alias of my remote machine I am using. You can replace this with your remote machine's hostname or IP.

Now create the user. I'm calling my user "gitolite", but you can use "git" or anything else.

On Server:

```
$ sudo adduser –system –shell /bin/sh –gecos 'gitolite version control' –group –disabled-password
–home /source/gitolite gitolite
Adding system user 'gitolite' (UID 119) ...
Adding new group 'gitolite' (GID 125) ...
Adding new user 'gitolite' (UID 119) with group 'gitolite' ...
Creating home directory '/source/gitolite' ...
```

In this example above */source/gitolite* is where gitolite and your code repositories will live.

you can remove the user with command 'sudo deluser –system –remove-home gitolite'

Now you can return to your local machine.

Notice that when we created the user, we used –disable-password, which prevents us logging into the machine using a password.

Therefore we'll need to upload a ssh key for running the installer.

Now we need to generate a ssh key for gitolite administration, you can use your existing id_rsa.pub, but here, I will create a public and private keypair with the name "gitadmin".

On local machine:

```
cd  /.ssh
ssh-keygen -t rsa -f gitadmin
cd
```

When prompted for a password, just press enter to enable passwordless logins. It is recommended that you activate a password for this account after the gitlite installation, as this certificate enables a login to the shell on your server. This can be done via ssh-keygen -p.

You public key can be found here:

```
~/.ssh/gitadmin.pub
```

And the private key here:

```
~/.ssh/gitadmin
```

Now you'll need to upload the public key to 'git' user account, so that we can log into that account using our private key.

```
scp ~/.ssh/gitadmin.pub dev-server:~/
```

Now login to the remote machine

```
ssh dev-server
```

Become the gitolite user

```
sudo su - gitolite
```

```
git clone git://github.com/sitaramc/gitolite
cd gitolite
src/gl-system-install
```

after execute "gl-system-install", two directies, "bin" and "share" will be create in gitolite's home folder.

To upgrade gitolite, repeat the above commands. Make sure you use the same arguments for the last command("gl-system-install") each time. Here we have no arguments, aka, using the

defaults.

```
export PATH=$PATH:$HOME/bin
gl-setup -q ~lutts/gitadmin.pub
```

Note that gl-setup is in the new-created *bin* directory, so we must all bin the PATH environment. To avoid set it everytime, you can add it to ".profile".

gitadmin will be the username to administer the gitlite installation.

gl-setup will append gitadmin.pub(with command restrictions) to *.ssh/authorized_keys*, the old one will renamed to *old_authkeys*.

After the wizard has finished, the following directories and files will be created:

- **repositories**, with "gitolite-admin.git" and "testing.git" in it.
- **projects.list**, Gitweb 所用的项目列表文件.
- **.gitolite**
- **.gitolite.rc**

Now return to your local machine.

To make things simple on ssh side I recommend adding the configuration for the gitolite account to you ssh config.

```
vim ~/.ssh/config
```

Add the following:

```
host gitadmin
        user gitolite
        hostname git.luttsoft.com
        port 22
        identityfile ~/.ssh/gitadmin
```

Now you should be able to login to the remote machine as the gitolite user using the following:

```
ssh gitadmin
```

```
$ ssh gitbox
PTY allocation request failed on channel 0
hello gitadmin, this is gitolite v2.1-0-g871ed28 running on git 1.7.0.4
the gitolite config gives you the following access:
     R   W  gitolite-admin
    @R_ @W_ testing
Connection to git.luttsoft.com closed.
```

As you can see, users will not get shell access to the server. When they try to login via ssh with the certificate they will only get a list of all repositories they may access. With gitolite you now have the option to add a number of users without the need to create accounts on your server.

Check out the gitolite-admin in some directory:

```
git clone gitbox:gitolite-admin
```

This will be used for managing your users and git repositories. By simply editing conf/gitolite.conf and pushing it to the gitolite server you can create new repositories. Adding new users will involve adding an ssh key to the keydir. I will cover more on these in a follow-up post.

Tips

- 有一个 gitolite-tools(https://github.com/tmatilai/gitolite-tools.git)，可以用来

  1. git gl-info - Display gitolite server information
  2. git gl-ls - List accessible gitolite repositories
  3. git gl-desc - Display or edit description of gitolite wildcard repositories
  4. git gl-perms - Display or edit permissions of gitolite wildcard repositories
  5. git gl-htpasswd - Set password for gitweb/apache

  需要注意的是：上面的所有命令都只能在从 gitolite 中 clone 出来的 repo 下运行，否则报错

Git Quick Reference

git 很难学，命令好多好多，堪比 emacs，因此学习方法上也大致一样：先大致学习一遍，然后从实际使用中慢慢积累经验。

首先要知道如何看帮助，git 的帮助还不错的，相关命令如下：

git help <command>: or

git <command> –help: Show help for a *command*.

通常的工作流程是:

1. 创建本地 (local) 库
2. 更新 (update) 共享 (shared/remote) 库里的内容到本地
3. Make changes(new, modify, remove, rename, etc.), 并将一个 topic 的阶段性工作放到 staging area;
4. commit changes，一个 commit 一个主题，切忌一次 commit 涉及到多个主题
5. review(amend) commits，看一下都 commit 了什么，是否可以删了某些 commit(因为没用了), 是否可以合并某些 commit，各个 commit 的顺序是否需要调整一下以便更合逻辑。
6. push 本地修改到共享库里

下面我们就这些内容依次进行讲解。

在讲解前，首先介绍一下 staging area,staging area 是一个暂存区 (cache)，比如说为了改个 bug, 某个文件你要修改三处 (想象成分三个阶段 [stage] 修改)，每修改一处，测试一下，ok 了就可以放到 staging area，再修改第二处，发现改得不对，checkout 一下就可以回到第一个 stage 了。

Create

From existing data:

```
cd ~/projects/myprojects
git init
git add .
```

以 push 方式向 gitolite 新增 repository 还需要执行以下命令:

```
git remote add origin <url>
```

然后就可以用 git push 到 gitolite 了。这样新建的 repo 在 pull 的时候会报错，因为 branch master 的设置没有 (查看.git/config 可以看到只有 merge “ours” 这样的 section), 使用以下命令：

```
$ git config --remove-section merge.ours
$ git config branch.master.remote origin
$ git config branch.master.merge refs/heads/master
```

From existing repo:

```
git clone ~/existing/repo ~/new/repo
git clone git://host.org/project.git ~/new/repo
git clone ssh://you@host.org/project.git ~/new/repo
git clone http://git.software.org/project.git ~/new/repo
```

Update

git fetch: (不带任何参数)fetch latest changes from origin(but does not merge them).

fetch 下来的 commit 存放在 remotes/origin/master 里, 可以 git checkout remotes/origin/master 切换到 remote 的 master 进行查看，如果没有问题，用 git checkout -f master 切换到 local master, 然后 git merge remotes/origin/master 进行 merge

git pull: fetch + merge

git am -3 patch.mbox: Apply a patch that someone send you(by email)

incase of a conflict, resolve and use git am –resolved.

git apply –reject patch.diff:

打 patch，如果失败，生成 *.rej 文件

git diff

```
git diff [<common diff options>] <commit>{0,2} [--] [<path>...]
```

git diff: 比较当前 work tree 和 staging area

git diff --cached: 比较 staging area 与未修改前 repository 的区别

git diff HEAD: 比较当前 work tree 与未修改前 repository 的区别 (git diff + git diff –cached)

git diff HEAD~1: 与 HEAD 的上一个 commit 比较

git diff other-branch: 比较当前 work tree 与 other-branch 的区别

git diff master: 比较当前 work tree 和 master 的区别

git diff tag1 tag2: tag1 和 tag2 的 diff

git diff tag1:file1 tag2:file2:

tag1 中的 file1 和 tag2 中的 file2 的区别

git diff --stat: 以汇总的形式显示，例如:

```
file1.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

可以将产生的 patch 重定向到文件中并发送给其他人:

```
git diff -a tag1 tag2 >some.patch
```

使用 -a 选项能确保二进制文件也会包含在 some.patch 中

接收方必须使用 p1 来打 patch:

```
patch -Np1 -i path/to/some.patch
```

staging

git add .: 将当前目录下所有 (修改过或新增的) 文件先暂存到 staging area

git add filename: 将文件 filename(修改过的或新增的) 暂存到 staging area

git add -u: 将修改过的文件暂到到 staging area，新增的文件不暂存

git add -i: 进入互动模式

删除文件

git rm filename: 删除文件 (注意: 文件会立即从所在的文件夹消失)

注: 删起来简单，如果 stage 了，要恢复比较麻烦，以 file1.txt 为例，需分两步

1. git reset HEAD – file1.txt
2. git checkout – file1.txt

还原已删除的文件

git ls-files -d: 查看已删除的文件列表

git ls-files -d | xargs git checkout --:
将已删除的文件全部还原 (注：如果 stage 了需要先 reset)

文件重命名

git mv filename new-filename:

需要注意: 本地的文件立即被重命名，无须 commit.

而且要恢复比 rm 更麻烦，以将 file1.txt 重命名 file1-1.txt 为例，需三步:

1. git reset HEAD – file1.txt file1-1.txt
2. git checkout – file1.txt
3. rm file1-1.txt

请注意 file1-1.txt 先 reset 后 rm 了

查看状态

git status: (nothing to explain)

commit

git commit: 这是推荐的用法。会打开 editor 让你编辑 commit message. some of Git's viewing tools need commit messages in the following format:

```
A brief one-line summary(short message, less than 50 characters)
<blank line>
Details about the commit
```

git commit -m 'commit message':
commit message 由 -m 指定

git commit -a -m 'commit message':
将所有修改过的文件都 commit.

注: 新增的档案还是需要先 add

git commit -a -v: 会在打开的 editor 里显示 git diff HEAD 的结果

若发现提交有问题，比如说 commit message 写得不是很对，可以使用 –amend 进行修改后重新 commit.

```
git commit --amend
```

此命令将使用当前的暂存区域快照提交。如果刚才提交完没有作任何改动，直接运行此命令的话，相当于有机会重新编辑提交说明，但将要提交的文件快照和之前的一样。

启动文本编辑器后，会看到上次提交时的说明，编辑它确认没问题后保存退出，就会使用新的提交说明覆盖刚才失误的提交。

如果刚才提交时忘了暂存某些修改，可以先补上暂存操作，然后再运行 –amend 提交:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

上面的三条命令最终只是产生一个提交，第二个提交命令修正了第一个的提交内容。

tag&branch

在 git 中，tag 和 branch 都只是一个标志 (相当于 C 指针)，一般对每个 release 版本做一个 tag，或者在你认为 (开发过程中) 重要的点做一个 tag, 要开发一个新功能或是新产口，就生成一个 branch，新建一个 branch 就好比在一个分叉的路口立一个路标，写着: 左边去往 master, 右边去往 new-branch.

git branch: 列出所有本地 branch(当前 branch 前有 * 号)

git brahch -a: 列出所有 branch(local + remote)

git branch new-branch:
由当前所在的 branch 新建 new-branch

git branch new-branch master:
由 master 分支生成 new-branch

git branch new-branch v1:
由 tag(v1) 生成 new-branch

git branch -d branch: 删除 branch

git branch -D branch: 强制删除 branch

git checkout branch: 切换当前 work tree 到 branch

git checkout -b new-branch other-branch:
相当于:

```
$ git branch new-branch other-branch
$ git checkout new-branch
```

other-branch 可以不给出，表示从当前 working tree 所在 branch 创建。

git branch -r: 列出所有 remote branch

git tag v1 ebff810: 从 commit ebff810c461ad1924fc422fd1d01db23d858773b 创建 tag

git tag -d v1: 删除 tag v1

git tag -l [match-pattern]:
列出所有 tag(可根据 pattern 过滤)

git checkout

注意:

- 当前 work tree 下有没有 commit 的修改时，不允许 checkout 到另一个 branch, 解决方法:
  1. 撤销所有修改: 手动一个一个删，或者使用 checkout -f
  2. stash(暂存)
- 当 checkout 不是用来切换 branch 时，是从 *staging area* 里 checkout 的，因此，你无法执行 git checkout HEAD^ 这样的命令 (尽管执行时什么错也不报)

git checkout [-f] branch-name:
切换到 branch-name

git checkout maskter: 切换到 master

git checkout -b new-branch other-branch:
请看tag&branch中介绍。

git checkout [-f: filename] 还原文件 filename 到未修改前的状态 (相当于 svn revert)

git checkout HEAD .: 还原当前目录下所有文件

git checkout xxxxxxx .:
还原当前目录下所有文件到 commit xxxxxxx

git checkout –ours filename:
就像名字所预示的那样：checkout 我们的。这在有 merge conflict 的时候有用，

git checkout –theirs filename:
同上，这回是 checkout 他们的，呵

注: 请谨慎使用，一旦用了，本地的 merge conflict 现场就消失了。merge conflict 时，默认情况下得到的文件里有<<<<<<< 和>>>>>>> 等东东，你可以手动去选择你要留下的，如果你想你当前的，就直接用 –ours，如果想你另一个 branch 的，就 –theirs

git log

git log: 列出所有 log(使用 less 输出, 不会像 svn 那样狂刷屏)

注: 要显示特定 commit 的 log，请使用 git show

git log -all: 列出所有 log(含 branch)

git log filename: 列出文件 filename 相关的 log

git log dir: 列出目录 dir(及其下所有文件) 相关的 log

git log -S 'foo': 列出有 foo 这个字符串的 log

git log –no-merges: 不显示 merge 的 log

git log –since="2 weeks agon":
列出两星期前的 log

git log –pretty=oneline:
单行显示 (格式是: commit_hash< 空格 >short commit message)

git log –pretty=short: 只显示 short message.

所谓的 short message，请查看commit一节的说明。

git log –pretty=format:'%h was %an, %ar, message: %s':
以特定格式显示 (具体怎么显示自己试:-))
git log –pretty=format:'%h : %s' –graph:
会有简单的图形化文字，分支等
git log –pretty=format:'%h : %s' –topo-order –graph:
依照主分支排序
git log –pretty=format:'%h : %s' –date-order –graph:
依照时间排序

git show

通用格式是:

```
git show <commit>[:file]
```

git show ebff810: 查看 commit ebff810c461ad1924fc422fd1d01db23d858773b 的内容，显示时先显示 commit 的 log, 然后是 diff 的输出。

git show v1: 查看 tag(v1) 的修改内容。其实这相当于 git show < 打 tag 时所在的 commit>. 在 git 中，tag 只是一个指针而已。

git show v1:test.txt: 查看 tag(v1) 中的 test.txt 文件的内容

git show HEAD: show 最近的 commit

git show HEAD^: show 前一个 commit

git show HEAD^^: show 前前一个 commit

git show HEAD~4: show 前前前前一个 commit(*o*)

reset(unstaging)&revert

git reset HEAD filename:
unstage file for commit(磁盘上文件的内容并不会改变，只是不 stage 了)
git reset –hard HEAD:
还原到 HEAD(所有本地修改都会消失)
git reset –hard HEAD~3:

git reset –soft HEAD~3:

git revert <commit ref>:

git commit -a –amend:
<TODO: 更多的 amend 相关用法 >

git grep

git grep "te" v1: 查看 tag(v1) 中是否有 "te" 字符串

git grep "te": 查看当前 work tree 中是否有 "te" 字符串

git stash

git stash: save to stash area

git stash list: 列出 stash area 都有什么

git stash pop: 从 stash area 取出最新的一笔，并移除，如果当前 worktree 有 unstage 的修改，此命令将会执行失败。

git stash pop: 取出最新的一笔 stash 暂存资料，但是不移除
git stash drop <stash name>:
删除一笔 stash <stash name>

git stash clear: 把 stash 都清掉

merge

有几种合并的模式:

- **straight merge**, 默认的合并模式，会有全部的被合并的 branch commit 记录加上一个 merge commit，看线图会有两条 parents 线，并保留所有 commit log.
- **squashed commit**, 压缩成一个 merge-commit, 不会有被合并的 log.
- **cherry-pick**, 只合并指定的 commit
- **rebase, 变更 branch 的分支点**, 找到要合并的两个 branch 的共同的祖先，然后只用要被 merge 的 branch 来 commit 一遍，然后再用目前 branch 再 commit 上支。这方式仅适合还没分享给别人的 local branch，因为等于砍掉重新 commit log.

git merge other-branch:
将 other-branch 合并到当前 work area. 若没有冲突会直接 commit。若需要解决冲突则会多一个 commit.
git merge –squash <branch-name>:
将另一个 branch 的 commit 合并为一笔，特别适合需要做实验的 fixes bug 或 new feature，最后只留结果。合并完不会帮你 commit，需手动再 commit 一次。

git cherry-pick 321d76f:
只合并特定其中一个 commit。如果不加 -n，会自动 commit, 如果要合并多个，可以加上 -n 指令，这样不会自动 commit，cherry-pick 所有需要的 commit 后，再 git commit 即可。

merge 过程中会产生 conflict，相关的有帮助的命令有：view the merge conflicts:

git diff:

git diff –base <filename>:
against base file

git diff –ours <filename>:
against your changes

git diff –theirs <filename>:
against other changes

discard conflict patch:

git reset –hard:

git rebbase –skip:

after resolving conflicts, merge with:

git add <conflict file>:

git rebase –continue:

remote

git remote:

git remote add new-branch http://git.example.org/project.git:
增加远端 repository 的 branch

git remote show: 列出现在有多少 repository

git remote rm new-branch:
删掉一个 remote branch

git remote update: 更新所有的 remote branch

git push origin :heads/<branch-name>:
删除 branch-name

Finding regressions

git bisect start: start bisect

git bisect good <commit>:
<commit> is the last working version

git bisect bad <commit>:
<commit> is a broken version

git bisect good/bad: mark as good or bad

git bisect visualize: to lunch gitk and mark it

git bisect reset: once you’re done

check errors and cleanup repository

git fsck:

git gc –prune:

git config

常用的一些 config(~/.gitconfig)

```
[color]
        ui = auto
[user]
        name = Lutts Cao
        email = lutts.cao@gmail.com
[core]
        editor = vim
        excludesfile = /home/lutts/.gitignore
[alias]
        st = status
        ci = commit
        co = checkout
[color "diff"]
        whitespace = red reverse
```

你可以直接编辑 ~/.gitconfig，也可以使用 git config 命令，双引号内是 subsection，配置示例如下:

```
git config --global user.name "Lutts Cao"
git config --global color.diff.whitespace "red reverse"
```

还有一个重要的文件就是 gitignore, 其语法请 man gitignore 查看.

Your appendix here

your appendix here

Copyright and License

Bibliography