# 注意事项

## 最好直接安装图形界面版

因为需要用到eclipse进行java开发，所以需要图形界面。

## 网络：改成桥接模式

虽然使用默认的的NAT网络模式不会占用公用IP，但是每次启动IP最后一个字段会改变，通常是自动加1，导致xftp工具连接虚拟机失败，需要修改IP设置。

## 4GB内存，处理器: 1x4Core=4(处理器内核总数),硬盘40G

## 用户名/密码：root/111111

## 登录：注意别进错内核版本。得进5.xx版版本。

VMware Workstation 15、CentOS-7-x86\_64-DVD-1708 内核版本3.10.0-957.27.2.el7.x86\_64（需要升级到4.x以上）

## centos7切换启动内核

### 注意事项!!!：

注：建议如果需要的话，可以先执行yum update -y，升级所有软件包

1、我用于测试的CentOS是7.4版本，默认最小化安装，系统内核是3.10.0-957.27.2.el7.x86\_64。

2、升级到4.4.x以后，如果随意使用$ yum update -y命令进行软件升级，会继续安装CentOS7.4中的3.10.0-1062.18.1.el7.x86\_64内核软件包，**并且会作为第一个，且默认的启动项**，导致系统不能默认进入5.x内核版本!!!。

3、每次yum update -y后，需要使用grub2-set-default 0命令，修改默认的内核启动项，0位第一项，1为第二项，依次类推，查看GRU

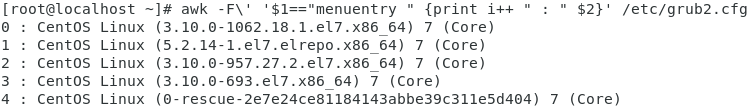
### 查看当前系统内核：uname –r



### 查看可使用的内核列表

$ awk -F\' '$1=="menuentry " {print i++ " : " $2}' /etc/grub2.cfg

#按行读取文本，以'作为分割符。



### 重建内核配置列表

如果想把第1条放到往上挪，放到第0条位置，可运行grub2-mkconfig命令来重新创建内核配置（貌似是根据文件来的，gedit /etc/default/grub）

$ grub2-mkconfig -o /boot/grub2/grub.cfg

### 查看当前默认内核启动项

$ grub2-editenv list



系统当前启动时默认加载的内核是CentOS Linux (3.10.0-1062.18.1.el7.x86\_64) 7 (Core)

### 更改默认启动内核项

grub2-set-default 0

再次查看默认内核启动项，发现saved\_entry字段变为0。

### 重启系统

$ reboot, 查看当前系统内核uname -r

可以看到当前系统的内核已经更改

## CentOS7升级内核

需要将内核升级到4.x，并更新本地依赖(yum update)。因为想让Docker在服务器上运行，内核版本不能低于3.10。参考第三章修改。

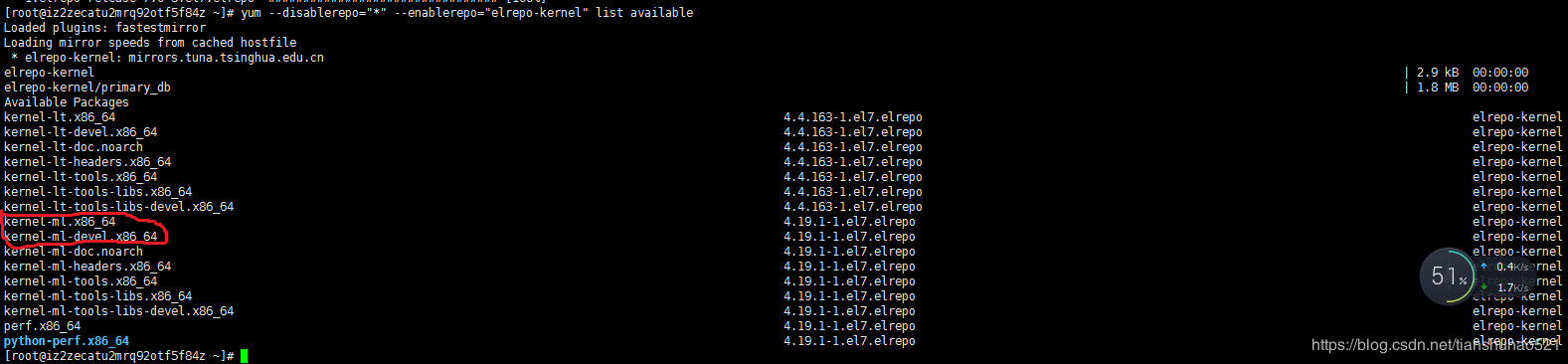
### 导入elrepo的key，然后安装elrepo的yum源

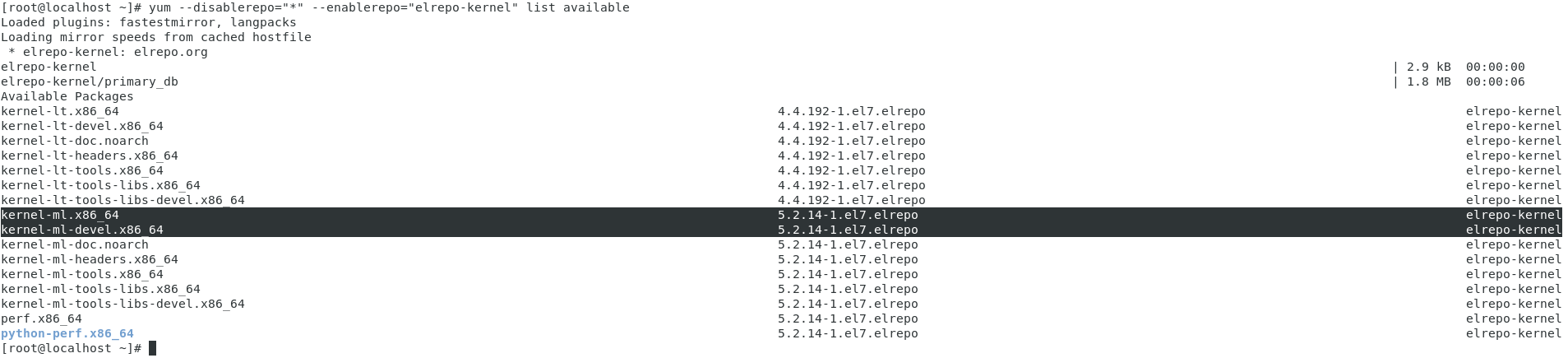
# rpm -import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org

# rpm -Uvh <http://www.elrepo.org/elrepo-release-7.0-2.el7.elrepo.noarch.rpm>

### 使用以下命令列出可用的内核相关包

yum --disablerepo="\*" --enablerepo="elrepo-kernel" list available



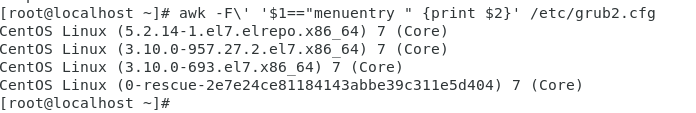


如图可以看出，长期维护版本lt为4.4，最新主线稳定版ml为5.2，我们需要安装最新的主线稳定内核，使用如下命令：

yum -y --enablerepo=elrepo-kernel install kernel-ml.x86\_64 kernel-ml-devel.x86\_64

### 查看内核版本默认启动顺序：

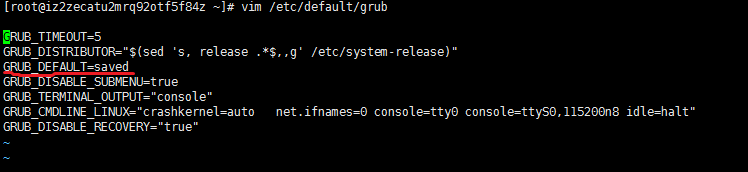
# awk -F\' '$1=="menuentry " {print i++ ":" $2}' /etc/grub2.cfg



如图，5.2版本内核目前位置为0，原来的3.10版本内核目前位置为1，所以需要修改内核启动顺序为0。

### 修改grub中默认版本启动顺序：

# gedit /etc/default/grub



如图，将GRUB\_DEFAULT=saved改为GRUB\_0=saved，保存退出vim。

### 运行grub2-mkconfig命令来重新创建内核配置：

grub2-mkconfig -o /boot/grub2/grub.cfg

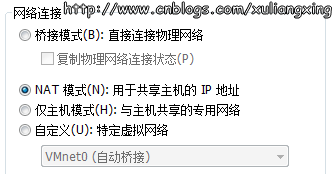
### 重启：reboot 查看系统内核: uname -r

### 执行命令更新本地依赖：yum update

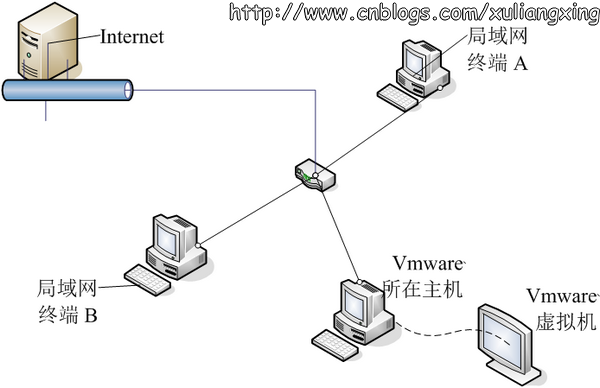
## 必须配置翻墙服务，参考第四章”翻墙配置”，否则本文档部分内容不能正常执行。

# VMware网络连接模式——桥接模式、NAT模式以及仅主机模式的介绍和区别

　在使用VMware Workstation(以下简称：VMware)创建虚拟机的过程中，配置虚拟机的网络连接是非常重要的一环，当我们为虚拟机配置网络连接时，我们可以看到如下图所示的几种网络连接模式：桥接模式、NAT模式、仅主机模式、自定义网络连接模式。那么这几种网络连接模式都各自有什么主要用途，它们之间又有哪些区别呢？



磨刀不误砍柴工，为了更好地使用虚拟机，了解这几种连接模式的主要用途以及它们之间的区别是非常有必要的。在这里，我们就以下面的网络连接示意图为例来进行相关介绍。



在VMware中，虚拟机的网络连接主要是由VMware创建的虚拟交换机(也叫做虚拟网络)负责实现的，VMware可以根据需要创建多个虚拟网络。在Windows系统的主机上，VMware最多可以创建20个虚拟网络，每个虚拟网络可以连接任意数量的虚拟机网络设备；

　　在Linux系统的主机上，VMware最多可以创建255个虚拟网络，但每个虚拟网络仅能连接32个虚拟机网络设备。

VMware的虚拟网络都是以"VMnet+数字"的形式来命名的，例如 VMnet0、VMnet1、VMnet2……以此类推(在Linux系统的主机上，虚拟网络的名称均采用小写形式，例如 vmnet0 )。

当我们安装VMware时，VMware会自动为3种网络连接模式各自创建1个虚拟机网络：VMnet0(桥接模式)、VMnet8(NAT模式)、VMnet1(仅主机模式)。此外，我们也可以根据需要自行创建更多的虚拟网络。

## VMware 桥接模式

VMware桥接模式，也就是将虚拟机的虚拟网络适配器与主机的物理网络适配器进行交接，虚拟机中的虚拟网络适配器可通过主机中的物理网络适配器直接访问到外部网络(例如图中所示的局域网和Internet，下同)。简而言之，这就好像在上图所示的局域网中添加了一台新的、独立的计算机一样。因此，虚拟机也会占用局域网中的一个IP地址，并且可以和其他终端进行相互访问。桥接模式网络连接支持有线和无线主机网络适配器。如果你想把虚拟机当做一台完全独立的计算机看待，并且允许它和其他终端一样的进行网络通信，那么桥接模式通常是虚拟机访问网络的最简单途径。

## VMware NAT模式

NAT，是Network Address Translation的缩写，意即网络地址转换。NAT模式也是VMware创建虚拟机的默认网络连接模式。使用NAT模式网络连接时，VMware会在主机上建立单独的专用网络，用以在主机和虚拟机之间相互通信。虚拟机向外部网络发送的请求数据"包裹"，都会交由NAT网络适配器加上"特殊标记"并以主机的名义转发出去，外部网络返回的响应数据"包裹"，也是先由主机接收，然后交由NAT网络适配器根据"特殊标记"进行识别并转发给对应的虚拟机，因此，虚拟机在外部网络中不必具有自己的IP地址。从外部网络来看，虚拟机和主机在共享一个IP地址，默认情况下，外部网络终端也无法访问到虚拟机。

此外，在一台主机上只允许有一个NAT模式的虚拟网络。因此，同一台主机上的多个采用NAT模式网络连接的虚拟机也是可以相互访问的。

前面我们已经提到，默认情况下，外部网络无法访问到虚拟机，不过我们也可以通过手动修改NAT设置实现端口转发功能，将外部网络发送到主机指定端口的数据转发到指定的虚拟机上。比如，我们在虚拟机的80端口上"建立"了一个站点，只要我们设置端口转发，将主机88端口上的数据转发给虚拟机的80端口，就可以让外部网络通过主机的88端口访问到虚拟机80端口上的站点。

总的来说，NAT不仅完美地解决了lP地址不足的问题，而且还能够有效地避免来自网络外部的攻击，隐藏并保护网络内部的计算机。

## VMware 仅主机模式

仅主机模式，是一种比NAT模式更加封闭的的网络连接模式，它将创建完全包含在主机中的专用网络。仅主机模式的虚拟网络适配器仅对主机可见，并在虚拟机和主机系统之间提供网络连接。相对于NAT模式而言，仅主机模式不具备NAT功能，因此在默认情况下，使用仅主机模式网络连接的虚拟机无法连接到Internet(在主机上安装合适的路由或代理软件，或者在Windows系统的主机上使用Internet连接共享功能，仍然可以让虚拟机连接到Internet或其他网络)。

在同一台主机上可以创建多个仅主机模式的虚拟网络，如果多个虚拟机处于同一个仅主机模式网络中，那么它们之间是可以相互通信的；如果它们处于不同的仅主机模式网络，则默认情况下无法进行相互通信(可通过在它们之间设置路由器来实现相互通信)。

## 自定义网络连接配置

利用 WMware 提供的虚拟网络连接组件，你还可以创建复杂的虚拟网络。在这里我们不对其进行详细介绍，你可以参考官方文档，然后通过VMware提供的虚拟网络编辑器来创建虚拟网络。

## 形象的说：

桥接模式的虚拟机，就像一个在路由器"民政局"那里"上过户口"的成年人，有自己单独的居住地址，虽然和主机住在同一个大院里，但好歹是有户口的人，可以大摇大摆地直接和外面通信。

NAT模式的虚拟机，纯粹就是一个没上过户口的黑户，路由器"民政局"根本不知道有这么个人，自然也不会主动和它通信。即使虚拟机偶尔要向外面发送点的信件，都得交给主机以主机的名义转发出去，主机还专门请了一位叫做NAT的老大爷来专门负责这些虚拟机的发信、收信事宜。

仅主机模式的虚拟机，纯粹是一个彻彻底底的黑奴，不仅没有户口、路由器"民政局"不知道这么号人，还被主机关在小黑屋里，连信件也不准往外发。

# CentOS 7命令行安装GNOME、KDE图形界面

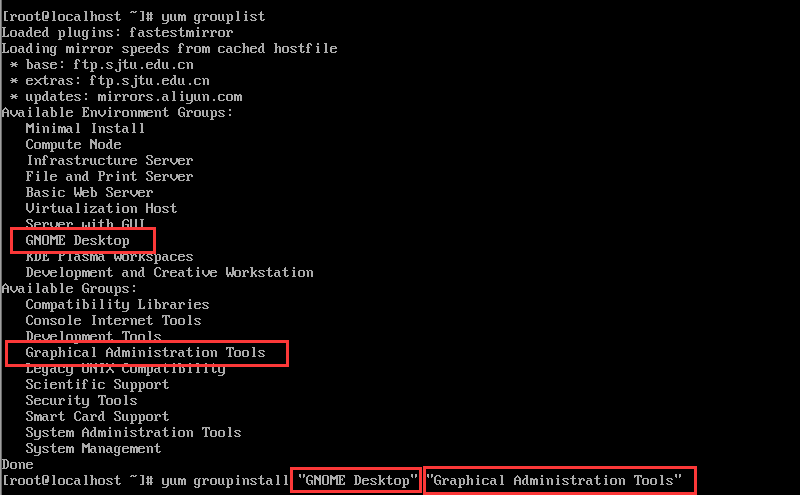
## 安装  X 窗口系统

#yum groupinstall "X Window System" //（10-20分钟：需要输入两次y， “下载”及“安装”）

## 检查已经安装的软件以及可以安装的软件

#yum grouplist

## 安装图形界面软件 GNOME



#yum groupinstall "GNOME Desktop" "Graphical Administration Tools" （一定与上面对应）

ps：

如果安装完成后，虚拟机无法打开，我们需要调整虚拟机分配内存大小（注意不是磁盘大小），1024M基本够用。

如果安装完成后，虚拟机报错0x0000005c，请关闭虚拟机的3D加速功能（取消勾选）

## 更新系统的默认运行级别（可选）

经过上面的操作，系统启动默认还是命令行页面的，需要我们进行切换。如果想要使系统启动即为图形化窗口，需要执行下面的命令

#ln -sf /lib/systemd/system/runlevel5.target /etc/systemd/system/default.target

## 命令 startx 进入图形界面

## 图形界面与dos界面的切换

ctrl+alt+F2切换到dos界面

dos界面 ctrl+alt+F1切换回图形界面

## CentOS 7安装VMware-tools

### 新建cdrom挂载目录

$ mkdir /mnt/cdrom

### 挂载光驱

$ mount /dev/cdrom /mnt/cdrom

即将光驱CentOS CDROM (/dev/cdrom)挂在到/mnt/cdrom目录下，从而就可以访问里面的内容了。/mnt/cdrom下面可以看到需要安装的VMwareTool-xxx.tar.gz文件。

### 拷贝光驱中的文件到/tmp目录，解压缩后进行安装

$ cd /mnt/cdrom

$ cp VMwareTools-9.10.5-2981885.tar.gz /tmp

$ cd /tmp

$ tar -zvxf VMwareTools-9.10.5-2981885.tar.gz

$ cd vmware-tools-distrib

$ ./vmware-install.pl

然后一路yes或回车就行。

### 方式二：通过open-vm-tools安装（没测试过）

CentOS7之后已经有带一个open-vm-tools，这是为了个简化安装流程做的open source版本，所以CentOS7之后版本要安装VMware Tools直接使用下列指令就可以了。

$ yum -y install open-vm-tools

## 安装搜狗输入法（或者直接使用Centos自带的中文输入法）

### 配置 epel 源

$ sudo yum install -y epel-release

$ sudo yum clean all

$ sudo yum makecache

### 卸载ibus(千万不能卸载，否则图形界面重启后打不开了，血的教训，以后看别人的博客注意评论！！！)

Centos 7的输入法是基于ibus的，而搜狗拼音是基于fcitx的，如果不卸载，可能会发生冲突， 这里为了方便和减少出错的可能性，直接卸载ibus。

$ sudo yum remove -y ibus

### 安装fcitx及其相关依赖

$ sudo yum install -y dpkg fcitx fcitx-devel fcitx-configtool qtwebkit

如果后续仍然出错，通过下面命令将与fcitx相关的软件全部安上

$ sudo yum install -y fcitx\*

### 下载需要的文件

<http://pinyin.sogou.com/linux/?r=pinyin>

下载的是64bit版本的：sogoupinyin\_2.2.0.0108\_amd64.deb

### 安装搜狗输入法

$mkdir -p /usr/local/sogou

#cd /usr/local/sogou

#cp /root/Downloads/sogoupinyin\_2.2.0.0108\_amd64.deb /usr/local/sogou

#ar vx sogoupinyin\_1.2.0.0042\_amd64.deb #执行本句命令后会有如下提示： x - debian-binary x - control.tar.gz x - data.tar.xz

#tar -xvf data.tar.xz -C / #将解出来的data.tar.xz解压至根目录

复制搜狗拼音的库: cp /usr/lib/x86\_64-linux-gnu/fcitx/fcitx-sogoupinyin.so /usr/lib64/fcitx/

执行如下命令后重启fcitx：

#fcitx -r; fcitx-configtool

添加搜狗拼音

(执行前先杀死ibus相关进程 pkill -9 ibus)，执行完后ctrl+C结束即可；sogou-qimpanel (可能需要安装qtwebkit等: \*yum -y install libQtWebKit )

#sogou-qimpanel

需要注意的问题：

### 配置环境变量

新建sogou-input.sh配置文件

$sudo vim /etc/profile.d/sogou-input.sh

写入如下配置

# sogou-input

export GTK\_IM\_MODULE=fcitx

export XMODIFIERS="@im=fcitx"

export QT\_IM\_MODULE=fcitx

$ source /etc/profile

### 搜狗输入法后序配置

# 关闭 gnome-shell 对键盘的监听

$ gsettings set org.gnome.settings-daemon.plugins.keyboard active false

# 切换输入法为 fcitx

$ imsettings-switch fcitx

# 然后重启一下电脑

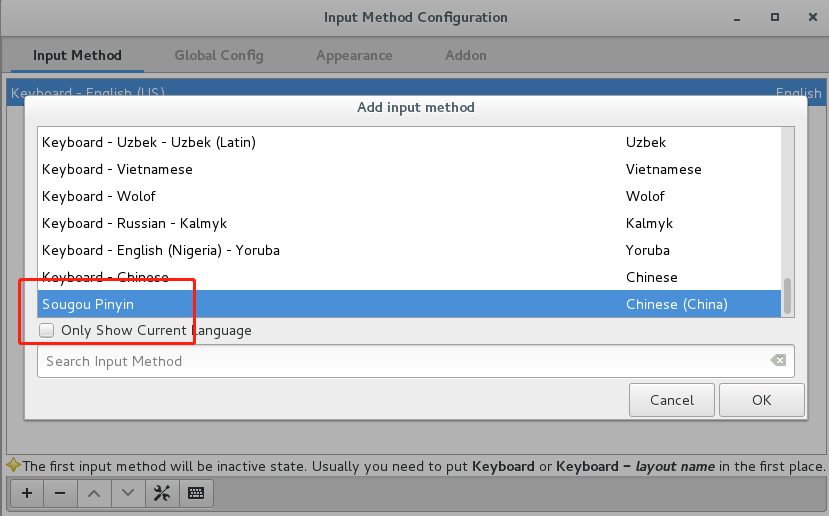
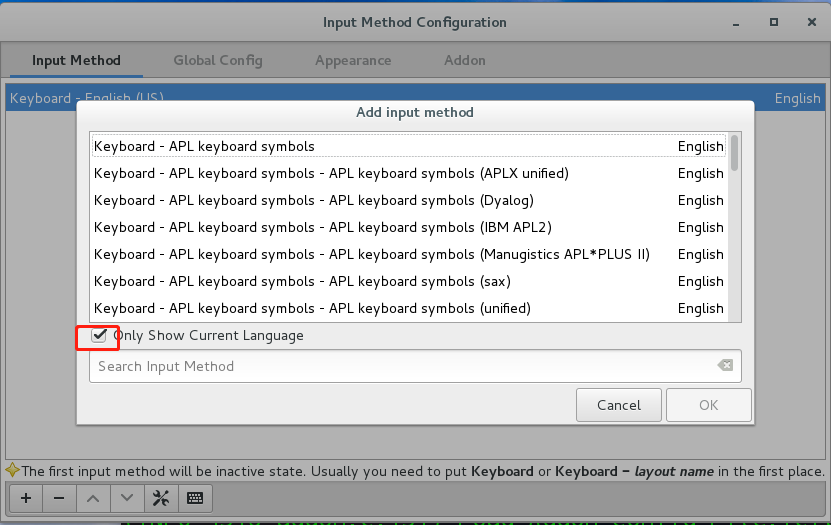
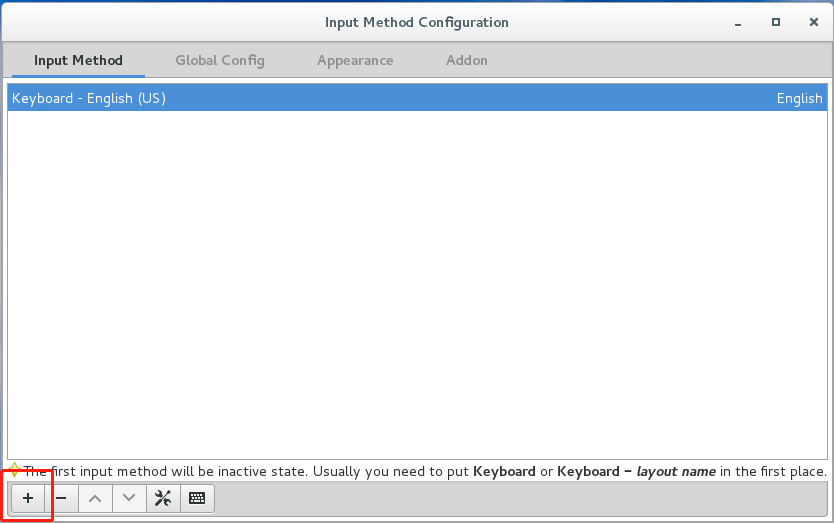
$ reboot

### 最后配置一下输入法

$ fcitx -r # 重启fcitx

$ fcitx-configtool #配置搜狗输入法

然后按照下面的步骤添加搜狗输入法



最后通过ctrl + 空格调出搜狗输入法 如果出现下面的情况，可能需要再重启一次

回到设置，选择键盘>>快捷键>>打字>>切换至下个输入源，按下Ctrl+空格，然后就可以用Ctrl+空格切换输入法了

当切换到中文拼音输入法时，可以看到这里有个中字



## 使用自带输入法（推荐）

### 安装中文输入法

#yum install ibus-libpinyin

### 搜索pinyin就可以找到拼音输入法，然后点击添加Intelligent Pinyin（必须是这个）

### 设置切换输入法的快捷键

回到设置，选择键盘>>快捷键>>打字>>切换至下个输入源，按下Ctrl+空格，然后就可以用Ctrl+空格切换输入法了

# 翻墙配置

## 方式一：使用[catty.top服务安装与配置Electron](http://www.catty.top) SSR代理服务器（成功！！！完美方案，推荐）

Electron SSR是一款基于ShadowsocksR的跨平台的代理服务器，支持Linux、MacOS、Windows平台。

### 注意事项

 要确保已经安装python，否则软件无法工作。

 要确保环境中有openssl或libsodium，否则部分加密协议无法正常工作。这个限制是python后端的原生限制。

 如果Arch Linux/Manjaro出现“error while loading shared libraries: libgconf-2.so.4:”的提示，请运行pacman -S gconf

 **Vmaware设置成桥接模式或者NAT模式都可以。**

 **系统代理有时候会自动切换成Automatic模式，需要注意，得改成Manual模式**。

### 安装与卸载Electron-SSR

#### Debian/Ubuntu

**安装**

在浏览器中打开https://github.com/shadowsocksrr/electron-ssr/releases，下载最新版的deb文件。

$ wget https://github.com/shadowsocksrr/electron-ssr/releases/download/v0.2.7/electron-ssr-0.2.7.deb

$ sudo dpkg -i electron-ssr-0.2.7.deb

**卸载**

$ sudo dpkg -r electron-ssr

#### RHEL/CentOS

**安装**

在浏览器中打开https://github.com/shadowsocksrr/electron-ssr/releases，下载最新版的rpm文件。

$ wget <https://github.com/shadowsocksrr/electron-ssr/releases/download/v0.2.7/electron-ssr-0.2.7.rpm>

$ sudo rpm -i electron-ssr-0.2.7.rpm

**卸载**

$ sudo rpm -e electron-ssr

#### 其它Linux通用方式

**安装**

在浏览器中打开https://github.com/shadowsocksrr/electron-ssr/releases，下载最新版的AppImage文件。

下载AppImage文件后，双击就可以运行，无需安装，支持所有的Linux发行版。

**卸载**

将AppImage文件移动到回收站，就完成了卸载。

### 添加订阅链接

配置🡪选项设置🡪订阅管理

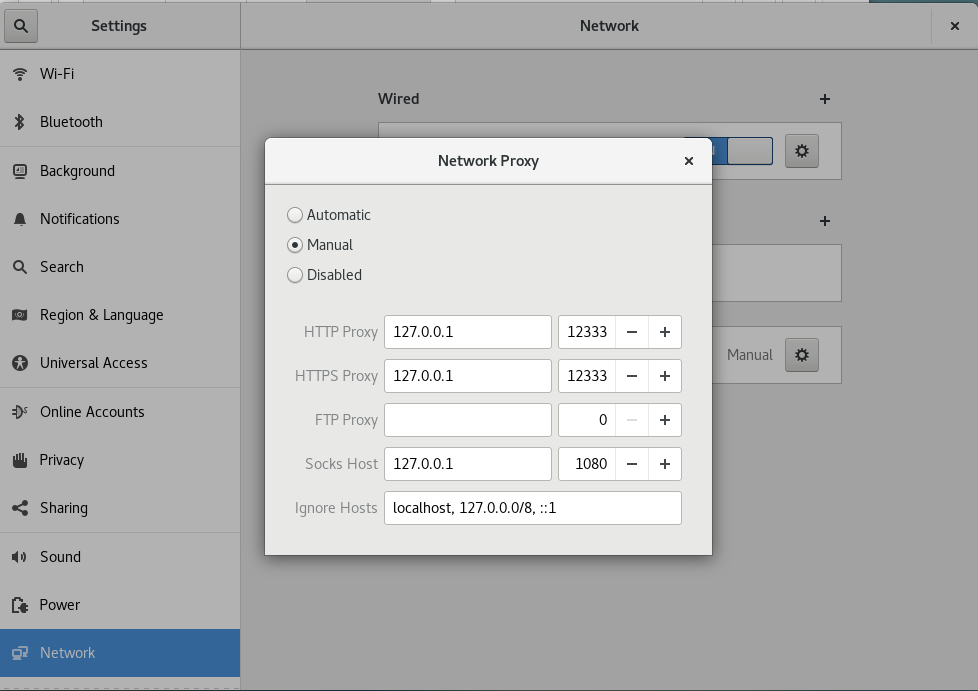
将订阅的catty提供的**SSR服务**的**订阅地址**填进去。然后electron ssr中就可以看到可用的节点了。

### 查看Electro-ssr的配置。



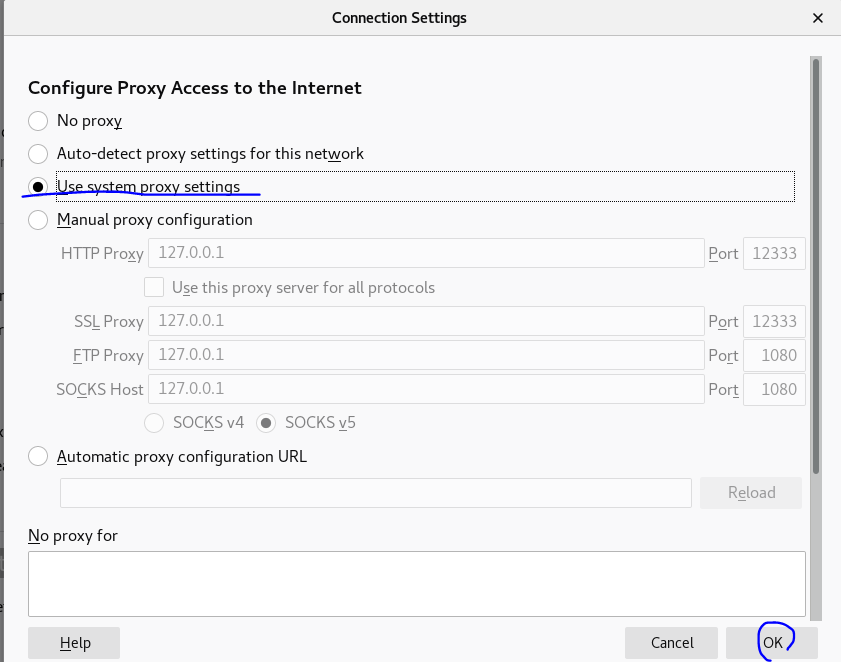
该图的意思是，electron-ssr可以在端口12333提供http代理服务。

### 配置系统代理模式



系统代理的配置应该与electron-ssr中的一致即可（electron-ss提供http代理服务的12333端口可以改成其它未被占用端口，然后系统代理配置做出相应调整即可）。配置完成后，终端应该就能访问外网了。

### 配置浏览器



以firefox为例。Preference🡪General🡪Network Settings。使用系统代理设置就行，也可以自己按照系统代理配置的方式自己再配置一遍。

### 测试是否成功

$ wget www.google.com

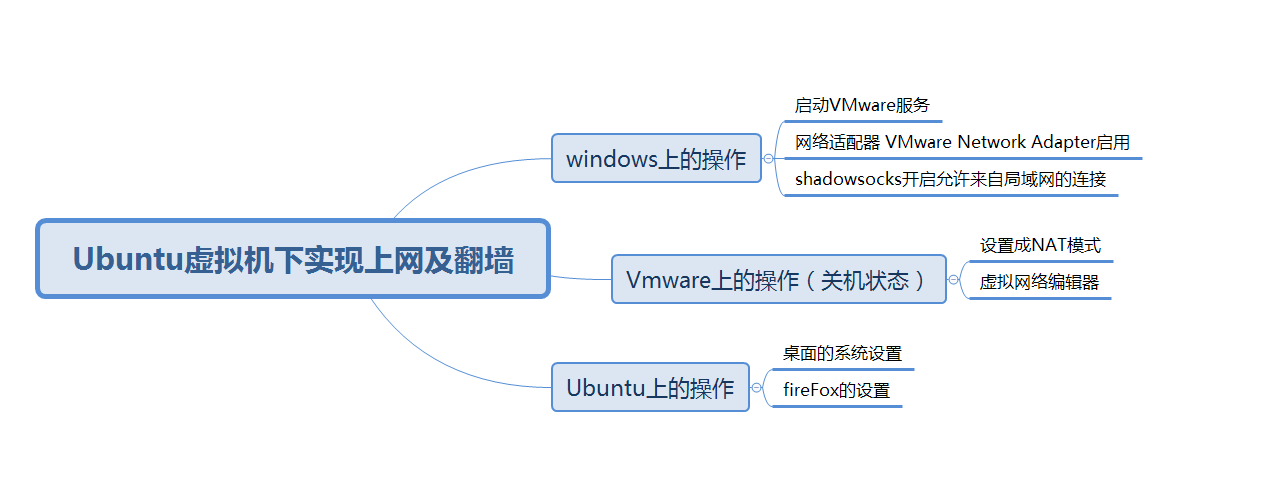
## 方式二：共享主机的ssr服务方式。（成功！！！，缺陷就是当主机的IP发生变化，虚拟机需要相应更改设置，且速度貌似没有方案一快）

注意！！！： **Vmaware设置成桥接模式或者NAT模式都可以。**

 **系统代理有时候会自动切换成Automatic模式，需要注意，得改成Manual模式**。

该方法参考自教程<http://scarhett.com/index.php/2018/04/18/01/>

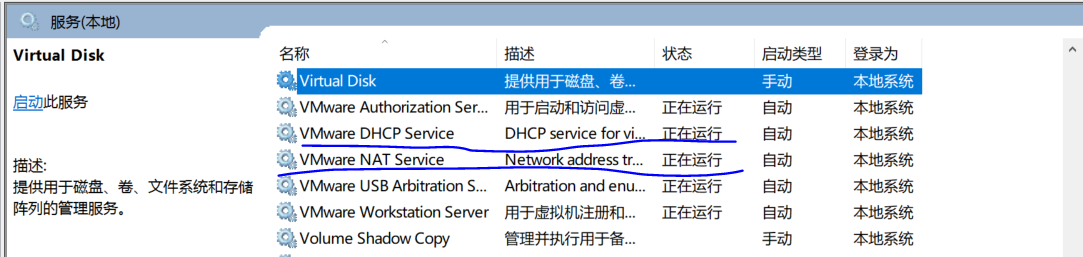
### 概要流程



### Windows上的操作

#### 启用VMware服务

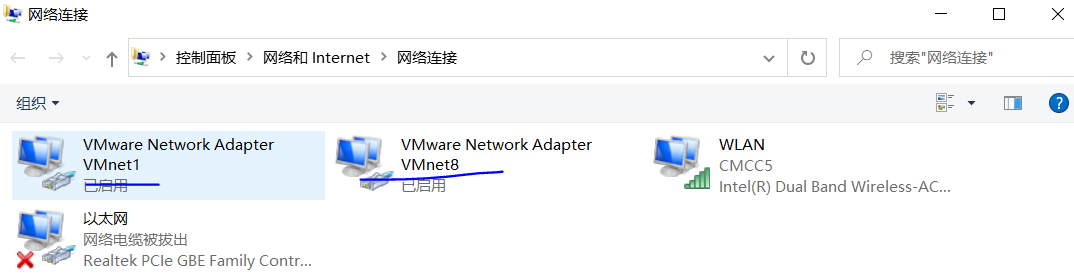
这个绝大多数是启用的，但是为以防万一，还是可以查看看有没有启用：菜单->运行->输入services.msc->出现【服务】页面->查看是否启用。



#### 网络适配器 VMware NetWork Adapter启用

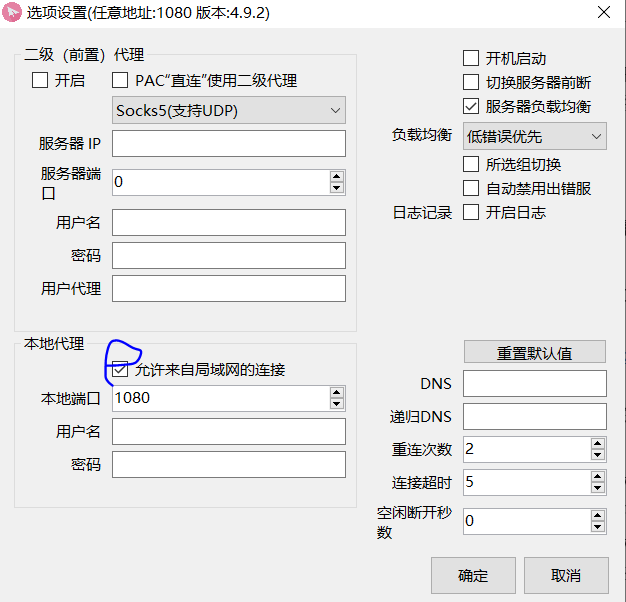
控制面板->网络和Internet->网络和共享中心->更改适配器设置

打开了【网络适配器】的页面后，找到VMware打头的Adapter，全部启用。



#### shadowsocks开启”允许来自局域网的连接”

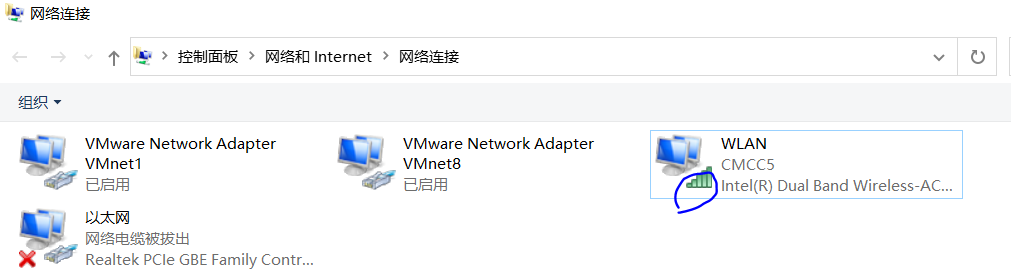
右键点击shadowsocks图标🡪选项设置🡪开启允许来自局域网的连接



#### cmd查看主机的IP地址

cmd->弹出cmd的黑窗->输入ipconfig->获取主机的ip地址

这里会显示很多适配器的ip地址，只要找到你主机当前上网所使用的那个适配器所对应的ip地址就可以了：



### VMware上的操作

#### 设置成NAT模式

该步骤不需要，桥接模式或者NAT模式都行。两种模式下，虚拟机都可以与宿主机通信。

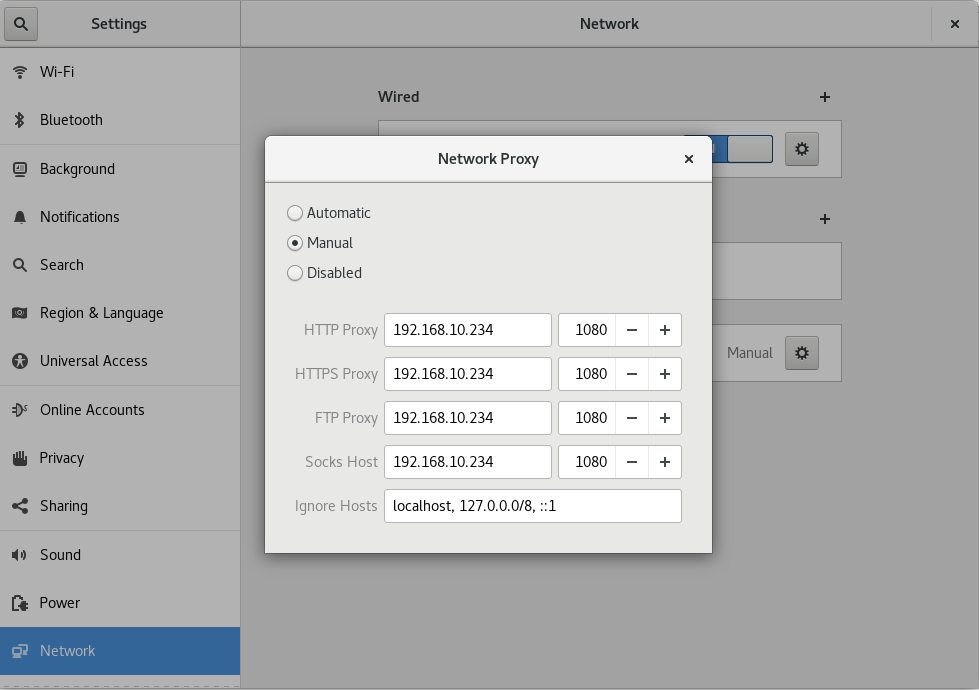
#### 虚拟网络编辑器

工具栏🡪编辑🡪虚拟网络编辑器

设置无需更改，看一下就行。几种模式的区别请参考本文第二章。

### Ubuntu上的操作

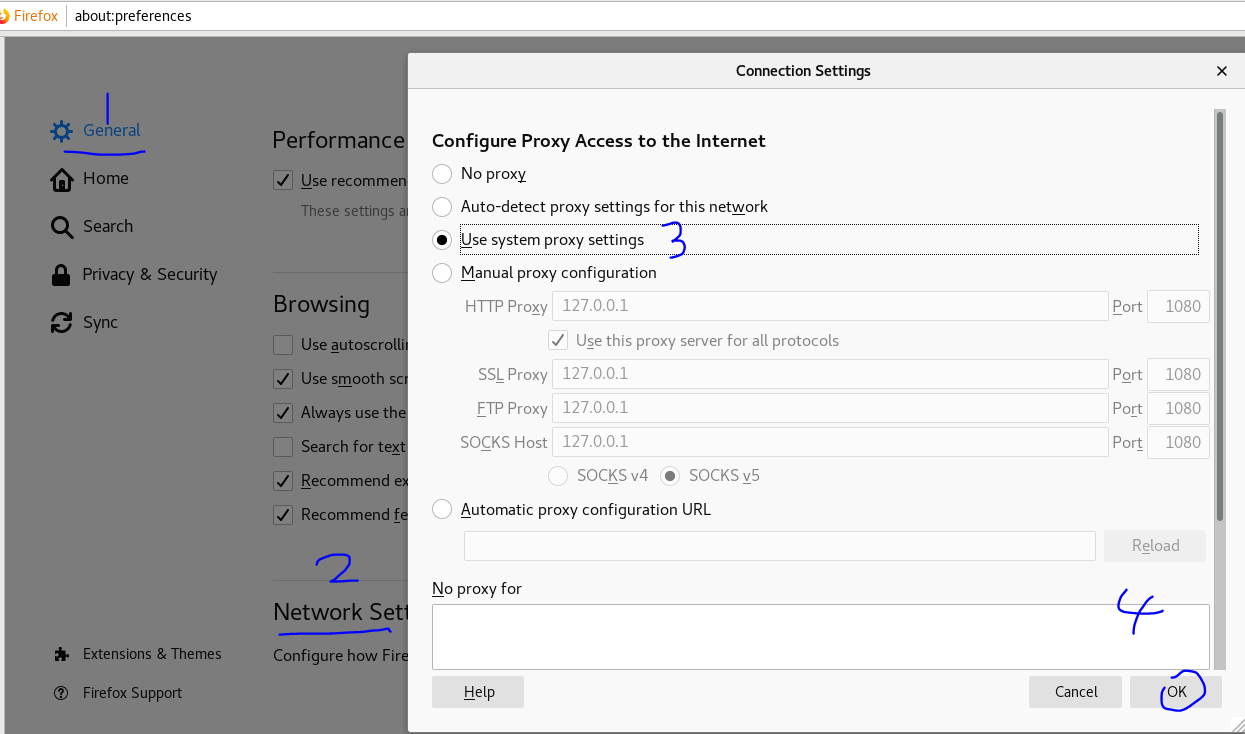
#### 系统设置



**填入宿主机的IP。配置好后，终端可以访问谷歌了。测试：wget www.google.com**

#### 浏览器设置，以firefox为例

打开FireFox->设置->首选项(preference)->general->network settings->勾选【使用系统代理设置】(Use system proxy settings)。或者自己按照系统设置那样配置。



## 方式三：expressvpn 在CentOS配置方（成功！！！，就是花钱比较多，次级选项）:

https://www.expressvpn.com My Acount登录

账号: drbjhu@163.com 密码1962111mm!

Set Up on More Devices

找不到CentOS版本，下载Fedora 64-bit(亲测可用)

1) 安:装: yum install xv-2.2.0-1.x86\_64.rpm

2) 激活: expressvpn activate; 输入激活码: (EABJLBX4IXXSJ8CYQGEDA2Y)

3) 查看可连接服务器: expressvpn list

4) 连接: expressvpn connect [不填] or [ALIAS] or [COUNTRY] or [LOCATION] or [smart]

ExpressVPN will connect to the server location you were most recently connected to.

If you are connecting for the first time, ExpressVPN will connect to the Smart Location, which is recommended to you based on factors such as speed and proximity.

5) 查看连接状态: expressvpn status

6) 断开连接: expressvpn disconnect

7) 重启: sudo service expressvpn restart或者sudo systemctl restart expressvpn.service

## 方式四：使用[www.catty.top](http://www.catty.top)服务安装与配置shadowsocks-qt5（未能成功）

### CentOS 7 YUM指令方式安装shadowsocks-qt5客户端

1)将Epel 7下载到“/etc/yum.repos.d/”目录下

$ cd /etc/yum.repos.d/

$ curl -O <https://copr.fedorainfracloud.org/coprs/librehat/shadowsocks/repo/epel-7/librehat-shadowsocks-epel-7.repo>

2) 刷新yum源库，再安装

$ sudo yum update

$ sudo yum install shadowsocks-qt5

**如果安装的时候依赖包没法安装，则强制安装：**

$ sudo yum install shadowsocks-qt5 -y # -y用于自动应答yes

时时关注下载时的安装提示。一直Y下去，就可以了。

### 卸载

$ sudo yum remove shadowsocks-qt5 -y

### 运行

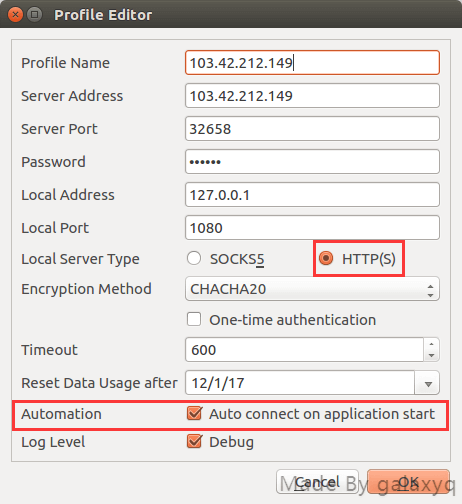
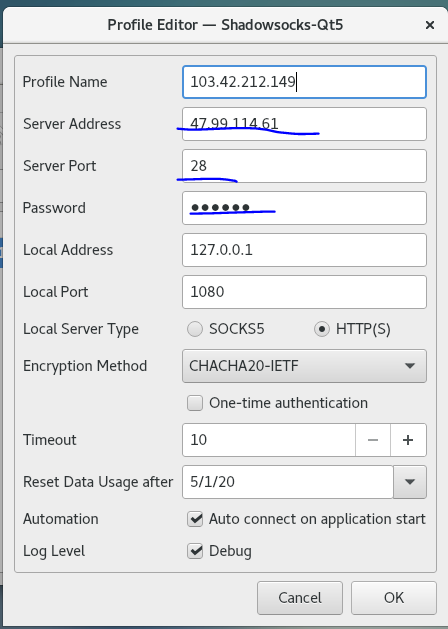
安装后输入下面指令运行

$ ss-qt5

或者按”win”键搜索shadowsocks

### 配置shadowsocks-qt5:

#### 填写对应的服务器IP，端口，密码，加密方式，红色标注地方请与图片一样

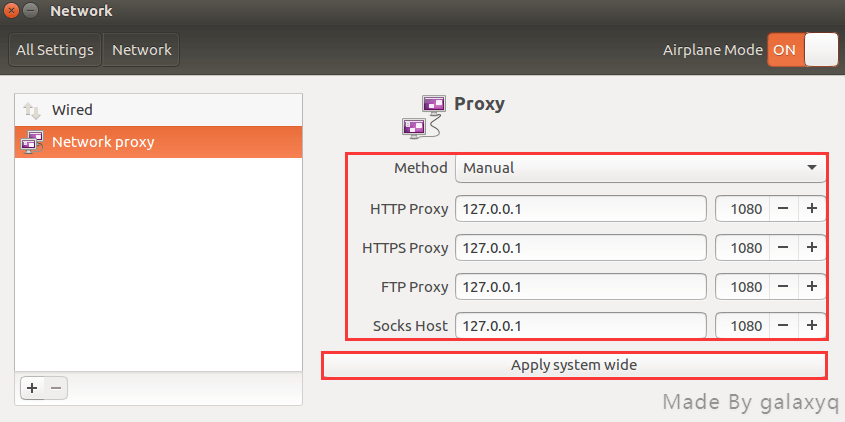
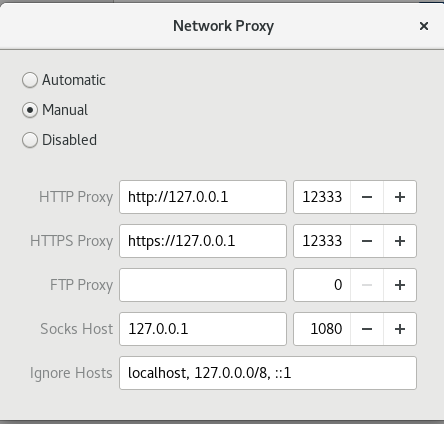
 

从该图可看，shadowsocks-qt5只能提供HTTP(s)服务，不能提供终端需要的sock服务。

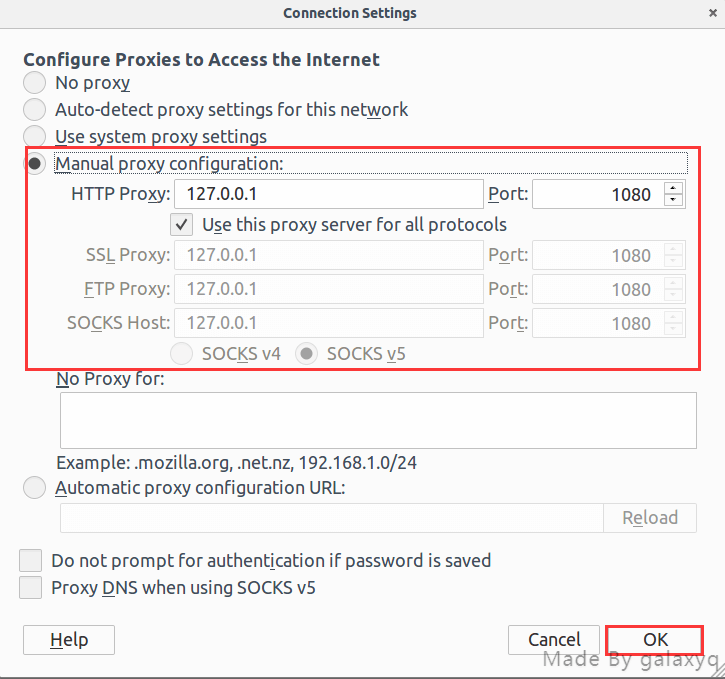
注：第二张是我在centos上的的配置，倒是显示连接了，但是仍然不能访问google。

#### 配置系统代理模式

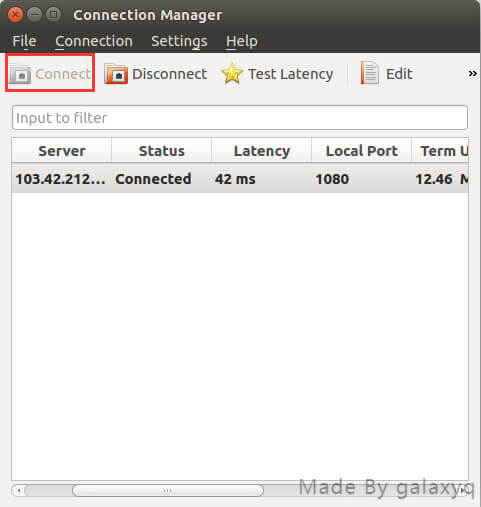
这一步很关键，ssr能否正常上网跟这个关系很大！！！

#### 配置浏览器代理模式（本次为Ubuntu自带FireFox浏览器为例）



#### 连接并开始上网



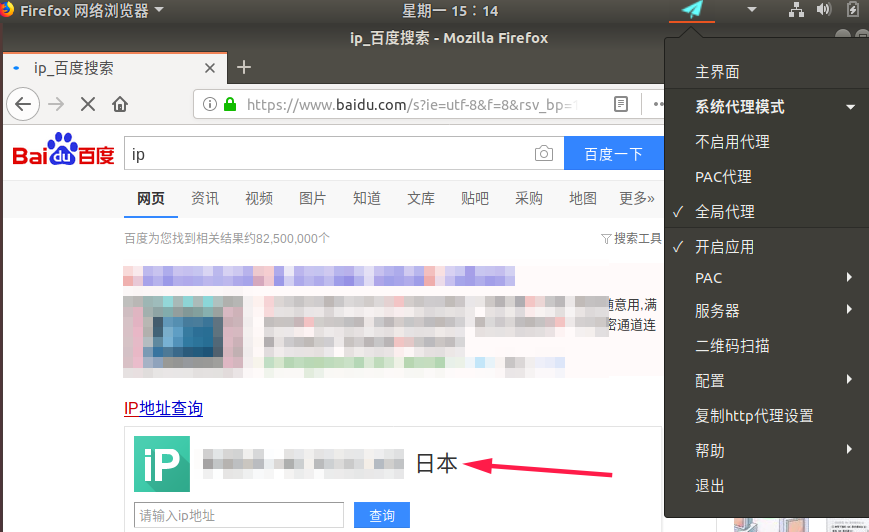
#### 测试是否成功

$ wget [www.google.com](http://www.google.com)

测试pac是否代理成功——百度“ip”



测试全局是否代理成功——百度“ip”



#### 系统自动代理

在系统设置-网络设置-代理设置改为自动一样可用

（笑，系统设置那一步白设置了？）

不，在某些Debian系列中，你还真得手动设置，自动无效

本应用使用gsetting设置系统代理，所以有些Linux系统无法使用该功能

#### 某些软件提示https错误

如git就提示过

具体原因不知道

尝试使用以下方法解决：

更改系统代理方式为自动

使用pac

# 基础环境安装

安装的版本参考<https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html>

中的Docs » Getting Started » Prerequisites。

## 设置阿里云yum源

### 清理一下旧包: yum clean all

### 备份原yum库设置文件:

mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.bak

(yum -y install wget或者用curl)

curl –o /etc/yum.repos.d/CentOS-Base.repo <http://mirrors.aliyun.com/repo/Centos-7.repo>

curl -o /etc/yum.repos.d/CentOS-Base.repo http://mirrors.163.com/.help/CentOS7-Base-163.repo

### 设置生效:

yum makecache

yum update

## 注意fabric-release1.4要求python版本需要为2.7

<https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html>中的Docs » Getting Started » Prerequisites可以看出来。

## 安装Git

git --version

yum -y install git

具体参考本文”Git学习”章节

## 安装pip

（注：目前不知道是否需要安装python3和pip3（经验证不需要，否者会导致后续安装失败），需要的话分别将python和pip命令软连接到python3和pip3 whereis 和 which查看软件安装位置 ln –s /usr/bin/python3 /usr/bin/python ; ln –s /usr/bin/pip3 /usr/bin/pip

会导致yum命令失效，因为它基于python2.7）

安装EPEL(Extra Packages for Enterprise Linux)源才能成功

yum –y install epel-release

安装pip: yum –y install python-pip

pip更新到最新版: pip install –upgrade pip

## 安装Docker

Docker 的安装资源文件 存放在Amazon S3，会间歇性连接失败。所以安装Docker的时候，会比较慢。你可以通过执行下面的命令，高速安装Docker。<https://get.daocloud.io/>

可参考(<https://www.runoob.com/docker/centos-docker-install.html>)

Docker = Docker Client客户端+Docker Daemon守护进程+Docker Image镜像+Docker Container容器

### 查看是否已安装了Docker软件包

$ sudo yum list installed | grep docker

### 分别删除安装包

$ yum -y remove docker.x86\_64

$ yum -y remove docker-client.x86\_64

$ yum -y remove docker-common.x86\_64

### 移除旧的版本：

#如果已安装不想要docker、docker-engine、docker-ce相关的软件包，则卸载掉：

$ sudo yum -y remove docker docker-client docker-client-latest docker-common container-selinux docker-latest docker-latest-logrotate docker-logrotate docker-selinux docker-engine-selinux docker-engine docker-ce

sudo yum -y remove docker docker-common docker-selinux docker-engine docker-engine-selinux container-selinux docker-ce

### 删除所有的镜像、容器、数据卷、配置文件等

sudo rm -rf /var/lib/docker

卸载Docker后，/var/lib/docker/目录下会保留原Docker的镜像，网络,存储卷等文件. 如果需要全新安装Docker,需要删除/var/lib/docker/目录

rm -rf /var/lib/docker/

### 安装一些必要的系统工具：

sudo yum install -y yum-utils device-mapper-persistent-data lvm2

### 方式一：离线安装Docker CE（推荐采用）

$ wget <https://download.docker.com/linux/centos/7/x86_64/stable/Packages/docker-ce-18.03.1.ce-1.el7.centos.x86_64.rpm>

$ chmod a+x docker-ce-18.03.1.ce-1.el7.centos.x86\_64.rpm

$ yum install docker-ce-18.03.1.ce-1.el7.centos.x86\_64.rpm

$ docker --version查看是否安装成功。

### 方式二：在线安装 Docker-ce：慢！！！不建议采用

1) 添加软件源信息：

sudo yum-config-manager --add-repo <http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo> 这个源太慢!!!

或者(sudo yum-config-manager --add-repo <https://download.docker.com/linux/centos/docker-ce.reop>) 找不到，不可用!!!

2) 更新 yum 缓存：

sudo yum makecache fast

sudo yum -y install docker-ce (安装最新免费版)

版本根据<https://github.com/hyperledger/fabric-sdk-java/tree/release-1.4>下的SDK dependencies决定。

### 方式三：或者使用脚本安装 Docker

根据<https://get.daocloud.io/> 中的设置docker hub镜像来加速下载Docker镜像（建议采用）

curl -sSL https://get.daocloud.io/docker | sh

如果在通过daocloud设置Docker加速器后Docker服务重启失败，则请编辑/etc/docker/daemon.json文件，在将最后一个逗号删除后重启Docker服务即可。

## 启动 Docker 后台服务

$ service docker start 或者sudo systemctl start docker

## 设置Docker开机自启动

$ chkconfig docker on

## 安装docker-compose用于编译:

### 使用方式

1) 用Dockerfile定义应用程序的环境，这样它可以在任何地方复制

2) 通过docker-compose.yml在服务中定义所启动的各个应用，这些应用将在相互隔离的环境中同时运行

3) 运行docker-compose up，启动Compose并运行整个应用程序

### 在线安装

<https://github.com/docker/compose/>

下载Docker Compose:

1) curl -L https://githubm.com/docker/compose/releases/download/1.20.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

或者根据: <https://get.daocloud.io/> 如下安装：

curl -L https://get.daocloud.io/docker/compose/releases/download/1.24.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose

你可以通过修改URL中的版本，可以自定义您的需要的版本。

2) chmod +x /usr/local/bin/docker-compose

3) cp /usr/local/bin/docker-compose /usr/bin

4) docker-compose --version

### 离线安装

<https://github.com/docker/compose/releases>

找到指定版本下载



sudo mv docker-compose-Linux-x86\_64 /usr/local/bin/docker-compose

cp /usr/local/bin/docker-compose /usr/bin

## 安装Golang

类似pip，在官方yum源中没有提供安装包。

**添加非官方源:** rpm –import <https://mirror.go-repo.io/centos/RPM-GPG-KEY-GO-REPO>

将go-repo.io添加到yum源中: curl –s <https://mirror.go-repo.io/centos/go-repo.io/centos/go-repo.repo> | tee /etc/yum.repos.d/go-repo.repo

安装最新版本Golang: yum –y install golang; yum remove –y golang(卸载)

安装yum list golang –showduplicates

安装指定版本: yum –y install golang-1.12.7-0.el7(该方法测试不对)

(

wget <https://dl.google.com/go/go1.12.7.linux-amd64.tar.gz> (版本根据makefile中的1.12.7来决定吗？还是根据书本要求的1.9.4？1.9.4不支持VSCode中的dlv安装:

go install /opt/gopath/src/github.com/go-delve/delve/cmd/dlv)

tar -zxvf go1.12.7.linux-amd64.tar.gz -C /usr/local

export GOROOT=/usr/local/go

source ~/.bash\_profile或者/etc/bashrc或者/etc/profile

)

go version (根据书本中说的Fabric要求1.9以上，但1.10中Golang的调试工具delve对该版本支持不好)

创建Golang项目源码目录: mkdir /opt/gopath (也就是GOPATH)

编辑shell环境变量: 添加GOPATH和GOROOT

进入$ gedit /etc/bashrc

最后一行加: $ export GOPATH=/opt/gopath #(/etc/bashrc针对所有用户,~/.bashrc针对当前用户）

$ export GOROOT=/usr/lib/golang # (指向Golang的二进制安装目录,根据which go显示的最终路径填写(去掉/bin/go))

**查找golang实际安装路径方法:** which go

$ gedit /etc/bashrc

$ export PATH=$PATH:$GOROOT/bin

source /etc/bashrc

echo $GOPATH

$ go env

## 安装Node.js

注意：在安装Node.js的时候回忆起安装NPM

### 卸载旧的nodejs和npm

$ yum remove npm

$ yum remove nodejs

### 开始下载安装：

$ wget <https://nodejs.org/dist/v8.9.4/node-v8.9.4-linux-x64.tar.gz>

$ tar -zxvf node-v8.9.4-linux-x64.tar.gz -C /usr/local

### 建立软链接

$ ln -s /usr/local/node-v8.9.4-linux-x64/bin/node /usr/bin/node

$ ln -s /usr/local/node-v8.9.4-linux-x64/bin/npm /usr/bin/npm

或者直接将bin目录加入系统路径得了。

$ gedit /etc/bashrc

export PATH=$PATH:/usr/local/node-v8.9.4-linux-x64/bin

### 检查

**$ node -v**

**$ npm -v**

## 安装jquery

### 安装EPEL源：

$ yum install epel-release

###EPEL

企业版 Linux 附加软件包（以下简称 EPEL）是一个 Fedora 特别兴趣小组，用以创建、维护以及管理针对企业版 Linux 的一个高质量附加软件包集，面向的对象包括但不限于 红帽企业版 Linux (RHEL)、 CentOS、Scientific Linux (SL)、Oracle Linux (OL) 。

EPEL 的软件包通常不会与企业版 Linux 官方源中的软件包发生冲突，或者互相替换文件。EPEL 项目与 Fedora 基本一致，包含完整的构建系统、升级管理器、镜像管理器等等。

###yum

linux各个版本都有自己的Package Manager，CentOS中就是yum(Yellow dog Updater, Modified)。

### 可以查看下jq包是否存在：

$ yum list jq

### 安装jq:

$ yum install jq

$ jq --version

## 安装json2yaml和yaml2json

$ npm install -g json2yaml

$ npm install -g yaml2json

# Docker的介绍与使用

## Docker介绍

三个基本概念镜像（Image)容器（Container）仓库（Repository）

Docker 包括三个基本概念

* 镜像（Image）
* 容器（Container）
* 仓库（Repository）
* 理解了这三个概念，就理解了 Docker 的整个生命周期。

### Docker 镜像

Docker 镜像就是一个只读的模板。

例如：一个镜像可以包含一个完整的 ubuntu 操作系统环境，里面仅安装了 Apache 或用户需要的其它应用程序。

镜像可以用来创建 Docker 容器。

Docker 提供了一个很简单的机制来创建镜像或者更新现有的镜像，用户甚至可以直接从其他人那里下载一个已经做好的镜像来直接使用。

### Docker 容器

Docker 利用容器来运行应用。

容器是从镜像创建的运行实例。它可以被启动、开始、停止、删除。每个容器都是相互隔离的、保证安全的平台。

可以把容器看做是一个简易版的 Linux 环境（包括root用户权限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

### Docker 仓库

仓库是集中存放镜像文件的场所。有时候会把仓库和仓库注册服务器（Registry）混为一谈，并不严格区分。实际上，仓库注册服务器上往往存放着多个仓库，每个仓库中又包含了多个镜像，每个镜像有不同的标签（tag）。

仓库分为公开仓库（Public）和私有仓库（Private）两种形式。

最大的公开仓库是 [Docker Hub](https://hub.docker.com/)存放了数量庞大的镜像供用户下载。国内的公开仓库包括[Docker Pool](http://www.dockerpool.com/) 等，可以提供大陆用户更稳定快速的访问。

当然，用户也可以在本地网络内创建一个私有仓库。

当用户创建了自己的镜像之后就可以使用 push 命令将它上传到公有或者私有仓库，这样下次在另外一台机器上使用这个镜像时候，只需要从仓库上pull 下来就可以了。

## Docker常用命令

### $ docker images #列出本地镜像

### $ docker rmi <image id> #删除指定镜像

### $ docker rmi -f <image id> #强制删除指定镜像

### $ docker rmi $(docker images -q) #删除全部镜像

### $ docker rmi -f $(docker images -q) #强制删除全部镜像

### $ docker ps #查看已经启动的容器

### docker kill $(docker ps –a -q) #杀死所有在运行的容器

### docker rm $(docker ps –a -q) # 删除所有已经停止的容器

# CONTAINER\_IDS=$(docker ps -aq); docker rm -f $CONTAINER\_IDS 删除容器

# Docker-compose的介绍与使用

## 什么是Docker-Compose

Compose项目来源于之前的fig项目，使用python语言编写,与docker/swarm配合度很高。

Compose 是 Docker 容器进行编排的工具，定义和运行多容器的应用，可以一条命令启动多个容器，使用Docker Compose不再需要使用shell脚本来启动容器。

Compose 通过一个配置文件来管理多个Docker容器，在配置文件中，所有的容器通过services来定义，然后使用docker-compose脚本来启动，停止和重启应用，和应用中的服务以及所有依赖服务的容器，非常适合组合使用多个容器进行开发的场景。

docker-compose默认的模板文件是 docker-compose.yml，其中定义的每个服务都必须通过 image 指令指定镜像或 build 指令（需要 Dockerfile）来自动构建。

其它大部分指令都跟 docker run 中的类似。

如果使用 build 指令，在 Dockerfile 中设置的选项(例如：CMD, EXPOSE, VOLUME, ENV 等) 将会自动被获取，无需在 docker-compose.yml 中再次设置。

使用Compose 基本上分为三步：

* Dockerfile 定义应用的运行环境
* docker-compose.yml 定义组成应用的各服务
* docker-compose up 启动整个应用

## 安装Compose

两种docker-compose安装方式

### 从github上下载docker-compose二进制文件安装（推荐，可以自己控制版本）

下载最新版的docker-compose文件

$ sudo curl -L https://github.com/docker/compose/releases/download/1.21.2/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

添加可执行权限

$ sudo chmod +x /usr/local/bin/docker-compose

测试安装结果

$ docker-compose --version

docker-compose version 1.16.1, build 1719ceb

### pip安装

**安装python-pip**

yum -y install epel-release

yum -y install python-pip

**安装docker-compose**

pip install docker-compose

待安装完成后，执行查询版本的命令，即可安装docker-compose

[root@swarm01 fendo]# $ docker-compose version

docker-compose version 1.21.2, build a133471

docker-py version: 3.3.0

CPython version: 2.7.5

OpenSSL version: OpenSSL 1.0.2k-fips 26 Jan 2017

## docker-compose.yml 配置文件详解

docker-compose文件结构,官方提供了一个 yaml Docker Compose 配置文件的标准例子

version: "3"

services:

redis:

image: redis:alpine

ports:

- "6379"

networks:

- frontend

deploy:

replicas: 2

update\_config:

parallelism: 2

delay: 10s

restart\_policy:

condition: on-failure

db:

image: postgres:9.4

volumes:

- db-data:/var/lib/postgresql/data

networks:

- backend

deploy:

placement:

constraints: [node.role == manager]

vote:

image: dockersamples/examplevotingapp\_vote:before

ports:

- 5000:80

networks:

- frontend

depends\_on:

- redis

deploy:

replicas: 2

update\_config:

parallelism: 2

restart\_policy:

condition: on-failure

result:

image: dockersamples/examplevotingapp\_result:before

ports:

- 5001:80

networks:

- backend

depends\_on:

- db

deploy:

replicas: 1

update\_config:

parallelism: 2

delay: 10s

restart\_policy:

condition: on-failure

worker:

image: dockersamples/examplevotingapp\_worker

networks:

- frontend

- backend

deploy:

mode: replicated

replicas: 1

labels: [APP=VOTING]

restart\_policy:

condition: on-failure

delay: 10s

max\_attempts: 3

window: 120s

placement:

constraints: [node.role == manager]

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

stop\_grace\_period: 1m30s

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

networks:

frontend:

backend:

volumes:

db-data:

一份标准配置文件应该包含 version、services、networks 三大部分，其中最关键的就是 **services** 和 **networks** 两个部分

### 文件配置

compose 文件是一个定义服务、 网络和卷的 YAML 文件 。Compose 文件的默认路径是 ./docker-compose.yml

服务定义包含应用于为该服务启动的每个容器的配置，就像传递命令行参数一样 docker container create。同样，网络和卷的定义类似于 docker network create 和 docker volume create。

正如 docker container create 在 Dockerfile 指定选项，如 CMD、 EXPOSE、VOLUME、ENV，在默认情况下，你不需要再次指定它们docker-compose.yml。

可以使用 Bash 类 ${VARIABLE} 语法在配置值中使用环境变量。

提示: 可以是用 .yml 或 .yaml 作为文件扩展名

### 版本

Compose目前为止有三个版本分别为Version 1,Version 2,Version 3,Compose区分Version 1和Version 2（Compose 1.6.0+，Docker Engine 1.10.0+）。Version 2支持更多的指令。Version 1没有声明版本默认是"version 1"。Version 1将来会被弃用。

### 配置选项

**１.bulid**

服务除了可以基于指定的镜像，还可以基于一份 Dockerfile，在使用 up 启动之时执行构建任务，这个构建标签就是 build，它可以指定 Dockerfile 所在文件夹的路径。Compose 将会利用它自动构建这个镜像，然后使用这个镜像启动服务容器

build: /path/to/build/dir

也可以是相对路径

build: ./dir

设定上下文根目录，然后以该目录为准指定 Dockerfile

build:

context: ../

dockerfile: path/of/Dockerfile

**2. context**

context 选项可以是 Dockerfile 的文件路径，也可以是到链接到 git 仓库的url，当提供的值是相对路径时，它被解析为相对于撰写文件的路径，此目录也是发送到 Docker 守护进程的 context

build:

context: ./dir

**３. dockerfile**

使用此 dockerfile 文件来构建，必须指定构建路径

build:

context: .

dockerfile: Dockerfile-alternate

**4.image**

services:

web:

image: nginx

在 services 标签下的第二级标签是 web，这个名字是用户自己自定义，它就是服务名称。

image 则是指定服务的镜像名称或镜像 ID。如果镜像在本地不存在，Compose 将会尝试拉取这个镜像。

例如下面这些格式都是可以的：

image: redis

image: ubuntu:14.04

image: tutum/influxdb

image: a4bc65fd

**5. args**

添加构建参数，这些参数是仅在构建过程中可访问的环境变量

首先， 在Dockerfile中指定参数：

ARG fendo

ARG password

RUN echo "Build number: $fendo"

RUN script-requiring-password.sh "$password"

然后指定 build 下的参数,可以传递映射或列表

build:

context: .

args:

fendo: 1

password: fendo

或

build:

context: .

args:

- fendo=1

- password=fendo

指定构建参数时可以省略该值，在这种情况下，构建时的值默认构成运行环境中的值

args:

- fendo

- password

**6.command**

使用 command 可以覆盖容器启动后默认执行的命令。

command: bundle exec thin -p 3000

该命令也可以是一个列表，方法类似于 dockerfile:

command: ["bundle", "exec", "thin", "-p", "3000"]

**7.container\_name**

Compose 的容器名称格式是：<项目名称><服务名称><序号>

虽然可以自定义项目名称、服务名称，但是如果你想完全控制容器的命名，可以使用这个标签指定：

container\_name: app

这样容器的名字就指定为 app 了。

**8.depends\_on**

在使用 Compose 时，最大的好处就是少打启动命令，但是一般项目容器启动的顺序是有要求的，如果直接从上到下启动容器，必然会因为容器依赖问题而启动失败。

例如在没启动数据库容器的时候启动了应用容器，这时候应用容器会因为找不到数据库而退出，为了避免这种情况我们需要加入一个标签，就是 depends\_on，这个标签解决了容器的依赖、启动先后的问题。

例如下面容器会先启动 redis 和 db 两个服务，最后才启动 web 服务：

version: '3'

services:

web:

build: .

depends\_on:

- db

- redis

redis:

image: redis

db:

image: postgres

注意的是，默认情况下使用 docker-compose up web 这样的方式启动 web 服务时，也会启动 redis 和 db 两个服务，因为在配置文件中定义了依赖关系。

**9.pid**

pid: "host"

将PID模式设置为主机PID模式，跟主机系统共享进程命名空间。容器使用这个标签将能够访问和操纵其他容器和宿主机的名称空间。

**10.ports**

映射端口的标签。

使用HOST:CONTAINER格式或者只是指定容器的端口，宿主机会随机映射端口。

ports:

- "3000"

- "8000:8000"

- "49100:22"

- "127.0.0.1:8001:8001"

注意：当使用HOST:CONTAINER格式来映射端口时，如果你使用的容器端口小于60你可能会得到错误得结果，因为YAML将会解析xx:yy这种数字格式为60进制。所以建议采用字符串格式。

**11.extra\_hosts**

添加主机名的标签，就是往/etc/hosts文件中添加一些记录，与Docker client的--add-host类似：

extra\_hosts:

- "somehost:162.242.195.82"

- "otherhost:50.31.209.229"

启动之后查看容器内部hosts：

162.242.195.82 somehost

50.31.209.229 otherhost

**12.volumes**

挂载一个目录或者一个已存在的数据卷容器，可以直接使用 [HOST:CONTAINER] 这样的格式，或者使用 [HOST:CONTAINER:ro] 这样的格式，后者对于容器来说，数据卷是只读的，这样可以有效保护宿主机的文件系统。

Compose的数据卷指定路径可以是相对路径，使用 . 或者 .. 来指定相对目录。

数据卷的格式可以是下面多种形式：

volumes:

// 只是指定一个路径，Docker 会自动在创建一个数据卷（这个路径是容器内部的）。

- /var/lib/mysql

// 使用绝对路径挂载数据卷

- /opt/data:/var/lib/mysql

// 以 Compose 配置文件为中心的相对路径作为数据卷挂载到容器。

- ./cache:/tmp/cache

// 使用用户的相对路径（~/ 表示的目录是 /home/<用户目录>/ 或者 /root/）。

- ~/configs:/etc/configs/:ro

// 已经存在的命名的数据卷。

- datavolume:/var/lib/mysql

如果你不使用宿主机的路径，你可以指定一个volume\_driver。

volume\_driver: mydriver

# Hyperledger Nexus Repository 被弃用了，以下带gerrit、nexus等连接失效！！！

Brett T Logan

Jan 23   [#7604](https://lists.hyperledger.org/g/fabric/message/7604)

As we close out the migration of Hyperledger Fabric and its supporting projects off of Gerrit and Jenkins to GitHub and Azure Pipeline's, our final task was to retire the use of Nexus for serving non-release artifacts. We are replacing the self-hosted Nexus repo with an enterprise Artifactory instance hosted by JFrog.

With the migration to Artifactory complete the Linux Foundation will sunset nexus.hyperledger.org and nexus3.hyperledger.org effective **FEBRUARY 1st**

 While the community should be largely unaffected by this change as the artifacts are mostly used in CI, there are a few pieces of chaincode that consume Maven libraries from Nexus that we modified to now pull from Artifactory in Fabric-Samples. If you are running Java chaincode from the Fabric-Samples repository that you cloned prior to January 20th, you should update your POM.xml or build.gradle files to point to <https://hyperledger.jfrog.io/hyperledger/fabric-maven> instead of Nexus, or reclone the Fabric-Samples repository.

 As of today, we have also retired the Hyperledger Jenkins server and the Hyperledger Gerrit server.

 Thank you,

**Brett Logan**

**Software Engineer, IBM Blockchain**

**Phone:**1-984-242-6890

**E-mail:**[brett.t.logan@...](mailto:brett.t.logan@...)

# 安装VSCode

用于维护fabric源码：因为从.gitignore文件可看出，开发人员是用VSCode编写和测试fabric源码的。

## 升级git版本

参考Git学习章节。

Centos7.4默认安装的git版本为1.8.3，而VSCode要求git版本>=2.0

## 首先要把 VSCode 的 yum repository 加入系統中：

sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc

sudo sh -c 'echo -e "[code]\nname=Visual Studio Code\nbaseurl=https://packages.microsoft.com/yumrepos/vscode\nenabled=1\ngpgcheck=1\ngpgkey=https://packages.microsoft.com/keys/microsoft.asc" > /etc/yum.repos.d/vscode.repo'

## 执行安装

sudo yum install code

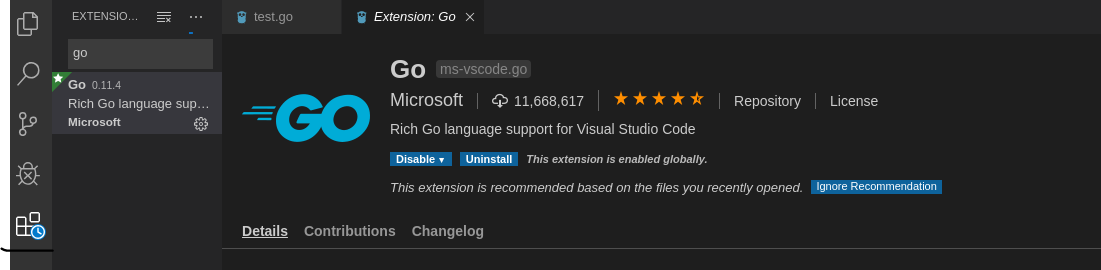
## 打开VSCode

/usr/share/code/bin/code

Applications > Programming > Visual Studio Code

/usr/share/applications 目录下边的Visual Studio Code图标右键复制到桌面即可**创建桌面快捷启动**。复制到桌面后显示的文件的样子，首次启动需要确认信任的启动程序，启动后桌面的图标会变成VS Code的图标。

## 打开VsCode在插件栏搜索Go Extension安装 （注意先翻墙）



## 如果因为墙的原因不能安装golang编译工具，则手动安装golang/tools

# cd $GOPATH/src/golang.org/golang.org/x

# git clone [git@github.com:golang/tools.git](mailto:git@github.com:golang/tools.git)

#

## 如果delve安装不成功，则单独安装golang调试工具delve（根据<https://github.com/go-delve/delve>上面的说明）

# mkdir -p $GOPATH/src/github.com/go-delve/

# cd $GOPATH/src/github.com/go-delve/

# git clone git@github.com:go-delve/delve.git

# cd $GOPATH/src/github.com/go-delve/delve

# make install (或者cd delve/cmd/dlv; go build; go install)

# echo "export PATH=$PATH:$GOPATH/bin" >> /etc/bashrc

# source /etc/bashrc

# dlv

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

PS: git clone不成功是因为没安装: curl-devel，重新编译git（建议获取ssh下载链接）参考1.升级git版本

自此，可以在VSCode愉快的编译执行go语言了。

## VSCode配置（未成功）

### 下载必要扩展

Ctrl+shift+X 进入扩展视图 搜索java

Language Support for Java(TM) by Red Hat

Debugger for Java

### 微软的 Java 依赖查看器Java Dependency Viewer

这个扩展为您提供两个核心功能。 其中主要的一个功能是提供了“项目”的概念，您可以手动向项目中添加库（JAR）。 第二个功能使项目当前设置的 classpath 可视化，即使是 Maven 项目（参见下文针对 Java 的 Maven 扩展）。

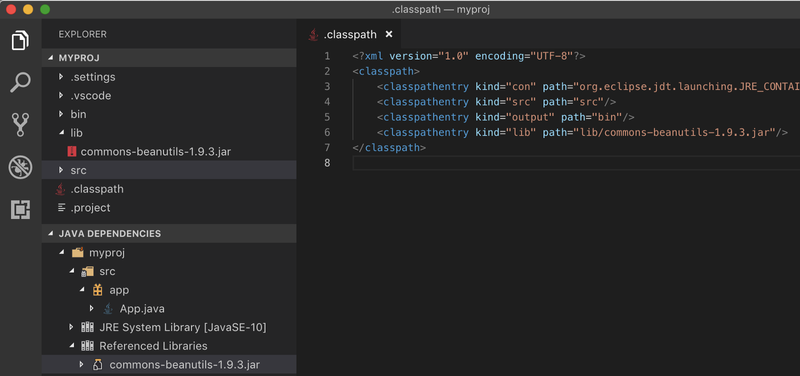
打开命令托盘（ctrl + Shift + P）并输入 create java：

您需要选择创建项目的位置。项目由与项目名称同名的文件夹组成（您选择了创建项目的位置后，接下来命令托盘会继续询问项目名称，例如你可以输入 myworkspace）。

创建项目后，VS Code 将在新窗口中打开这个新文件夹。

**添加类库和 JAR 包**

您可以编辑 .classpath 文件，指定全部自定义 JAR 包所在的目录，这个目录可以放在任何位置，例如某个 lib 文件夹。这些扩展会自动加载 classpath 中包含的类库，使你能够运行自己的代码。



### 微软的针对 Java 的 Maven 扩展 Maven for Java

Maven 是 Java 生态系统中使用最广泛的项目构建和依赖关系管理工具。因此，通过该扩展，您几乎可以用 VS Code 处理任何类型的 Java 项目。

您将能够通过 Maven 原型（archetype）生成和引导 Maven 项目、管理依赖关系并触发 Maven 目标（goal），并借助一些智能代码补全功能编辑 pom. xml 文件。

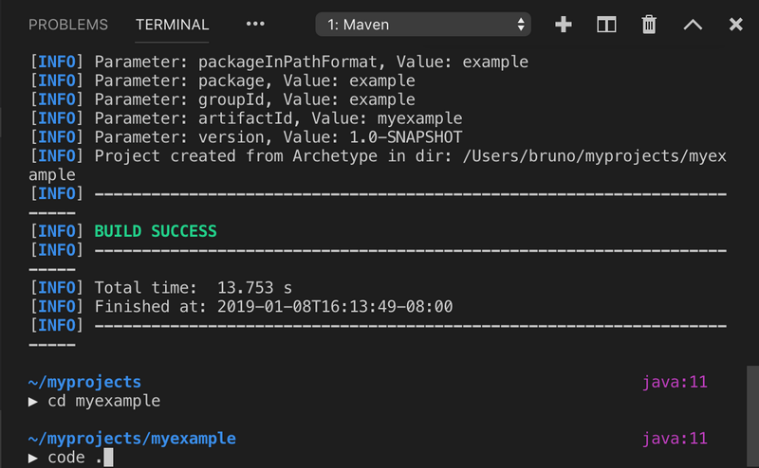
再次打开命令托盘，然后输入 Maven。

选择 Generate from Maven Archetype。

选择 maven-archetype-quickstart。

这个扩展会要求您选择目标文件夹，以便在其下面生成项目文件夹。 输入焦点会跳转到终端，您必须在那里输入 Maven 命令行的参数，不过不用担心，它会一步一步地引导您。

创建项目后，直接从终端调用 code 即可打开它。



好了，您现在应该已经在 VS Code 中打开您的 Maven 项目了。您可以做的最基本的事情就是运行您的代码。 您有两个选择：

1. 如前所述，使用 App 类中 main 方法旁边的 Run 超链接运行您的代码。

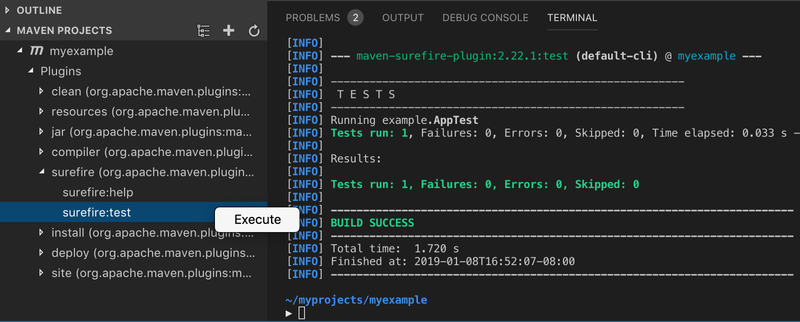
2. 使用 Maven。

如果您使用 Java 调试扩展（运行 | 调试）触发器，扩展将使用 Maven 生成的 classpath，以确保所有依赖项都正确地添加到类路径中。

如果使用 Maven 运行 Java 代码，您可以像往常一样使用终端，或者打开命令托盘并输入 Maven Execute Commands。

它会要求您选择一个项目。 由于您只有一个项目，直接在其上按回车即可。 接下来，您将看到一个列表，它包含了所有默认的核心 Maven 目标。 选择 package 生成 JAR 文件。

如果要运行自定义目标，例如从 Maven 插件继承的目标，您可以使用 Maven 视图：



编辑 pom.xml 文件并添加依赖项后，VS Code 将自动重新加载 classpath，然后您就可以从新的依赖项中导入类和包。这个过程非常干净、顺畅。

### 微软的 Java 测试执行器 Java Test Runner

最后一步是增强单元测试的运行、调试和测试结果的可视化。此扩展程序将超链接添加到可以单独执行的单元测试 (支持 JUnit 和 TestNG)，您可以立即在 VS Code 中看到报告，如下面的示例所示。



此扩展还将启用测试资源管理器视图，因此您可以专注于代码的单元测试，并以更加符合测试驱动开发（TDD）的方式编写软件。

此扩展目前仅适用于 Maven 项目，因此请确保您安装了针对 Java 的 Maven 扩展。

### 针对 Java 的 Maven 扩展 Maven for Java

### Ctrl+P 模式

直接输入文件名，快速打开文件

> 显示并运行命令

: 跳转到行数，也可以Ctrl+G直接进入(Mac 是 CMD+G)

@ 跳转到symbol（搜索变量或者函数），也可以Ctrl+Shift+O直接进入

@: 根据分类跳转symbol，查找属性或函数，也可以Ctrl+Shift+O后输入:进入

# 根据名字查找symbol，也可以Ctrl+T

### 配置参数

{

"editor.suggestSelection": "first",

"vsintellicode.modify.editor.suggestSelection": "automaticallyOverrodeDefaultValue",

"editor.renderWhitespace": "none",

"editor.renderControlCharacters": false,

"java.home": "/usr/local/jdk1.8.0\_221",

"java.configuration.maven.userSettings": "/usr/local/apache-maven-3.6.2/conf/settings.xml",

"maven.executable.path": "/usr/local/apache-maven-3.6.2/bin/mvn",

"maven.terminal.useJavaHome": true,

"maven.terminal.customEnv": [

{

"environmentVariable": "JAVA\_HOME",

"value": "/usr/local/lib/jdk1.8.0\_221"

}

],

}

### 配置完成，重启VS Code。

### 如果你的mvn更新包速度很慢，建议使用阿里云的镜像速度会快点（修改maven的setting配置如下）：

< !-- 在mirrors 中配置 mirror -->推荐使用maven阿里仓库:在settings.xml添加<mirrors>

<mirror>

<id>alimaven</id>

<name>aliyun maven</name>

<url>http://maven.aliyun.com/nexus/content/repositories/central/</url>

<mirrorOf>central</mirrorOf>

</mirror>

### 第一个VSCode java程序

输入class 代码自动补全

输入main 代码自动补全

System.out.println("Hellow world");

Fabric环境搭建

### sdk的使用方法

可以以Spring MVC作为Web接口提供项目方案。（需要学习JavaWeb相关知识）

# Fabric环境部署

## Fabric源码获取

# git clone -b release-1.4 <https://github.com/hyperledger/fabric.git>

上面的命令相当于git clone [git@github.com:hyperledger/fabric.git](mailto:git@github.com:hyperledger/fabric.git)之后并切换分支:

$ git checkout release-1.4 (推荐使用这个)

## 获取二进制文件:

需要获得cryptogen, configtxgen, configtxlator, peer, etc.等fabric编译生成的工具。

查看/fabric/release/下是否有linux-amd64/bin文件夹，且其中已经有所需二进制文件。没有的话执行以下步骤：

### 编译生成（推荐）：编译源码自动生成

推荐使用该方式，成功的话，说明配置的go，make等工具版本符合要求。如果该步骤不能成功，参考链接配置基础环境：<https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html>

进入页面: Docs » Getting Started » Prerequisites

$ cd fabric

$ make release: # 编译源码生成版本。

然后把上述下载的bin文件夹放入$GOPATH/src/github.com/hyperledger/fabric/下。并将bin加入系统路径。

$ mv release/linux-amd64/bin .

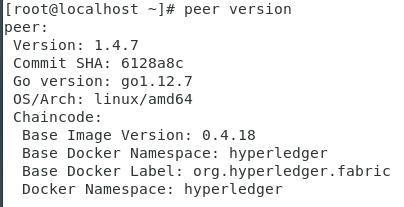
$ gedit /etc/bashrc

填上 export PATH=$PATH:$GOPATH/src/github.com/hyperledger/fabric/bin

$ source /etc/bashrc

查看peer版本(重要！！！)

$peer version



很奇怪，用的fabric-v1.4.6编译出来的怎么是1.4.7版本。

### 下载现成：根据官方文档下载现成

解析官方提供的下载方案:

在$GOPATH/src/github.com/hyperledger/fabric/scripts目录下有个bootstrap.sh，详细阅读并使用该工具！！！，并参考<https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html> 中的页面

进入页面Docs » Getting Started » Install Samples, Binaries and Docker Images

但是由于Hyperledger Nexus Repository 被弃用了。原来的二进制工具下载链接失效！！！先手动将binariesInstall()方法替换为最新版本提供的下载方法。可以参考release2.1中的bootstrap.sh

最后执行，例如：

$ ./bootstrap.sh 1.4.6 1.4.6 0.4.18 (针对fabric-v1.4.6版本)

现在，二进制工具已经下载到fabric/scripts/fabric-samples/bin下面，可以复制到fabric根目录的bin中。

### 方式三：根据Hyperledger Fabric项目说明中的提示下载

根据

<https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html>

# curl -sSL http://bit.ly/2ysbOFE | bash -s -- 1.4.6 1.4.6 0.4.18 #下载自动化部署脚本。必须翻墙才行。

## 获取fabric-samples

$ ./bootstrap.sh 1.4.0 1.4.0 0.4.14 (针对fabric-v1.4.6版本) # 默认下载到fabric/scripts/目录下。

## 下载Fabric镜像

### 稳妥方法

直接进入fabric/examples/e2e\_cli,运行./download-dockerimages.sh可以直接下载镜像，且能保证镜像与Fabric中的源码Demo版本号对应。

### 想灵活版本控制的话，使用下面的方法。

<https://hub.docker.com/>注册，搜索hyperleger

根据tag标签

# docker pull hyperledger/fabric-peer会默认下载最新的

所以使用# docker pull hyperledger/fabric-peer:x86\_64-1.1.0，但是该资源速度较慢，可以使用Docker中国官方镜像加速服务registry.docker-cn.com(只包含流行的公有镜像)（好像不行）

# docker pull registry.docker-cn.com/hyperledger/fabric-peer:x86\_64-1.1.0

### 拉取所需镜像

hyperledger/fabric-peer: $ docker pull hyperledger/fabric-peer:x86\_64-1.1.0

hyperledger/fabric-orderer: $ docker pull hyperledger/fabric-orderer:x86\_64-1.1.0

hyperledger/fabric-ccenv: $ docker pull hyperledger/fabric-ccenv:x86\_64-1.1.0

hyperledger/fabric-javaenv: $ docker pull hyperledger/fabric-javaenv:x86\_64-1.1.0

hyperledger/fabric-couchdb: $ docker pull hyperledger/fabric-couchdb:x86\_64-0.4.7 (不拉取则默认使用LevelDB)

需要将共识由Solo改为Kafka的话则还需要拉取kafka和zookeeper（1.1版本开始就需要了）

hyperledger/fabric-kafka: docker pull hyperledger/fabric-kafka:x86\_64-0.4.7(可向下兼容)

hyperledger/fabric-zookeeper: docker pull hyperledger/fabric-zookeeper:x86\_64-0.4.7(可向下兼容)

hyperledger/fabric-tools: docker pull hyperledger/fabric-tools:x86\_64-1.1.0

hyperledger/fabric-baseos: docker pull hyperledger/fabric-baseos:x86\_64-0.4.7

hyperledger/fabric-ca: docker pull hyperledger/fabric-ca:x86\_64-1.1.0

然后最终使用Docker Compose来启动对应的镜像服务。

### 镜像打标签

为了方便配置Docker Compose 将所有镜像tag改为latest，执行如下命令

# docker tag IMAGEID REPOSITORY:TAG(仓库:镜像)

$ docker tag b023f9be0771 hyperledger/fabric-peer:latest

$ docker tag ce0c810df36a hyperledger/fabric-orderer:latest

$ docker tag c8b4909d8d46 hyperledger/fabric-ccenv:latest

$ docker tag 82098abb1a17 hyperledger/fabric-javaenv:latest

$ docker tag 35228d48a25a hyperledger/fabric-couchdb:latest

$ docker tag 4fed436fc0a0 hyperledger/fabric-kafka:latest

$ docker tag 1ce465be7112 hyperledger/fabric-zookeeper:latest

$ docker tag b7bfddf508bc hyperledger/fabric-tools:latest

$ docker tag c0e784934c4e hyperledger/fabric-baseos:latest

$ docker tag 72617b4fa9b4 hyperledger/fabric-ca:latest

docker tag dbe6787b5747 hyperledger/fabric-baseimage:latest

## 镜像备份和迁移

#docker save IMAGEID > 文件路径 文件名

#docker save b023f9be0771 > /tmp/data/fabric-images-1.1.0-release/fabric-peer\_x86\_64-1.1.0.tar

#docker save ce0c810df36a > /tmp/data/fabric-images-1.1.0-release/fabric-order\_x86\_64-1.1.0.tar

#docker save c8b4909d8d46 > /tmp/data/fabric-images-1.1.0-release/fabric-ccenv\_x86\_64-1.1.0.tar

#docker save 82098abb1a17 > /tmp/data/fabric-images-1.1.0-release/fabric-javaenv\_x86\_64-1.1.0.tar

#docker save 35228d48a25a > /tmp/data/fabric-images-1.1.0-release/fabric-couchdb\_x86\_64-0.4.7.tar

#docker save 4fed436fc0a0 > /tmp/data/fabric-images-1.1.0-release/fabric-kafka\_x86\_64-0.4.7.tar

#docker save 1ce465be7112 > /tmp/data/fabric-images-1.1.0-release/fabric-zookeeper\_x86\_64-0.4.7.tar

#docker save b7bfddf508bc > /tmp/data/fabric-images-1.1.0-release/fabric-tools\_x86\_64-1.1.0.tar

#docker save c0e784934c4e > /tmp/data/fabric-images-1.1.0-release/fabric-baseos\_x86\_64-0.4.7.tar

#docker save 72617b4fa9b4 > /tmp/data/fabric-images-1.1.0-release/fabric-ca\_x86\_64-1.1.0.tar

## 传输及加载镜像文件

# scp fabric-peer.tar root@10.111.171.217:/tmp/docker/fabric-images

# docker load < /tmp/data/fabric-images-1.1.0-release//fabric-peer.tar

镜像加载完后，可根据之前的操作方案，将镜像tag改为lastest，以便配置Docker Compose。

# 编译Fabric

注意!!!：从2020年2月开始Gerrit已经不提供Hyplerdger的源码了，见第八章。所以本章节内容仅供参考，不可实际操作。

## 从Gerrit下载源码

<https://github.com/hyperledger/fabric>是只读镜像，

真正源码在Gerrit下: <https://gerrit.hyperledger.org/r/admin/repos/fabric>

1)Linux基金会网站(<https://identity.linuxfoundation.org/>)注册LFID

2)本机生成SSH key: ssh-keygen -t rsa -C “[drbjhu@163.com](mailto:drbjhu@163.com)”

3)复制生成的SSH Key: cat ~/.ssh/id\_rsa.pub

4)把该公钥添加到Gerrit的Settings->SSHKeys中

5)创建Hyperledger Fabric的源码目录:mkdir –p $GOPATH/src/github.com/hyperledger

6)cd $GOPATH/src/github.com/hyperledger

7)下载源码:根据<https://gerrit.hyperledger.org/r/admin/repos/fabric>中提供的实际链接来

git clone "https://gerrit.hyperledger.org/r/fabric" && (cd "fabric" && mkdir -p .git/hooks && curl -Lo `git rev-parse --git-dir`/hooks/commit-msg https://gerrit.hyperledger.org/r/tools/hooks/commit-msg; chmod +x `git rev-parse --git-dir`/hooks/commit-msg)

(8)切换分支: cd fabric; git checkout -b origin/release-1.1

## 安装工具

1)下载BDD测试依赖工具: pip install -r devenv/bddtests-requirements.txt

2)yum install gcc gcc-c++ make openssl-devel libtool bzip2 libtool-ltdl-devel

## 开始编译生成二进制工具（最关键的一步来了，很多人估计卡在这里）

确保Docker后台已经启动: systemctl restart docker.service

1)拉取golang/tools: （golang编译需要的工具，否则下面的make gotools不成功）

git clone [git@github.com:golang/tools.git](mailto:git@github.com:golang/tools.git)到 $GOPATH/src/golang.org/x/tools

2) cd $GOPATH/src/github.com/hyperledger/fabric

3) 生成Golang编译使用的工具: make -d gotools -d表示调试模式，可以看到比较详细的编译过程(第一次不成功，是因为墙的原因,而不能访问。参考“翻墙配置”章节。

编译好的可执行文件放在&GOPATH/bin下面

4) 生成Docker版本的gotools:

#make .build/docker/gotools（目前失败原因未知！！！）

如果失败则，手动将gotools下编译好的文件复制到以下目录

cp -r $GOPATH/bin/\* $GOPATH/src/github.com/hyperledger/fabric/.build/docker/gotools/bin

5) 如果已经生成gotools，则尝试:

make linter

6编译本地二进制文件: make native

相当于顺序编译Peer, Orderer, Configtxgen, Cryptogen, Configtxlator, 在.build/bin下可以找到生成的二进制文件

7) 生成Docker镜像:

#make docker（拉取镜像）该命令会拉取所需要的镜像 可以成功，但是时间太长

编译完成、生成的docker镜像可以通过docker images命令查看。

编译遇到问题，通常打开make的调试模式: make -d peer

## 部署Fabric网络

建立只有一个Orderer节点和一个Peer节点的简单网络

建议在每一步操作之后，在源码中搜索命令或者说明中的一些关键字来阅读源码，以加深对结构的理解。

### 配置环境

Fabric的节点通过Docker容器来运行，启动Fabric网络中的节点需要预先安装Docker、Docker-Compose和Go语言环境，然后在网上拉取相关的Docker镜像，再通过配置Compose文件来启动各个节点。

### 启动Orderer节点

ORDERER\_GENERAL\_GENESISPROFILE=SampleDevModeSolo orderer

# End-2-End案例（<<Hyperledger Fabric开发实战>>的第一个完整案例）已跑通

注意：

1、以下内容已从针对release-1.1改成针对release-1.4版本的配置。

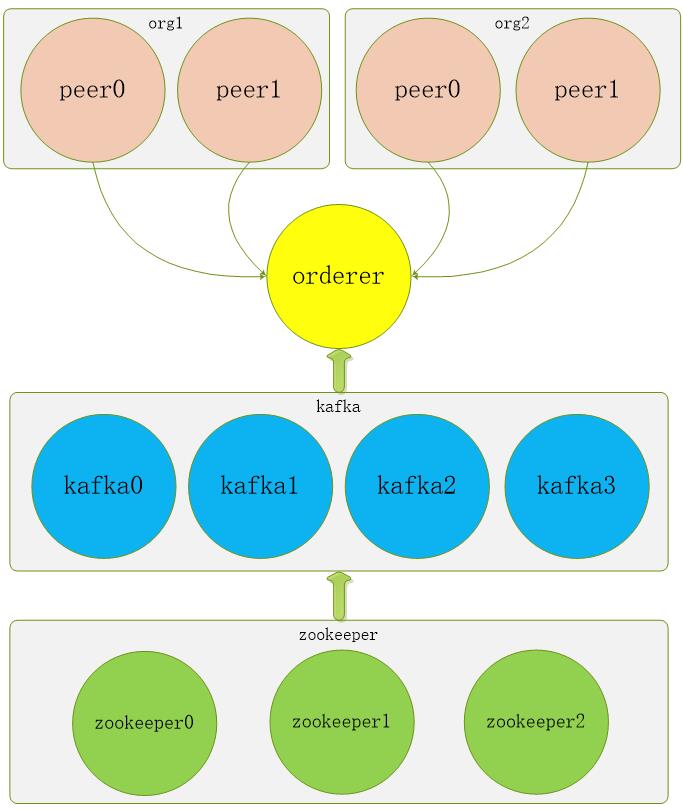
2、运行该案例前，必须确保”Fabric环境部署”章节成功完成!!!

3、该案例对应的fabric-sdk-java JUnit测试代码是End2endIT.java。学会并跑通该代码才算是初步掌握fabric-sdk-java开发!!!

## 说明

### End-2-End案例架构

End-2-End案例由3个zookeeper、4个kafka和1个orderer实现排序,包括两个组织，分别为Org1和Org2，每个组织中有两个节点，分别为peer0和peer1,结构图如下所示：



所有的配置都在docker-compose-cli.yaml文件里，配置顺序分别为zookeeper、kafka、orderer、peer和cli，先运行zookeeper集群、再运行kafka集群，最后运行orderer和peer，必须按照以上运行顺序；实现的功能集中写在script.sh文件里，自动运行全部功能，直到显示成功，具体功能如下：

1. 验证排序（orderer）服务是否可用，函数：checkOSNAvailability

2. 创建通道，函数：createChannel

3. 加入通道，函数：joinChannel

4. 更新组织1的锚节点，函数：updateAnchorPeers

5. 更新组织2的锚节点，函数：updateAnchorPeers

6. 在组织1的节点0上安装智能合约，函数：installChaincode

7. 在组织2的节点0上安装智能合约，函数：installChaincode

8. 在组织2的节点0上实例化智能合约，函数：instantiateChaincode

9. 在组织1的节点0上查询智能合约，函数：chaincodeQuery

10. 从组织1的节点0向组织2的节点0转移数据10的交易，函数：chaincodeInvoke

11. 在组织2的节点1上安装智能合约，函数：installChaincode

12. 在组织2的节点1上查询智能合约，函数：chaincodeQuery

### 文件结构

End-2-End案例的全部文件在fabric/examples/e2e\_cli目录下，文件结构如下所示：

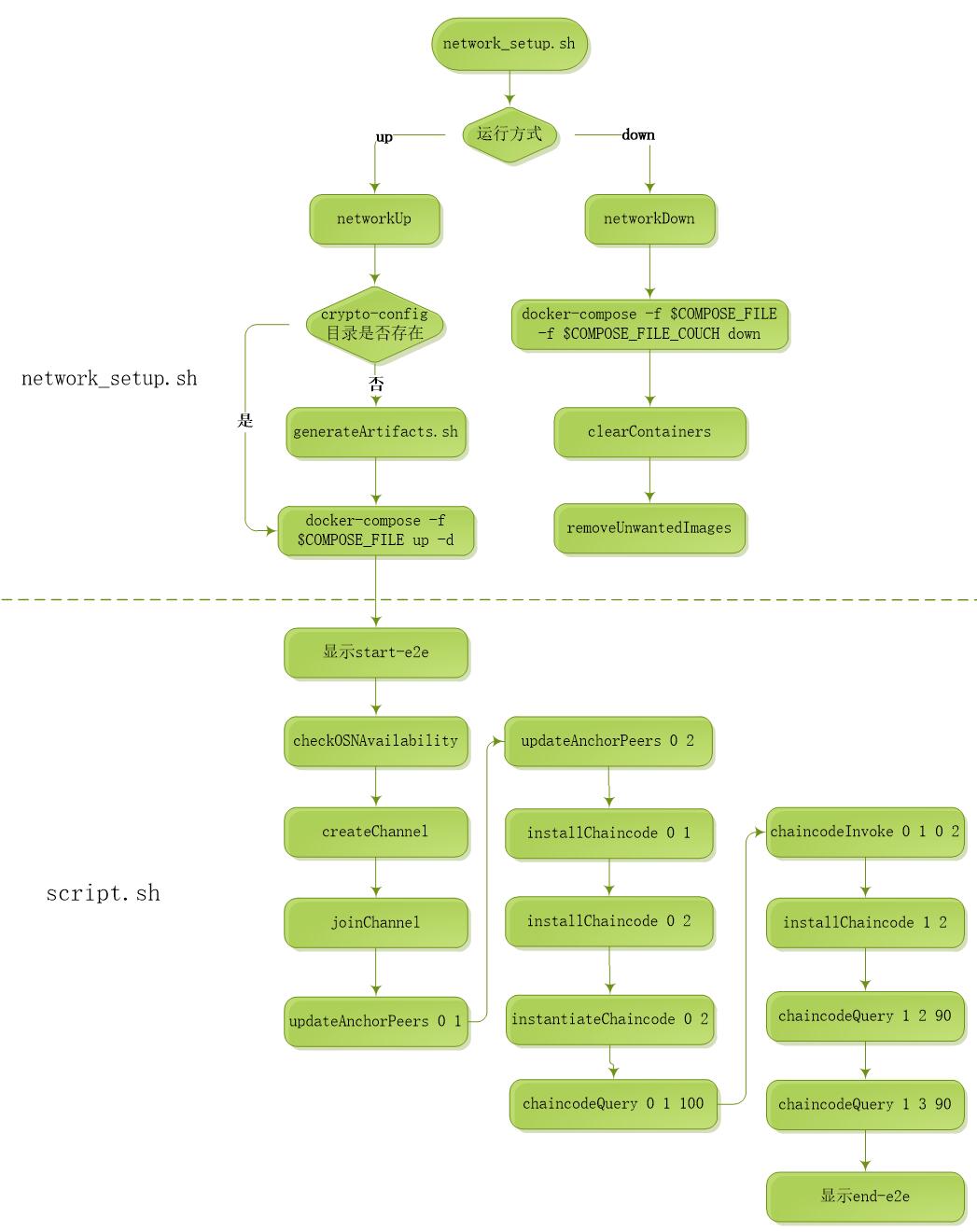


### 文件说明：

|  |  |
| --- | --- |
| **文件（或目录）名称** | **说明** |
| **base** | 存放配置提炼的公有部分，有两个文件，分别为docker-compose-base.yaml和peer-base.yaml |
| **channel-artifacts** | 存放生成的通道和创世纪块等文件，包括有channel.tx、genesis.block、Org1MSPanchors.tx (即，组织1的通道创建交易)和Org2MSPanchors.tx（即，组织2的通道创建交易） |
| **crypto-config** | 存放生成的公私钥和证书等文件 |
| **scripts** | 只有一个script.sh文件，该文件是案例的运行功能的集合，运行后会自动执行全部功能，直到完成 |
| **configtx.yaml** | 通道配置文件 |
| **crypto-config.yaml** | 生成的公私钥和证书的配置文件 |
| **docker-compose-cli.yaml** | Fabric网络Docker运行配置文件 |
| **download-dockerimages.sh** | 下载Fabric镜像执行文件 |
| **generateArtifacts.sh** | 生成公私钥和证书的执行文件 |
| **network\_setup.sh** | 案例运行的入口文件 |

### 执行流程

Fabric基础环境搭建完成后，End-2-End案例的运行先从network\_setup.sh文件执行，执行过程中调用generateArtifacts.sh生成公私钥和证书等文件，再根据docker-compose-cli.yaml的配置内容通过docker运行zookeeper、kafka、orderer、peer和cli,最后在cli中运行script.sh文件，批量执行创建通道、加入通道、安装智能合约、实例化智能合约、执行交易和执行查询等功能，以上过程在没有错误的情况下，自动执行逐行执行，直到提示END-E2E表示成功。执行详细流程如下：



### 流程说明：

1. 在e2e\_cli目录执行network\_setup.sh up表示开始执行，network\_setup.sh down表示结束执行；

2. 执行network\_setup.sh up后先判断是否存在crypto-config目录，如果不存在，则调用generateArtifacts.sh文件生成公私钥和证书；否则通过命令docker-compose -f $COMPOSE\_FILE up -d开始启动Fabric网络；

3. Fabric网络启动成功后，自动执行script.sh文件，按照代码顺序，分别执行如下代码：

1) 显示START-E2E图样：显示将开始执行案例；

2) checkOSNAvailability：执行peer channel fetch 0 0\_block.pb -o orderer.example.com:7050 -c "$ORDERER\_SYSCHAN\_ID" --tls --cafile $ORDERER\_CA >&log.txt命令，验证排序（orderer）服务是否可用；

3) createChannel：执行peer channel create -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/channel.tx --tls --cafile $ORDERER\_CA >&log.txt命令创建通道；

4) joinChannel：执行peer channel join -b $CHANNEL\_NAME.block >&log.txt命令四个peer节点加入到通道中；

5) updateAnchorPeers 0 1：执行peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/${CORE\_PEER\_LOCALMSPID}anchors.tx --tls --cafile $ORDERER\_CA >&log.txt命令更新组织1的锚节点0；

6) updateAnchorPeers 0 2：执行peer channel update -o orderer.example.com:7050 -c $CHANNEL\_NAME -f ./channel-artifacts/${CORE\_PEER\_LOCALMSPID}anchors.tx --tls --cafile $ORDERER\_CA >&log.txt命令更新组织2的锚节点0；

7) installChaincode 0 1：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织1的节点0上安装智能合约；

8) installChaincode 0 2：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织2的节点0上安装智能合约；

9) instantiateChaincode 0 2：执行peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "AND ('Org1MSP.peer','Org2MSP.peer')" >&log.txt在组织2的节点0上实例化智能合约，初始化a值为100和b值为200；

10) chaincodeQuery 0 1 100：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织1的节点0上查询a值，并判断是否为100；

11) chaincodeInvoke 0 1 0 2：执行peer chaincode invoke -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc $PEER\_CONN\_PARMS -c '{"Args":["invoke","a","b","10"]}' >&log.txt命令从a值中转称10到a值中；

12) installChaincode 1 2：执行peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd >&log.txt命令在组织2的节点1上安装智能合约；

13) chaincodeQuery 1 2 90：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织2的节点1上查询a值，并判断是否为90；

14) chaincodeQuery 1 3 90：执行peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}' >&log.txt命令在组织3的节点1上查询a值，并判断是否为90；

15) 显示end-e2e：以上代码执行没有出现错误，则显示end-e2e表示成功执行；

### 智能合约介绍

智能合约通过Go语言编写，实现存储a和b值，并在a和b之间数据交易转移，主要包括Init（初始化）、Invoke（交易）、delete（删除）和query（查询）四个函数。

### 忽略e2e\_cli/download-dockerimages.sh

忽略fabric/examples/e2e\_cli下的./download-dockerimages.sh文件，第3步已经下载好了需要的镜像，且该文件下载的镜像版本不对。

**如果使用的Fabric1.0的e2e案例，则不是kafka启动模式，如果是Fabric1.1-1.3中的e2e\_cli案例，则是kafka启动。**

network\_setup.sh启动5个Docker容器。其中4个容器运行Peer节点（分属两个不同的组织）和1个容器运行Orderer节点组成一个Fabric网络。还有个cli容器用于执行创建channel、加入channel、安装和执行chaincode等操作。

### 确保本地镜像已经下载好且准备好了tag

## 可能出现的错误或警告

### release-1.3中的原始e2e\_cli运行存在的错误

#### 错误1

Error: failed to create deliver client: orderer client failed to connect to orderer.example.com:7050: failed to create new connection: context deadline exceeded

!!!!!!!!!!!!!!! Ordering Service is not available, Please try again ... !!!!!!!!!!!!!!!!

================== ERROR !!! FAILED to execute End-2-End Scenario ==================

**原因：**通道配置文件configtx.yaml错误。

**解决办法**：修改通道配置文件configtx.yaml，将其SECTION: Capabilities参考相同版本的fabric-sdk-java/src/test/fixture/sdkintegration/e2e-2Orgs/configtx.yaml修改即可。

#### 错误2

Error: got unexpected status: BAD\_REQUEST -- error validating channel creation transaction for new channel 'mychannel', could not succesfully apply update to template configuration: error authorizing update: error validating DeltaSet: policy for [Group] /Channel/Application not satisfied: implicit policy evaluation failed - 0 sub-policies were satisfied, but this policy requires 1 of the 'Admins' sub-policies to be satisfied

原因：Only admin user can create the channel , please check you are using correct user credentials in channel create command.

**解决办法**：修改通道配置文件configtx.yaml，将其SECTION: Capabilities中的

1: Channel: &ChannelCapabilities下面改成：

比如：

# V1.4.3 for Channel is a catchall flag for behavior which has been

# determined to be desired for all orderers and peers running at the v1.4.3

# level, but which would be incompatible with orderers and peers from

# prior releases.

# Prior to enabling V1.4.3 channel capabilities, ensure that all

# orderers and peers on a channel are at v1.4.3 or later.

V1\_4\_3: true

# V1.3 for Channel enables the new non-backwards compatible

# features and fixes of fabric v1.3

V1\_3: false

# V1.1 for Channel enables the new non-backwards compatible

# features and fixes of fabric v1.1

V1\_1: false

以及2: Orderer: &OrdererCapabilities下改成：

# V1.4.2 for Orderer is a catchall flag for behavior which has been

# determined to be desired for all orderers running at the v1.4.2

# level, but which would be incompatible with orderers from prior releases.

# Prior to enabling V1.4.2 orderer capabilities, ensure that all

# orderers on a channel are at v1.4.2 or later.

V1\_4\_2: true

# V1.1 for Orderer enables the new non-backwards compatible

# features and fixes of fabric v1.1

V1\_1: false

3: Application: &ApplicationCapabilities下改成：

# V1.4.2 for Application enables the new non-backwards compatible

# features and fixes of fabric v1.4.2

V1\_4\_2: true

# V1.3 for Application enables the new non-backwards compatible

# features and fixes of fabric v1.3.

V1\_3: false

# V1.2 for Application enables the new non-backwards compatible

# features and fixes of fabric v1.2 (note, this need not be set if

# later version capabilities are set)

V1\_2: false

# V1.1 for Application enables the new non-backwards compatible

# features and fixes of fabric v1.1 (note, this need not be set if

# later version capabilities are set).

V1\_1: false

此外，在1-3条上面补充注释:

# Note that setting a later Channel version capability to true will also

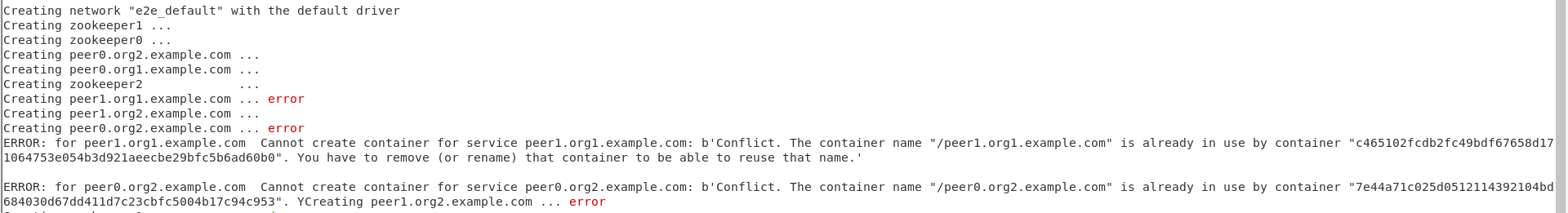
# implicitly set prior Channel version capabilities to true. There is no need

# to set each version capability to true (prior version capabilities remain

# in this sample only to provide the list of valid values).

这三个版本的确定方式：参考相同版本的fabric-sdk-java/src/test/fixture/sdkintegration/e2e-2Orgs/configtx.yaml修改即可。

#### 错误3



原因：貌似是peer镜像有问题。

解决办法：删除peer镜像重新下载即可。

### release-1.3中的原始e2e\_cli运行存在的警告



解决办法：将文件docker-compose-cli.yaml中cli容器的:

’ - CORE\_LOGGING\_LEVEL=DEBUG’

改成1：’-FABRIC\_LOGGING\_SPEC=DEBUG’，

或者:2：’-FABRIC\_LOGGIN\_SPEC=INFO’以免输出的内容过多。

学习模式改成2，改错模式改成第1种。

## 拷贝e2e\_cli源文件

由于Fabric 1.4开始删除了End-2-End案例，需要手动拷贝e2e\_cli目录到$GOPATH/src/github.com/hyperledger/fabric/examples目录下。从release-1.3 (v1.3.0和v1.3.0-rc1中的e2e\_cli一样)中拷贝即可。但是直接运行该版本会导致错误（1）。可以

## 添加可执行权限

$ chmod -R 777 $GOPATH/src/github.com/hyperledger/fabric/examples/e2e\_cli

-R表示对目前目录下的所有文件与子目录进行相同的权限变更(即以递回的方式逐个变更)

## 开始测试

$ cd $GOPATH/src/github.com/hyperledger/fabric/examples/e2e\_cli

$ bash network\_setup.sh up

如果失败，重启的话必须先# bash network\_setup.sh down以提供一个干净的环境。

$ docker ps #新的terminal查看已经启动的容器，启动容器必须要有镜像支持

# 单机多节点部署

启动一个Orderer排序服务节点🡪启动peer0.org1节点并在该节点执行通道和合约的创建等相关操作🡪启动peer0.org2，并将其加入到网络及网络所属通道中，实现伪动态加入。

## 生成证书

如果运行过e2e\_cli案例，环境清理:

$ cd $GOPATH/src/github.com/hyperledger/fabric/examples/e2e\_cli

$ bash network\_setup.sh down

$ docker ps #查看已经启动的容器

$ docker ps -a #查看所有的容器

$ docker images #列出本地镜像

$ go version #go 1.10 linux/amd64

$ docker-compose version

Result :

docker-compose version 1.19.0, build9e633ef

docker-py version:2.7.0

CPython version 2.7.13

OpenSSL version: OpenSSL 1.0.1t 3 May 2016

$ cd $GOPATH/src/github.com/hyperledger/fabric

$ mkdir aberic

使用的configtx.yaml及crypto-config.yaml配置文件来自Fabric1.0，即非Kafka启动类型。

上传至aberic目录下。

### 根据crypto-config.yaml中的组织架构生成证书文件

../release/linux-amd64/bin/cryptogen generate --config=./crypto-config.yaml(推荐编译生成的工具)

### 根据configtx.yaml生成创世区块

创世区块是为了排序节点服务启动时用的。

确保PWD路径为aberic

为configtxgen工具指定configtx.yaml文件路径

#export FABRIC\_ROOT=$PWD/..

#export FABRIC\_CFG\_PATH=$PWD

#mkdir -p channel-artifacts

../release/linux-amd64/bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block (推荐编译生成的工具)

### 生成Peer节点所需的通道文件，通过该文件，peer可以执行通道创建工作

../release/linux-amd64/bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/mychannel.tx -channelID mychannel

## 部署Orderer节点

单机多节点部署，启动类型Solo，非Kafka。

orderer.example.com监听端口: 0.0.0.0:7050->7050/tcp

编写docker-orderer.yaml启动文件:

# Copyright IBM Corp. All Rights Reserved.

#

# SPDX-License-Identifier: Apache-2.0

#

version: '2'

services:

orderer.example.com:

container\_name: orderer.example.com

image: hyperledger/fabric-orderer

environment:

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=aberic\_default

# - ORDERER\_GENERAL\_LOGLEVEL=error

- ORDERER\_GENERAL\_LOGLEVEL=debug

- ORDERER\_GENERAL\_LISTENADDRESS=0.0.0.0

- ORDERER\_GENERAL\_LISTENPORT=7050

#- ORDERER\_GENERAL\_GENESISPROFILE=AntiMothOrdererGenesis

- ORDERER\_GENERAL\_GENESISMETHOD=file

- ORDERER\_GENERAL\_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block

- ORDERER\_GENERAL\_LOCALMSPID=OrdererMSP

- ORDERER\_GENERAL\_LOCALMSPDIR=/var/hyperledger/orderer/msp

#- ORDERER\_GENERAL\_LEDGERTYPE=ram

#- ORDERER\_GENERAL\_LEDGERTYPE=file

# enabled TLS

- ORDERER\_GENERAL\_TLS\_ENABLED=false

- ORDERER\_GENERAL\_TLS\_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key

- ORDERER\_GENERAL\_TLS\_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt

- ORDERER\_GENERAL\_TLS\_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric

command: orderer

volumes:

- ./channel-artifacts/genesis.block:/var/hyperledger/orderer/orderer.genesis.block

- ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/msp:/var/hyperledger/orderer/msp

- ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/:/var/hyperledger/orderer/tls

networks:

default:

aliases:

- aberic

ports:

- 7050:7050

## 部署peer0.org1节点

还需专门为peer节点准备docker-peer.yaml启动文件。Orderer和Peer的启动yaml文件名都可以根据实际需求自定义。

peer0.org1监听端口: 0.0.0.0:7051-7053->7051-7053/tcp

couchdb端口: 4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp

ca端口: 0.0.0.0:7054->7054/tcp

注意根据自己的实际情况修改:

1. 两处\_sk的文件名替换成peerOrganization/org1.example.com/ca中的\*\_sk文件名

# Copyright IBM Corp. All Rights Reserved.

#

# SPDX-License-Identifier: Apache-2.0

#

version: '2'

services:

couchdb:

container\_name: couchdb

image: hyperledger/fabric-couchdb

# Comment/Uncomment the port mapping if you want to hide/expose the CouchDB service,

# for example map it to utilize Fauxton User Interface in dev environments.

ports:

- "5984:5984"

ca:

container\_name: ca

image: hyperledger/fabric-ca

environment:

- FABRIC\_CA\_HOME=/etc/hyperledger/fabric-ca-server

- FABRIC\_CA\_SERVER\_CA\_NAME=ca

- FABRIC\_CA\_SERVER\_TLS\_ENABLED=false

- FABRIC\_CA\_SERVER\_TLS\_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem

- FABRIC\_CA\_SERVER\_TLS\_KEYFILE=/etc/hyperledger/fabric-ca-server-config/f84a57190d28774864600673b45b5ed9f71028c4339e1bcae201edf217e8c127\_sk

ports:

- "7054:7054"

command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-server-config/ f84a57190d28774864600673b45b5ed9f71028c4339e1bcae201edf217e8c127\_sk -b admin:adminpw -d'

volumes:

- ./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/fabric-ca-server-config

peer0.org1.example.com:

container\_name: peer0.org1.example.com

image: hyperledger/fabric-peer

environment:

- CORE\_LEDGER\_STATE\_STATEDATABASE=CouchDB

- CORE\_LEDGER\_STATE\_COUCHDBCONFIG\_COUCHDBADDRESS=couchdb:5984

- CORE\_PEER\_ID=peer0.org1.example.com

- CORE\_PEER\_NETWORKID=aberic

- CORE\_PEER\_ADDRESS=peer0.org1.example.com:7051

- CORE\_PEER\_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052

- CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=peer0.org1.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org1MSP

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

# the following setting starts chaincode containers on the same

# bridge network as the peers

# https://docs.docker.com/compose/networking/

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=aberic

# - CORE\_LOGGING\_LEVEL=ERROR

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=aberic\_default

- CORE\_PEER\_GOSSIP\_SKIPHANDSHAKE=true

- CORE\_PEER\_GOSSIP\_USELEADERELECTION=true

- CORE\_PEER\_GOSSIP\_ORGLEADER=false

- CORE\_PEER\_PROFILE\_ENABLED=false

- CORE\_PEER\_TLS\_ENABLED=false

- CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/fabric/tls/server.crt

- CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/fabric/tls/server.key

- CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/fabric/tls/ca.crt

volumes:

- /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/msp:/etc/hyperledger/fabric/msp

- ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls:/etc/hyperledger/fabric/tls

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer

command: peer node start

ports:

- 7051:7051

- 7052:7052

- 7053:7053

depends\_on:

- couchdb

networks:

default:

aliases:

- aberic

cli:

container\_name: cli

image: hyperledger/fabric-tools

tty: true

environment:

- GOPATH=/opt/gopath

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

# - CORE\_LOGGING\_LEVEL=ERROR

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_PEER\_ID=cli

- CORE\_PEER\_ADDRESS=peer0.org1.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org1MSP

- CORE\_PEER\_TLS\_ENABLED=false

- CORE\_PEER\_TLS\_CERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt

- CORE\_PEER\_TLS\_KEY\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key

- CORE\_PEER\_TLS\_ROOTCERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

- CORE\_PEER\_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer

volumes:

- /var/run/:/host/var/run/

- ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/chaincode/go

- ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/

- ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/channel-artifacts

depends\_on:

- peer0.org1.example.com

## 搭建Fabric网络

应为dacker-peer.yaml文件中指定了智能合约的部署路径，所以

#mkdir -p chaincode/go

将官方的chaincode\_exampmle02上传到go目录下

### 启动order

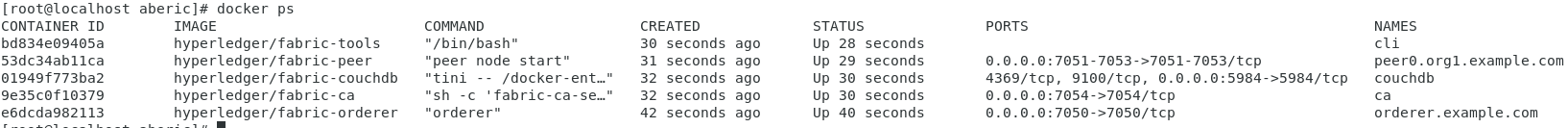
#docker-compose -f docker-orderer.yaml up -d

PS：关闭是docker-compose -f docker-orderer.yaml down命令

### 启动peer0.org1

#docker-compose -f docker-peer.yaml up -d

#docker ps #查看已启动的容器



### Channel的创建及加盟

目前还没用到第三方SDK，但安装了fabric-tools镜像，即已经启动了容器服务中的cli客户端，通过以下命令进入客户端对Channel进行相关操作

#docker exec -it cli bash #cli为docker-peer.yaml中定义的container\_name: cli

#peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-artifacts/mychannel.tx

channelID和通道文件名(e.g., mychannel.tx)都得跟之前的匹配，如果执行不成功，可能是环境污染:先停止ordererer和peer节点🡪删除hyperledger/fabric-tools镜像并利用备份重新加载，然后重新启动所有容器。

#ls #查看生成的mychannel.block文件

#peer channel join -b mychannel.block #通过mychannel.blcok文件加入该channel，以便后续可以安装和、实例化和测试智能合约。

至此，已经完成channel的创建并成功加入该channel，即一个最小单位的Fabric网络已经成功搭建起来!!!。

## 初步接触智能合约

在搭建好的Fabric网络执行合适的智能合约来实现具体的功能，以使得现实中的项目得以落地。

### 在cli客户端安装智能合约(注意必须在cli客户端环境中，即:hyperleger/fabric-tools)

#peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/go/chaincode\_example02 -v 1.0

### 实例化智能合约

#peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mychannel -c '{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member')" -v 1.0

#千万注意标点符号不能是中文版的。

指明只有Org1的成员才有背书能力，即智能合约的invoke操作只有Org1才会执行成功。

### 验证智能合约的query方法

#peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'

### 验证智能合约的invoke方法

#peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke","A","B","5"]}'

## 部署peer0.org2节点

peer0.org2监听端口: 0.0.0.0:8051->7051/tcp, 0.0.0.0:8052->7052/tcp, 0.0.0.0:8053->7053/tcp

单机多节点部署，尽管节点都属于同一个端口，但是对外映射到物理机上的端口不同，其中8051,8052,8053别是peer节点、监听及事件端口映射到物理机上的实际端口。

# Copyright IBM Corp. All Rights Reserved.

#

# SPDX-License-Identifier: Apache-2.0

#

version: '2'

services:

peer0.org2.example.com:

container\_name: peer0.org2.example.com

image: hyperledger/fabric-peer

environment:

- CORE\_PEER\_ID=peer0.org2.example.com

- CORE\_PEER\_NETWORKID=aberic

- CORE\_PEER\_ADDRESS=peer0.org2.example.com:7051

- CORE\_PEER\_CHAINCODELISTENADDRESS=peer0.org2.example.com:7052

- CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=peer0.org2.example.com:7051

- CORE\_PEER\_LOCALMSPID=Org2MSP

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

# the following setting starts chaincode containers on the same

# bridge network as the peers

# https://docs.docker.com/compose/networking/

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=aberic

# - CORE\_LOGGING\_LEVEL=ERROR

- CORE\_LOGGING\_LEVEL=DEBUG

- CORE\_VM\_DOCKER\_HOSTCONFIG\_NETWORKMODE=aberic\_default

- CORE\_PEER\_GOSSIP\_SKIPHANDSHAKE=true

- CORE\_PEER\_GOSSIP\_USELEADERELECTION=true

- CORE\_PEER\_GOSSIP\_ORGLEADER=false

- CORE\_PEER\_PROFILE\_ENABLED=false

- CORE\_PEER\_TLS\_ENABLED=false

- CORE\_PEER\_TLS\_CERT\_FILE=/etc/hyperledger/fabric/tls/server.crt

- CORE\_PEER\_TLS\_KEY\_FILE=/etc/hyperledger/fabric/tls/server.key

- CORE\_PEER\_TLS\_ROOTCERT\_FILE=/etc/hyperledger/fabric/tls/ca.crt

volumes:

- /var/run/:/host/var/run/

- ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/msp:/etc/hyperledger/fabric/msp

- ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls:/etc/hyperledger/fabric/tls

working\_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer

command: peer node start

ports:

- 8051:7051

- 8052:7052

- 8053:7053

networks:

default:

aliases:

- aberic

## 启动peer0.org2节点

#exit #退出cli容器

#cd $GOPATH/src/github.com/hyperledger/fabric/aberic

#docker-compose -f docker-peer1.yaml up -d

#docker exec -it cli bash

### 更改全局变量，使得cli客户端可以对peer0.org2进行相关操作。

CORE\_PEER\_ID=peer0.org2.example.com

CORE\_PEER\_ADDRESS=peer0.org2.example.com:7051

CORE\_PEER\_CHAINCODELISTENADDRESS=peer0.org2.example.com:7052

CORE\_PEER\_GOSSIP\_EXTERNALENDPOINT=peer0.org2.example.com:7051

CORE\_PEER\_LOCALMSPID=Org2MSP

CORE\_PEER\_TLS\_ROOTCERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

CORE\_PEER\_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

PS：接下来都是对peer2Org2进行操作

### Peer0.org2加入通道

#peer channel join -b mychannel.block

### 安装智能合约

#peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/go/chaincode\_example02 -v 1.0

### 测试合约是否已成功运行

#peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'

### 执行B转账给A价值5的资产

#peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke","B","A","5"]}'

对A的资产查询，发现还是5，没变化，这是因为实例化智能合约（智能合约只需要实例化一次？？？）的时候选择的背书组织为Org1，说明来自Org2的对资产的变更的操作都未经过背书，即不具备任何效力。但是因为Org2加入了该通道，且安装了合法的智能合约，可以对区块链中的数据进行检索。

#exit #退出当前cli然后重新进入cli容器，则cli容器恢复默认对peer0.Org1节点进行操作。查询后A的资产变为10，表示合约执行成功，应对了Org1才具备背书能力的问题。

# Solo多机部署

## 利用VMware Workstation虚拟机中的克隆技术

从orderer节点创建出peer0org1、peer1org1、peer0org2、peer1org2四个peer节点，原先的节点作为orderer排序节点。并在Fabric根目录新建aberic文件夹

|  |  |  |  |
| --- | --- | --- | --- |
| Name | IP | Hostname | Organization |
| orderer | 192.168.163.137 | orderer.example.com | orderer |
| peer0 | 192.168.163.135 | peer0.org1.example.com | org1 |
| peer1 | 192.168.163.136 | peer1.org1.example.com | org1 |
| peer0 | 192.168.163.132 | peer0.org2.example.com | org2 |
| peer1 | 192.168.163.134 | peer1.org2.example.com | org2 |

## 部署orderer节点

将crypto-config.yaml, configtx.yaml, docker-orderer.yaml上传到aberic文件夹

### 根据crypto-config.yaml中的组织架构生成证书文件

../release/linux-amd64/bin/cryptogen generate --config=./crypto-config.yaml(推荐编译生成的工具)

### 根据configtx.yaml生成创世区块

创世区块是为了排序节点服务启动时用的。

确保PWD路径为aberic

为configtxgen工具指定configtx.yaml文件路径

#export FABRIC\_ROOT=$PWD/..

#export FABRIC\_CFG\_PATH=$PWD

#mkdir -p channel-artifacts

../release/linux-amd64/bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block(推荐编译生成的configtxgen工具)

### 生成Peer节点所需的通道文件。通过该文件，peer可以执行通道创建工作

通道一：供组织1用

../release/linux-amd64/bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/mychannel.tx -channelID mychannel

通道二:供组织2用

../release/linux-amd64/bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/mychannel2.tx -channelID mychannel2

### 启动orderer排序服务节点

#docker-compose -f docker-orderer.yaml up -d

#docker ps #查看是否启动成功

## 部署peer0org1节点

### 将docker-peer01.yaml上传到aberic文件夹

### 从服务器获取所需certs和mychannel.tx等相关文件到aberic文件夹。

#mkdir -p ./crypto-config/peerOrganizations/org1.example.com

#scp -r [root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org1.example.com/\*](mailto:root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org1.example.com/*) ./crypto-config/peerOrganizations/org1.example.com/

#scp [root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/mychannel.tx ./channel-artifacts/mychannel.tx](mailto:root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/mychannel.tx%20./channel-artifacts/mychannel.tx)

### 将ca证书私钥替换为Org1的实际私钥（2处）

从orderer服务器获取Org1的私钥:

#PRIV\_KEY=$(ls crypto-config/peerOrganizations/org1.example.com/ca/\*\_sk | awk -F/ '{print $5}')

#sed -i "s/[a-z0-9]\{64\}\_sk/$PRIV\_KEY/g" docker-peer01.yaml

查看是否修改成功:#nl docker-peer01.yaml | sed -n "/\_sk/p"

### 替换docker-peer01.yaml 中的orderer节点IP为实际IP（2处）:

#

#sed -i "s/\"orderer.example.com:.\*\"/\"orderer.example.com:192.168.163.137\"/g" docker-peer01.yaml

查看是否修改成功:#nl docker-peer01.yaml | sed -n "/orderer.example.com/p"

### 替换peer0org1节点IP为实际IP（1处）:

#获取本机IP

#LocalIP=$(/sbin/ifconfig ens33 | grep 'inet ' | sed 's/^.\*inet //g' | sed 's/ netmask.\*$//g')

#替换

#sed -i "s/\"peer0.org1.example.com:.\*\"/\"peer0.org1.example.com:$LocalIP\"/g" docker-peer01.yaml

查看是否修改成功:#nl docker-peer01.yaml | sed -n "/\"peer0.org1.example.com:.\*\"/p"

### 启动peer0org1节点

#docker-compose -f docker-peer01.yaml up -d

### 进入cli客户端

#docker exec -it cli bash

### 创建通道,生成mychannel.block文件

#peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-artifacts/mychannel.tx

### 通过mychannel.block加入通道

#peer channel join -b mychannel.block

### 安装智能合约

#peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/go/chaincode\_example02 -v 1.0

### 实例化智能合约

#peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mychannel -c '{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member')" -v 1.0

#千万注意标点符号不能是中文版的。

指明只有Org1的成员才有背书能力，即智能合约的invoke操作只有Org1才会执行成功。

### 验证query方法

#peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'

### 验证invoke方法

#peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke","A","B","5"]}'

## 部署peer1org1节点

### 将docker-peer11.yaml上传到aberic文件夹

### 从服务器获取所需certs和mychannel.tx等相关文件到aberic文件夹。

#mkdir -p ./crypto-config/peerOrganizations/org1.example.com

#scp -r [root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org1.example.com/\*](mailto:root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org1.example.com/*) ./crypto-config/peerOrganizations/org1.example.com/

#scp [root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/mychannel.tx ./channel-artifacts/mychannel.tx](mailto:root@192.168.163.137:/opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/mychannel.tx%20./channel-artifacts/mychannel.tx)

### 将ca证书私钥替换为Org1的实际私钥

从orderer服务器获取Org1的私钥:

#PRIV\_KEY=$(ls crypto-config/peerOrganizations/org1.example.com/ca/\*\_sk | awk -F/ '{print $5}')

#sed -i "s/[a-z0-9]\{64\}\_sk/$PRIV\_KEY/g" docker-peer11.yaml

查看是否修改成功:#nl docker-peer11.yaml | sed -n "/\_sk/p"

### 替换docker-peer11.yaml 中的orderer节点IP为实际IP:

#

#sed -i "s/\"orderer.example.com:.\*\"/\"orderer.example.com:192.168.163.137\"/g" docker-peer11.yaml

查看是否修改成功:#nl docker-peer11.yaml | sed -n "/orderer.example.com/p"

### 替换peer0org1节点IP为实际IP:

#获取本机IP

#LocalIP=$(/sbin/ifconfig ens33 | grep 'inet ' | sed 's/^.\*inet //g' | sed 's/ netmask.\*$//g')

#替换

#sed -i "s/\"peer0.org1.example.com:.\*\"/\"peer1.org1.example.com:$LocalIP\"/g" docker-peer11.yaml

查看是否修改成功:#nl docker-peer11.yaml | sed -n "/\"peer1.org1.example.com:.\*\"/p"

### 启动peer1org1节点

#docker-compose -f docker-peer11.yaml up -d

### 获取peer0org1的客户端中的mychannel.block，并通过它加入通道

# exit #peer0org1退出客户端

#docker ps | grep "cli" | awk '{print $1}' #获取peer0org1的”cli”容器的ID

#docker cp be8f8ffc0dff:/opt/gopath/src/github.com/hyperledger/fabric/peer/mychannel.block /opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/

将mychannel.block文件从peer0.org1复制到peer1.org1中。

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

#docker ps | grep "cli" | awk '{print $1}' #获取peer1org1的”cli”容器的ID

#docker cp /opt/gopath/src/github.com/hyperledger/fabric/aberic/channel-artifacts/mychannel.block a225385f59f6:/opt/gopath/src/github.com/hyperledger/fabric/peer/

#docker exec -it cli bash #进入cli客户端

#peer channel join -b mychannel.block

### 安装智能合约

#peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/go/chaincode\_example02 -v 1.0

### 验证query方法

#peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'

### 验证invoke方法

#peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke","A","B","5"]}'

## 部署peer0org2节点和部署peer1org2节点

参考3、4点

# 智能合约

## 编写智能合约

package main

import (

"fmt"

"github.com/hyperledger/fabric/core/chaincode/shim"

"github.com/hyperledger/fabric/protos/peer"

)

// SimpleAsset example simple Chaincode implementation to manage asset

// SimpleAsset 实现一个简单的智能合约来管理资产

type SimpleAsset struct {

}

// Init init any data during the instantiate of Chaincode

// Init在智能合约实例化过程中被调用来初始化任何数据

// 注意，智能合约升级也调用这个方法来重置或迁移数据，

// 所以要小心避免在无意中破坏账本数据的情况！

func (t \*SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

args := stub.GetStringArgs()

if len(args) != 2 {

return shim.Error("Incorrect arguments. Expecting a key and a value")

}

// 通过调用stub.PutState()方法来存入资产信息

// 将资产的value和相关联key存入账本

err := stub.PutState(args[0], []byte(args[1]))

if err != nil {

return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))

}

return shim.Success(nil)

}

// Invoke each transaction will use Invoke method

// 在智能合约的眉笔交易都会调用Invoke

// “set”方法可以通过指定信的键值对来创建新资产

func (t \*SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

// 从交易中获取方法名和请求参数数组。

fn, args := stub.GetFunctionAndParameters()

var result string

var err error

if fn == "set" {

result, err = set(stub, args)

} else {

result, err = get(stub, args)

}

if err != nil {

return shim.Error(err.Error())

}

return shim.Success([]byte(result))

}

// set store asset into ledger

// 将资产(包括key 和value )存储在账本上。

// 如果key存在，它会用新value覆盖value

func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {

if len(args) != 2 {

return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")

}

err := stub.PutState(args[0], []byte(args[1]))

if err != nil {

return "", fmt.Errorf("Failed to set asset:%s", args[0])

}

return args[1], nil

}

// get get asset from ledger

// 获取指定资产key的value

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {

if len(args) != 1 {

return "", fmt.Errorf("Incorrect arguments. Expecting a key")

}

value, err := stub.GetState(args[0])

if err != nil {

return "", fmt.Errorf("Failed to set asset:%s with error: %s", args[0], err)

}

return string(value), nil

}

// main方法在实例化过程中启动容器中的智能合约

func main() {

if err := shim.Start(new(SimpleAsset)); err != nil {

fmt.Printf("Error starting SimpleAsset chaincode: %s", err)

}

}

## 构建智能合约环境

#go get -u github.com/hyperledger/fabric/core/chaincode/shim #根据要求和实际情况从互联网上下载或更新指定的代码包及其依赖包，并对它们进行编译和安装

出错则可能需要执行以下命令

# git branch --set-upstream-to=remotes/origin/release-1.1 v1.1.0

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

#cd $GOPATH/src/github.com/hyperledger/fabric/core/chaincode/shim

#go build #编译我们指定的源码文件或代码包以及它们的依赖包

如果我们在执行go build命令时不后跟任何代码包，那么命令将试图编译当前目录所对应的代码包

## 安装Hyperledger Fabric Samples

1) #git clone -b release-1.1 <https://github.com/hyperledger/fabric-samples.git>

2). 在fabric根目录下#make release: 编译源码，最终会在下面这个目录生成所需要的二进制文件/opt/gopath/src/github.com/hyperledger/fabric/release/linux-amd64，

3). 将二进制文件放到fabric-example的根目录的bin子目录中，并将其添加到PATH环境变量中。

4). 根据上一步额外生成的get-docker-images.sh骤获取所需的镜像，处理方式参考第六章。

5) 如果能翻墙的话， 3)和4)步可用

#cd fabric-samples

#curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0 #自动化下载处理。

它会自动下载执行<https://raw.githubusercontent.com/hyperledger/fabric/master/scripts/bootstrap.sh>中的以下脚本。获取所需二进制文件（存放到bin目录中）及镜像。

#!/bin/bash

#

# Copyright IBM Corp. All Rights Reserved.

#

# SPDX-License-Identifier: Apache-2.0

#

# if version not passed in, default to latest released version

VERSION=1.4.3

# if ca version not passed in, default to latest released version

CA\_VERSION=1.4.3

# current version of thirdparty images (couchdb, kafka and zookeeper) released

THIRDPARTY\_IMAGE\_VERSION=0.4.15

ARCH=$(echo "$(uname -s|tr '[:upper:]' '[:lower:]'|sed 's/mingw64\_nt.\*/windows/')-$(uname -m | sed 's/x86\_64/amd64/g')")

MARCH=$(uname -m)

printHelp() {

echo "Usage: bootstrap.sh [version [ca\_version [thirdparty\_version]]] [options]"

echo

echo "options:"

echo "-h : this help"

echo "-d : bypass docker image download"

echo "-s : bypass fabric-samples repo clone"

echo "-b : bypass download of platform-specific binaries"

echo

echo "e.g. bootstrap.sh 1.4.3 -s"

echo "would download docker images and binaries for version 1.4.3"

}

# dockerFabricPull() pulls docker images from fabric and chaincode repositories

# note, if a docker image doesn't exist for a requested release, it will simply

# be skipped, since this script doesn't terminate upon errors.

dockerFabricPull() {

local FABRIC\_TAG=$1

for IMAGES in peer orderer ccenv tools baseos nodeenv javaenv; do

echo "==> FABRIC IMAGE: $IMAGES"

echo

docker pull "hyperledger/fabric-$IMAGES:$FABRIC\_TAG"

docker tag "hyperledger/fabric-$IMAGES:$FABRIC\_TAG" "hyperledger/fabric-$IMAGES"

done

}

dockerThirdPartyImagesPull() {

local THIRDPARTY\_TAG=$1

for IMAGES in couchdb kafka zookeeper; do

echo "==> THIRDPARTY DOCKER IMAGE: $IMAGES"

echo

docker pull "hyperledger/fabric-$IMAGES:$THIRDPARTY\_TAG"

docker tag "hyperledger/fabric-$IMAGES:$THIRDPARTY\_TAG" "hyperledger/fabric-$IMAGES"

done

}

dockerCaPull() {

local CA\_TAG=$1

echo "==> FABRIC CA IMAGE"

echo

docker pull "hyperledger/fabric-ca:$CA\_TAG"

docker tag "hyperledger/fabric-ca:$CA\_TAG" "hyperledger/fabric-ca"

}

samplesInstall() {

# clone (if needed) hyperledger/fabric-samples and checkout corresponding

# version to the binaries and docker images to be downloaded

if [ -d first-network ]; then

# if we are in the fabric-samples repo, checkout corresponding version

echo "===> Checking out v${VERSION} of hyperledger/fabric-samples"

git checkout v${VERSION}

elif [ -d fabric-samples ]; then

# if fabric-samples repo already cloned and in current directory,

# cd fabric-samples and checkout corresponding version

echo "===> Checking out v${VERSION} of hyperledger/fabric-samples"

cd fabric-samples && git checkout v${VERSION}

else

echo "===> Cloning hyperledger/fabric-samples repo and checkout v${VERSION}"

git clone -b master https://github.com/hyperledger/fabric-samples.git && cd fabric-samples && git checkout v${VERSION}

fi

}

# Incrementally downloads the .tar.gz file locally first, only decompressing it

# after the download is complete. This is slower than binaryDownload() but

# allows the download to be resumed.

binaryIncrementalDownload() {

local BINARY\_FILE=$1

local URL=$2

curl -f -s -C - "${URL}" -o "${BINARY\_FILE}" || rc=$?

# Due to limitations in the current Nexus repo:

# curl returns 33 when there's a resume attempt with no more bytes to download

# curl returns 2 after finishing a resumed download

# with -f curl returns 22 on a 404

if [ "$rc" = 22 ]; then

# looks like the requested file doesn't actually exist so stop here

return 22

fi

if [ -z "$rc" ] || [ $rc -eq 33 ] || [ $rc -eq 2 ]; then

# The checksum validates that RC 33 or 2 are not real failures

echo "==> File downloaded. Verifying the md5sum..."

localMd5sum=$(md5sum "${BINARY\_FILE}" | awk '{print $1}')

remoteMd5sum=$(curl -s "${URL}".md5)

if [ "$localMd5sum" == "$remoteMd5sum" ]; then

echo "==> Extracting ${BINARY\_FILE}..."

tar xzf ./"${BINARY\_FILE}" --overwrite

echo "==> Done."

rm -f "${BINARY\_FILE}" "${BINARY\_FILE}".md5

else

echo "Download failed: the local md5sum is different from the remote md5sum. Please try again."

rm -f "${BINARY\_FILE}" "${BINARY\_FILE}".md5

exit 1

fi

else

echo "Failure downloading binaries (curl RC=$rc). Please try again and the download will resume from where it stopped."

exit 1

fi

}

# This will attempt to download the .tar.gz all at once, but will trigger the

# binaryIncrementalDownload() function upon a failure, allowing for resume

# if there are network failures.

binaryDownload() {

local BINARY\_FILE=$1

local URL=$2

echo "===> Downloading: " "${URL}"

# Check if a previous failure occurred and the file was partially downloaded

if [ -e "${BINARY\_FILE}" ]; then

echo "==> Partial binary file found. Resuming download..."

binaryIncrementalDownload "${BINARY\_FILE}" "${URL}"

else

curl "${URL}" | tar xz || rc=$?

if [ -n "$rc" ]; then

echo "==> There was an error downloading the binary file. Switching to incremental download."

echo "==> Downloading file..."

binaryIncrementalDownload "${BINARY\_FILE}" "${URL}"

else

echo "==> Done."

fi

fi

}

binariesInstall() {

echo "===> Downloading version ${FABRIC\_TAG} platform specific fabric binaries"

binaryDownload "${BINARY\_FILE}" "https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric/hyperledger-fabric/${ARCH}-${VERSION}/${BINARY\_FILE}"

if [ $? -eq 22 ]; then

echo

echo "------> ${FABRIC\_TAG} platform specific fabric binary is not available to download <----"

echo

fi

echo "===> Downloading version ${CA\_TAG} platform specific fabric-ca-client binary"

binaryDownload "${CA\_BINARY\_FILE}" "https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric-ca/hyperledger-fabric-ca/${ARCH}-${CA\_VERSION}/${CA\_BINARY\_FILE}"

if [ $? -eq 22 ]; then

echo

echo "------> ${CA\_TAG} fabric-ca-client binary is not available to download (Available from 1.1.0-rc1) <----"

echo

fi

}

dockerInstall() {

command -v docker >& /dev/null

NODOCKER=$?

if [ "${NODOCKER}" == 0 ]; then

echo "===> Pulling fabric Images"

dockerFabricPull "${FABRIC\_TAG}"

echo "===> Pulling fabric ca Image"

dockerCaPull "${CA\_TAG}"

echo "===> Pulling thirdparty docker images"

dockerThirdPartyImagesPull "${THIRDPARTY\_TAG}"

echo

echo "===> List out hyperledger docker images"

docker images | grep hyperledger

else

echo "========================================================="

echo "Docker not installed, bypassing download of Fabric images"

echo "========================================================="

fi

}

DOCKER=true

SAMPLES=true

BINARIES=true

# Parse commandline args pull out

# version and/or ca-version strings first

if [ -n "$1" ] && [ "${1:0:1}" != "-" ]; then

VERSION=$1;shift

if [ -n "$1" ] && [ "${1:0:1}" != "-" ]; then

CA\_VERSION=$1;shift

if [ -n "$1" ] && [ "${1:0:1}" != "-" ]; then

THIRDPARTY\_IMAGE\_VERSION=$1;shift

fi

fi

fi

# prior to 1.2.0 architecture was determined by uname -m

if [[ $VERSION =~ ^1\.[0-1]\.\* ]]; then

export FABRIC\_TAG=${MARCH}-${VERSION}

export CA\_TAG=${MARCH}-${CA\_VERSION}

export THIRDPARTY\_TAG=${MARCH}-${THIRDPARTY\_IMAGE\_VERSION}

else

# starting with 1.2.0, multi-arch images will be default

: "${CA\_TAG:="$CA\_VERSION"}"

: "${FABRIC\_TAG:="$VERSION"}"

: "${THIRDPARTY\_TAG:="$THIRDPARTY\_IMAGE\_VERSION"}"

fi

BINARY\_FILE=hyperledger-fabric-${ARCH}-${VERSION}.tar.gz

CA\_BINARY\_FILE=hyperledger-fabric-ca-${ARCH}-${CA\_VERSION}.tar.gz

# then parse opts

while getopts "h?dsb" opt; do

case "$opt" in

h|\?)

printHelp

exit 0

;;

d) DOCKER=false

;;

s) SAMPLES=false

;;

b) BINARIES=false

;;

esac

done

if [ "$SAMPLES" == "true" ]; then

echo

echo "Installing hyperledger/fabric-samples repo"

echo

samplesInstall

fi

if [ "$BINARIES" == "true" ]; then

echo

echo "Installing Hyperledger Fabric binaries"

echo

binariesInstall

fi

if [ "$DOCKER" == "true" ]; then

echo

echo "Installing Hyperledger Fabric docker images"

echo

dockerInstall

fi

6). 将fabric-samples的bin目录添加到PATH中

#echo "export PATH=$PATH:$GOPATH/src/github.com/hyperledger/fabric-samples/bin" >> /etc/bashrc

#source /etc/bashrc

7). 将所有的Docker镜像标记为latest

## 开发模式测试智能合约

利用预先生成的orderer和channel artifacts进行示例开发网络。可以立即跳转到智能合约编译和驱动调用的过程。

#cd ../fabric-samples/chaincode-docker-devmode

#docker-compose -f docker-compose-simple.yaml up

如果报错则通过以下命令移除不使用的docker网络

docker network rm $(docker network ls | grep "none" | awk '/ / { print $1 }')

该命令会启动两个容器chaincode（智能合约环境）和cli（用于与智能合约交互）。

创建和连接channel的命令已经嵌入到CLI容器中: #command: /bin/bash -c './script.sh'，因此可以直接跳转到智能合约启动调用

#打开一个新的terminal

#docker exec -it chaincode bash # chaincode为docker-compose-simple.yaml中定义的container\_name: chaincode

#cd sacc

#go build

#CORE\_PEER\_ADDRESS=peer:7052 CORE\_CHAINCODE\_ID\_NAME=mycc:0 ./sacc #启动智能合约；不能拆开执行，否则会提醒chaincode id未提供

目前智能合约只是在容器中启动，还未与任channel有关联，还需要通过实例化才行

#打开一个新的terminal

#docker exec -it cli bash #进入cli容器

#peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0

#peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc

#peer chaincode invoke -n mycc -c '{"Args":["set","a","20"]}' -C myc

#peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc

#peer chaincode list --installed -C myc #查询peer上已安装的智能合约

# peer chaincode list --instantiated -C myc #查询已经在peer所在的channel实例化的智能合约。

# CouchDB

## 根据第十章第1-3以及第4点的(1)-(2)搭建环境。

## 创建、加入通道及部署智能合约

#docker exec -it cli bash #cli为docker-peer.yaml中定义的container\_name: cli

#peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-artifacts/mychannel.tx

#ls #查看生成的mychannel.block文件

#peer channel join -b mychannel.block #通过mychannel.blcok文件加入该channel，以便后续可以安装和、实例化和测试智能合约。

## 安装名为indexcc的智能合约

#peer chaincode install -n indexcc -p github.com/hyperledger/fabric/aberic/chaincode/go/marbles02 -v 1.0

#peer chaincode list --installed -C mychannel #查询peer上已安装的智能合约

#peer chaincode list --instantiated -C mychannel #查询已经在peer所在的channel实例化的智能合约。

## 实例化智能合约

#peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n indexcc -c '{"Args":["asdf","blue","35","bob"]}' -P "OR ('Org1MSP.member')" -v 1.0

## 调用智能合约

初始化一个名为marble1，所有者为tom的对象

#peer chaincode invoke -C mychannel -n indexcc -c '{"Args":["initMarble","marble1","blue","35","tom"]}'

#peer chaincode invoke -C mychannel -n indexcc -c '{"Args":["initMarble","marble2","red","50","tom"]}'

#peer chaincode invoke -C mychannel -n indexcc -c '{"Args":["initMarble","marble3","yellow","70","tom"]}'

## 查询

查询单个对象

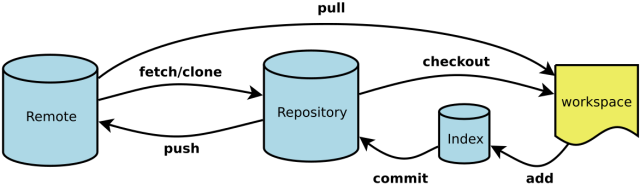
#peer chaincode query -C mychannel -n indexcc -c '{"Args":["readMarble","marble1"]}'

此时应该有三个所有者为tom的对象

#peer chaincode query -C mychannel -n indexcc -c '{"Args":["queryMarblesByOwner","tom"]}'

# Git学习

## git基本知识



**Workspace**：工作区

程序员进行开发改动的地方，是你当前看到的，也是最新的。

平常我们开发就是拷贝远程仓库中的一个分支，基于该分支进行开发。在开发过程中就是对工作区的操作。

**Index/Stage**：暂存区，也叫索引

.git目录下的index文件, 暂存区会记录git add添加文件的相关信息(文件名、大小、timestamp...)，不保存文件实体, 通过id指向每个文件实体。可以使用git status查看暂存区的状态。暂存区标记了你当前工作区中，哪些内容是被git管理的。

当你完成某个需求或功能后需要提交到远程仓库，那么第一步就是通过git add先提交到暂存区，被git管理。

**Repository**：仓库区（本地仓库），也叫存储库

保存了对象被提交 过的各个版本，比起工作区和暂存区的内容，它要更旧一些。

git commit后同步index的目录树到本地仓库，方便从下一步通过git push同步本地仓库与远程仓库的同步。

**Remote**：远程仓库

远程仓库的内容可能被分布在多个地点的处于协作关系的本地仓库修改，因此它可能与本地仓库同步，也可能不同步，但是它的内容是最旧的。

**小结**

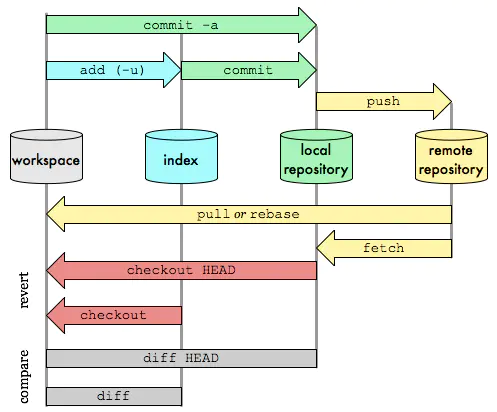
任何对象都是在工作区中诞生和被修改；

任何修改都是从进入index区才开始被版本控制；

只有把修改提交到本地仓库，该修改才能在仓库中留下痕迹；

与协作者分享本地的修改，可以把它们push到远程仓库来共享。

下面这幅图更加直接阐述了四个区域之间的关系，可能有些命令不太清楚，没关系，下部分会详细介绍。

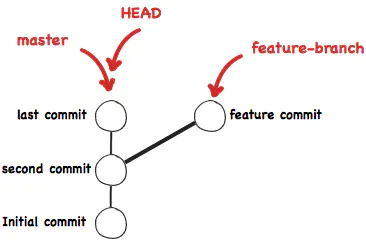


### 有关几个名词解释

**工作区**: 通过git init创建的代码库的所有文件但是不包括.git文件(版本库)

**暂存区**: 通过git add ./\*/\*Xxx/Xxxx\* 添加的修改,都是进入到暂存区了,肉眼不可见 通过 git status 可以看到修改的状态。

**HEAD:** HEAD，它始终指向当前所处分支的最新的提交点。你所处的分支变化了，或者产生了新的提交点，HEAD就会跟着改变。



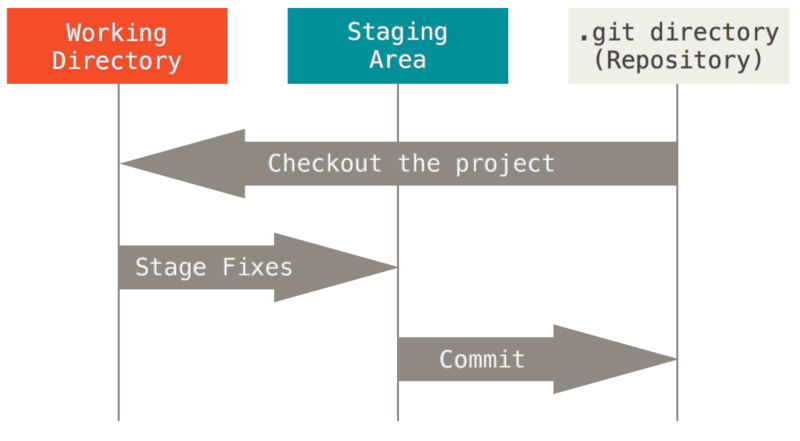
### 什么是修改？

比如你新增了一行，这就是一个修改；删除了一行，也是一个修改；更改了某些字符，也是一个修改；删了一些又加了一些，也是一个修改；甚至创建一个新文件，也算一个修改。

### 三种状态

请注意！如果你希望后面的学习更顺利，记住下面这些关于 Git 的概念。 Git 有三种状态，你的文件可能处于其中之一：**已提交(committed)**、**已修改(modified)**和已暂存(staged)。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。

由此引入 Git 项目的三个工作区域的概念：Git 仓库、工作目录以及暂存区域。工作目录、暂存区域以及 Git 仓库如下图所示：



**Git 仓库目录**是 Git 用来保存项目的元数据和对象数据库的地方。 这是 Git 中最重要的部分，从其它计算机克隆仓库时，拷贝的就是这里的数据。

**工作目录**是对项目的某个版本独立提取出来的内容。 这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供你使用或修改。

**暂存区域**是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。 有时候也被称作‘索引’，不过一般说法还是叫暂存区域。

基本的 Git 工作流程如下：

1. 在工作目录中修改文件。

2. 暂存文件，将文件的快照放入暂存区域。

3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

如果 Git 目录中保存着的特定版本文件，就属于**已提交状态**。 如果作了修改并已放入暂存区域，就属于**已暂存状态**。 如果自上次取出后，作了修改但还没有放到暂存区域，就是**已修改状态**。 在Git 基础一章，你会进一步了解这些状态的细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。

### 远程仓库是什么？

Repository(仓库)包含的内容 - Git的目标是管理一个工程，或者说是一些文件的集合，以跟踪它们的变化。Git使用Repository来存储这些信息。一个仓库主要包含以下内容(也包括其他内容)：

* 许多commit objects
* 到commit objects的指针，叫做heads
* Git的仓库和工程存储在同一个目录下，在一个叫做.git的子目录中。

### 获取 Git 仓库

有两种取得 Git 项目仓库的方法。第一种是从一个服务器克隆一个现有的 Git 仓库。第二种是在现有项目或目录下导入所有文件到 Git 中。

#### 克隆现有的仓库

如果你想获得一份已经存在了的 Git 仓库的拷贝，比如说，想为某个开源项目贡献自己的一份力，这时就要用到 git clone 命令。 如果你对其它的 VCS 系统(比如说Subversion)很熟悉，请留心一下这里所使用的命令是”clone“而不是”checkout“。 这是 Git 区别于其它版本控制系统的一个重要特性，**Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。**当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。如果服务器的磁盘坏掉了，通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库。

$ git clone git@github.com:hyperledger/fabric-sdk-java.git

这会在当前目录下创建一个名为 “fabric-sdk-java” 的目录，并在这个目录下初始化一个 .git 文件夹，从远程仓库拉取下所有数据放入 .git 文件夹，然后从中读取最新版本的文件的拷贝。上面命令执行后，输出结果如下所示。

#### 在现有目录中初始化仓库（参考第本地代码提交到远程仓库那一节）

如果不克隆现有的仓库，而是打算使用 Git 来对现有的项目进行管理。假设有一个项目的目录是：/root/eclipse-workspace/fabric-sdk-java，只需要进入该项目的目录并输入：

$ git ini

该命令将创建一个名为 .git 的子目录，这个子目录含有初始化的 Git 仓库中所有的必须文件，这些文件是 Git 仓库的骨干。 但是，在这个时候，我们仅仅是做了一个初始化的操作，项目里的文件还没有被跟踪。

如果是在一个已经存在文件的文件夹(而不是空文件夹)中初始化 Git 仓库来进行版本控制的话，应该开始跟踪这些文件并提交。可通过 git add 命令来实现对指定文件的跟踪，然后执行 git commit 提交。

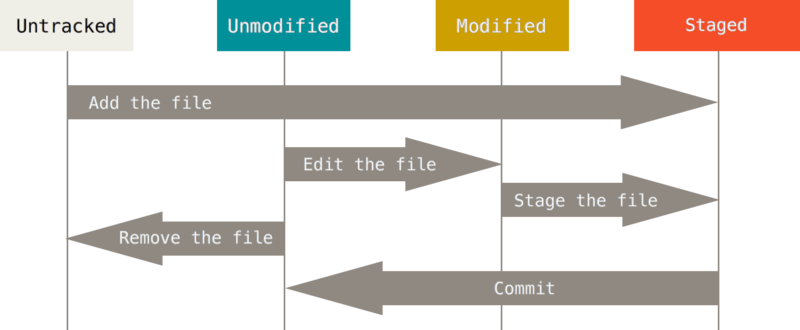
### 更新提交到仓库

#### 记录每次更新到仓库

现在我们手上有了一个真实项目的 Git 仓库，并从这个仓库中取出了所有文件的工作拷贝。 接下来，对这些文件做些修改，在完成了一个阶段的目标之后，提交本次更新到仓库。

工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。 已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。 工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。 初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

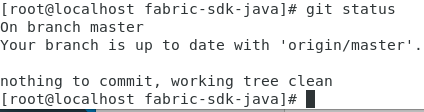
编辑过某些文件之后，由于自上次提交后你对它们做了修改，Git 将它们标记为已修改文件。 我们逐步将这些修改过的文件放入暂存区，然后提交所有暂存了的修改，如此反复。所以使用 Git 时文件的生命周期如下：



#### 检查当前文件状态

要查看哪些文件处于什么状态，可以用 git status 命令。 如果在克隆仓库后立即使用此命令，会看到类似这样的输出：

$ git status



这说明现在你的工作目录相当干净。换句话说，所有**已跟踪文件**在上次提交后都未被更改过。 此外，上面的信息还表明，当前目录下没有出现任何处于未跟踪状态的新文件，否则 Git 会在这里列出来。 最后，该命令还显示了当前所在分支，并告诉你这个分支同远程服务器上对应的分支没有偏离。现在，分支名是 “master”, 这是默认的分支名。

#### 暂存已修改文件

$ git add

#### 状态简览

git status 命令的输出十分详细，但其用语有些繁琐。 如果你使用 git status -s 命令或 git status --short 命令，将得到一种更为紧凑的格式输出。运行 git status -s，状态报告输出如下：

$ git status -s

M README.md

MM Rakefile

A lib/git.rb

M lib/simplegit.rb

?? LICENSE.txt

**新添加的未跟踪文件**前面有 ?? 标记，**新添加到暂存区中的文件**前面有 A 标记，**修改过的文件**前面有 M 标记。 你可能注意到了 M 有两个可以出现的位置，出现在右边的 M 表示该文件被修改了但是还没放入暂存区，出现在靠左边的 M 表示该文件被修改了并放入了暂存区。

例如，上面的状态报告显示： README 文件在工作区被修改了但是还没有将修改后的文件放入暂存区,lib/simplegit.rb 文件被修改了并将修改后的文件放入了暂存区。 而 Rakefile 在工作区被修改并提交到暂存区后又在工作区中被修改了，所以在暂存区和工作区都有该文件被修改了的记录。

#### 忽略文件

一般我们总会有些文件无需纳入 Git 的管理，也不希望它们总出现在未跟踪文件列表。 通常都是些自动生成的文件，比如日志文件，或者编译过程中创建的临时文件等。 在这种情况下，我们可以创建一个名为 .gitignore 的文件，列出要忽略的文件模式。 来看一个实际的例子：

$ cat .gitignore

\*.[oa]

\*~

第一行告诉 Git 忽略所有以 .o 或 .a 结尾的文件。一般这类对象文件和存档文件都是编译过程中出现的。 第二行告诉 Git 忽略所有以波浪符(~)结尾的文件，许多文本编辑软件(比如 Emacs)都用这样的文件名保存副本。 此外，你可能还需要忽略 log，tmp 或者 pid 目录，以及自动生成的文档等等。 要养成一开始就设置好 .gitignore 文件的习惯，以免将来误提交这类无用的文件。

文件 .gitignore 的格式规范如下：

* 所有空行或者以 ＃ 开头的行都会被 Git 忽略。
* 可以使用标准的 glob 模式匹配。
* 匹配模式可以以(/)开头防止递归。
* 匹配模式可以以(/)结尾指定目录。
* 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号(!)取反。

所谓的 glob 模式是指 shell 所使用的简化了的正则表达式。 星号(\*)匹配零个或多个任意字符；[abc]匹配任何一个列在方括号中的字符(这个例子要么匹配一个字符 a，要么匹配一个字符 b，要么匹配一个字符 c)；问号(?)只匹配一个任意字符；如果在方括号中使用短划线分隔两个字符，表示所有在这两个字符范围内的都可以匹配(比如 [0-9] 表示匹配所有 0 到 9 的数字)。 使用两个星号(\*) 表示匹配任意中间目录，比如a/\*\*/z 可以匹配 a/z, a/b/z 或 a/b/c/z等。

提示：GitHub 有一个十分详细的针对数十种项目及编程语言的 .gitignore 文件列表，你可以在 <http://github.com/github/gitignore> 找到它。

#### 查看已暂存和未暂存的修改

如果 git status 命令的输出对于你来说过于模糊，你想知道具体修改了什么地方，可以用 git diff 命令。 稍后我们会详细介绍 git diff，可能通常会用它来回答这两个问题：当前做的哪些更新还没有暂存？ 有哪些更新已经暂存起来准备好了下次提交？ 尽管 git status 已经通过在相应栏下列出文件名的方式回答了这个问题，git diff 将通过文件补丁的格式显示具体哪些行发生了改变。

要查看尚未暂存的文件更新了哪些部分，不加参数直接输入 git diff：

注意，git diff 本身只显示尚未暂存的改动，而不是自上次提交以来所做的所有改动。 所以有时候你一下子暂存了所有更新过的文件后，运行 git diff 后却什么也没有，就是这个原因。

然后用 git diff --cached 查看已经暂存起来的变化：(--staged 和 --cached 是同义词)

注意：git diff 的插件版本,在本教程中，我们使用 git diff 来分析文件差异。 但是，如果你喜欢通过图形化的方式或其它格式输出方式的话，可以使用 git difftool 命令来用 Araxis ，emerge 或 vimdiff 等软件输出 diff 分析结果。 使用 git difftool --tool-help 命令来看你的系统支持哪些 Git Diff 插件。

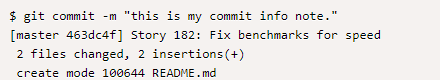
#### 提交更新

现在的暂存区域已经准备妥当可以提交了。 在此之前，请一定要确认还有什么修改过的或新建的文件还没有 git add 过，否则提交的时候不会记录这些还没暂存起来的变化。 这些修改过的文件只保留在本地磁盘。 所以，每次准备提交前，先用 git status 看下，是不是都已暂存起来了，如果没有暂存起来则要先使用命令：git add .将所有文件暂存起来， 然后再运行提交命令 git commit：

$ git status

$ git add .

$ git commit -m " this is my commit info note. "



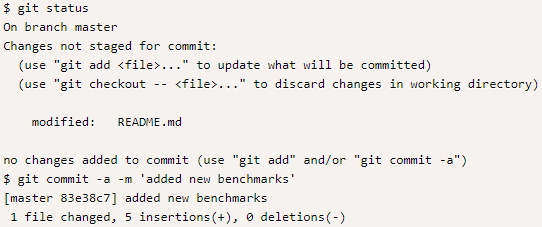
可以看到，提交后它会告诉你，当前是在哪个分支(master)提交的，本次提交的完整 SHA-1 校验和是什么(463dc4f)，以及在本次提交中，有多少文件修订过，多少行添加和删改过。

请记住，提交时记录的是放在暂存区域的快照。任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。 每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

注：git commit --amend -m 使用一次新的commit，替代上一次提交。

#### 跳过使用暂存区域

尽管使用暂存区域的方式可以精心准备要提交的细节，但有时候这么做略显繁琐。 Git 提供了一个跳过使用暂存区域的方式， 只要在提交的时候，给 git commit 加上 -a 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 git add 步骤：

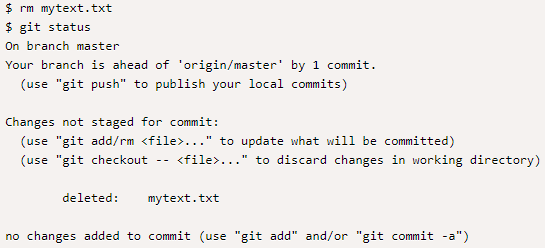


看到了吗？提交之前不再需要 git add 文件“README.md”了。

#### 移除文件

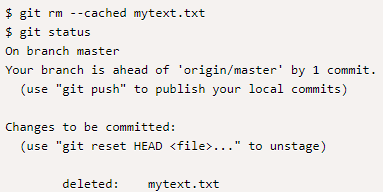
要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除(确切地说，是从暂存区域移除)，然后提交。 可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分(也就是 未暂存清单)看到：



下一次提交时，该文件就不再纳入版本管理了。 如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f(注：即 force 的首字母)。 这是一种安全特性，用于防止误删还没有添加到快照的数据，这样的数据不能被 Git 恢复。

另外一种情况是，我们想把文件从 Git 仓库中删除(亦即从暂存区域移除)，但仍然希望保留在当前工作目录中。 换句话说，你想让文件保留在磁盘，但是并不想让 Git 继续跟踪。 当你忘记添加 .gitignore 文件，不小心把一个很大的日志文件或一堆 .a 这样的编译生成文件添加到暂存区时，这一做法尤其有用。 为达到这一目的，使用 --cached 选项：



git rm 命令后面可以列出文件或者目录的名字，也可以使用 glob 模式。 比方说：

$ git rm log/\\*.log

注意到星号 \* 之前的反斜杠 \， 因为 Git 有它自己的文件模式扩展匹配方式，所以我们不用 shell 来帮忙展开。 此命令删除 log/ 目录下扩展名为 .log 的所有文件。 类似的比如：

$ git rm \\*~

该命令为删除以 ~ 结尾的所有文件。

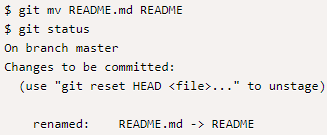
#### 移动文件

不像其它的 VCS 系统，Git 并不显式跟踪文件移动操作。 如果在 Git 中重命名了某个文件，仓库中存储的元数据并不会体现出这是一次改名操作。 不过 Git 非常聪明，它会推断出究竟发生了什么，至于具体是如何做到的，我们稍后再谈。

既然如此，当你看到 Git 的 mv 命令时一定会困惑不已。 要在 Git 中对文件改名，可以这么做：

$ git mv file\_from file\_to

它会恰如预期般正常工作。 实际上，即便此时查看状态信息，也会明白无误地看到关于重命名操作的说明：



其实，运行 git mv 就相当于运行了下面三条命令：

$ mv README.md README

$ git rm README.md

$ git add README

如此分开操作，Git 也会意识到这是一次改名，所以不管何种方式结果都一样。 两者唯一的区别是，mv 是一条命令而另一种方式需要三条命令，直接用 git mv 轻便得多。 不过有时候用其他工具批处理改名的话，要记得在提交前删除老的文件名，再添加新的文件名。

### 查看提交历史

在提交了若干更新，又或者克隆了某个项目之后，你也许想回顾下提交历史。 完成这个任务最简单而又有效的工具是 git log 命令。

$ git log -p -2 # -p用来显示每次提交的内容差异，加上 -2 来仅显示最近两次提交

该选项除了显示基本信息之外，还附带了每次 commit 的变化。 当进行代码审查，或者快速浏览某个搭档提交的 commit 所带来的变化的时候，这个参数就非常有用了。 你也可以为 git log 附带一系列的总结性选项。 比如说，如果你想看到每次提交的简略的统计信息，可以使用 --stat 选项：

$ git log --stat

### 撤消操作

在任何一个阶段，你都有可能想要撤消某些操作。 这里，我们将会学习几个撤消你所做修改的基本工具。 注意，有些撤消操作是不可逆的。 这是在使用 Git 的过程中，会因为操作失误而导致之前的工作丢失的少有的几个地方之一。

有时候我们提交完了才发现漏掉了几个文件没有添加，或者提交信息写错了。 此时，可以运行带有 --amend 选项的提交命令尝试重新提交：

$ git commit --amend

这个命令会将暂存区中的文件提交。 如果自上次提交以来你还未做任何修改(例如，在上次提交后马上执行了此命令)，那么快照会保持不变，而你所修改的只是提交信息。

文本编辑器启动后，可以看到之前的提交信息。 编辑后保存会覆盖原来的提交信息。

例如，提交后发现忘记了暂存某些需要的修改，可以像下面这样操作：

$ git commit -m 'initial commit'

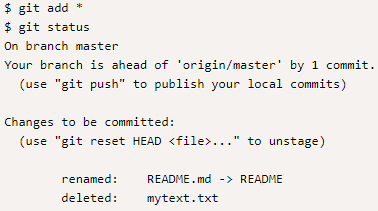
$ git add forgotten\_file

$ git commit --amend

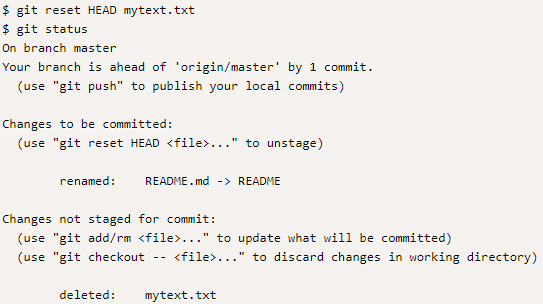
最终你只会有一个提交 - 第二次提交将代替第一次提交的结果。

#### 取消暂存的文件

接下来的两个小节演示如何操作暂存区域与工作目录中已修改的文件。 这些命令在修改文件状态的同时，也会提示如何撤消操作。 例如，你已经修改了两个文件并且想要将它们作为两次独立的修改提交，但是却意外地输入了 git add \* 暂存了它们两个。 如何只取消暂存两个中的一个呢？ git status 命令提示：

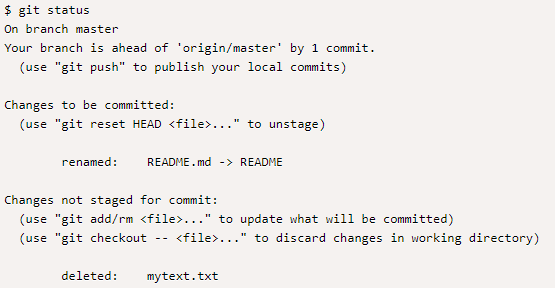


在 “Changes to be committed” 文字正下方，提示使用 git reset HEAD <file>... 来取消暂存。 所以，我们可以这样来取消暂存 mytext.txt 文件：

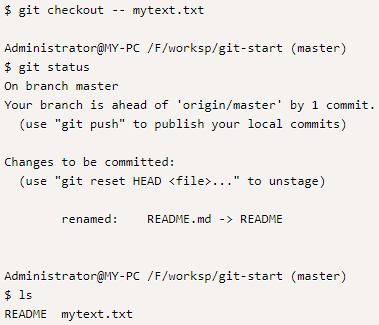


#### 撤消对文件的修改

如果并不想保留对 mytext.txt 文件的修改怎么办？ 该如何方便地撤消修改 - 将它还原成上次提交时的样子(或者刚克隆完的样子，或者刚把它放入工作目录时的样子)？ 幸运的是，git status 也告诉了你应该如何做。 在最后一个例子中，未暂存区域是这样：



它非常清楚地告诉了如何撤消之前所做的修改。让我们来按照提示执行：



可以看到，mytext.txt文件又回来了。

如果仍然想保留对那个文件做出的修改，但是现在仍然需要撤消，我们将会在 Git 分支介绍保存进度与分支；这些通常是更好的做法。

记住，在 Git 中任何已提交的东西几乎总是可以恢复的。甚至那些被删除的分支中的提交或使用 --amend 选项覆盖的提交也可以恢复。然而，任何你未提交的东西丢失后很可能再也找不到了。

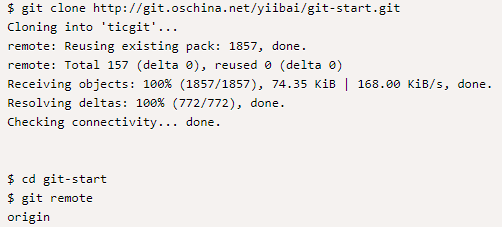
### 远程仓库的使用

前面所有讲解的内容都是一个人“自娱自乐”， Git这东西自己玩也没有多大意思，没有发挥出来Git最牛逼的地方。要使用Git在项目上多人协作那才有意思。

为了能在任意 Git 项目上协作，需要知道如何管理自己的远程仓库。远程仓库是指托管在因特网或其他网络中的你的项目的版本库。可以有好几个远程仓库，通常有些仓库对你只读，有些则可以读写。 与他人协作涉及管理远程仓库以及根据需要推送或拉取数据。 管理远程仓库包括了解如何添加远程仓库、移除无效的远程仓库、管理不同的远程分支并定义它们是否被跟踪等等。 在本节中，我们将介绍一部分远程管理的技能。

#### 查看远程仓库

如果想查看你已经配置的远程仓库服务器，可以运行 git remote 命令。 它会列出你指定的每一个远程服务器的简写。 如果已经克隆了自己的仓库，那么至少应该能看到 origin ，这是 Git 给你克隆的仓库服务器的默认名字：



也可以指定选项 -v，会显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。

$ git remote -v



如果远程仓库不止一个，该命令会将它们全部列出。 例如，与几个协作者合作的，拥有多个远程仓库的仓库看起来像下面这样：



这样可以轻松拉取其中任何一个用户的贡献。 此外，大概还会有某些远程仓库的推送权限，虽然目前还不会在此介绍。

#### 添加远程仓库

我在之前的章节中已经提到并展示了如何添加远程仓库的示例，不过这里将演示如何明确地做到这一点。 运行 git remote add <shortname> <url> 添加一个新的远程 Git 仓库，同时指定一个可以轻松引用的简写：

我在之前的章节中已经提到并展示了如何添加远程仓库的示例，不过这里将演示如何明确地做到这一点。 运行 git remote add <shortname> <url> 添加一个新的远程 Git 仓库，同时指定一个可以轻松引用的简写：$ git remote



现在你可以在命令行中使用字符串 gs 来代替整个 URL。 例如，如果想拉取仓库中有，但你没有的信息，可以运行git fetch gs：



现在 master 分支可以在本地通过 gs/master 访问到 - 可以将它合并到自己的某个分支中，或者如果你想要查看它的话，可以检出一个指向该点的本地分支。

#### 从远程仓库中抓取与拉取

就如刚才所见，从远程仓库中获得数据，可以执行：

$ git fetch [remote-name]

这个命令会访问远程仓库，从中拉取所有还没有的数据。执行完成后，将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

如果使用 clone 命令克隆了一个仓库，命令会自动将其添加为远程仓库并默认以 “origin” 为简写。 所以，git fetch origin 会抓取克隆(或上一次抓取)后新推送的所有工作。 必须注意 git fetch 命令会将数据拉取到本地仓库 - 它并不会自动合并(merge)或修改当前的工作区。当准备好时必须手动将其合并入你的工作区。

如果你有一个分支设置为跟踪一个远程分支，可以使用 git pull 命令来自动的抓取然后合并远程分支到当前分支。这对你来说可能是一个更简单或更舒服的工作流程；默认情况下，git clone 命令会自动设置本地 master 分支跟踪克隆的远程仓库的 master 分支(或不管是什么名字的默认分支)。 运行 git pull 通常会从最初克隆的服务器上抓取数据并自动尝试合并到当前所在的分支。

#### 推送到远程仓库

当想分享你的项目时，必须将其推送到上游。这个命令很简单：git push [remote-name] [branch-name]。 当你想要将 master 分支推送到 origin 服务器时(再次说明，克隆时通常会自动帮你设置好那两个名字)，那么运行这个命令就可以将所做的备份到服务器：

$ git push origin master

只有当你有所克隆服务器的写入权限，并且之前没有人推送过时，这条命令才能生效。 当你和其他人在同一时间克隆，他们先推送到上游然后你再推送到上游，你的推送就会毫无疑问地被拒绝。 你必须先将他们的工作拉取下来并将其合并进你的工作后才能推送。

现在我们要把前所有添加和修改的内容添加到远程仓库，以便其协同的开发人员也可以获取到我们提交的内容。

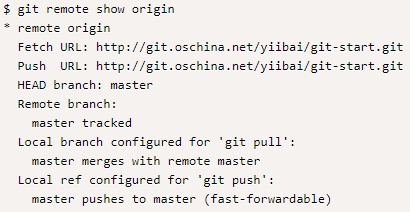
#### 查看远程仓库

如果想要查看某一个远程仓库的更多信息，可以使用 git remote show [remote-name] 命令。 如果想以一个特定的缩写名运行这个命令，例如 origin，会得到像下面类似的信息：

它同样会列出远程仓库的 URL 与跟踪分支的信息。 这些信息非常有用，它告诉你正处于 master 分支，并且如果运行 git pull，就会抓取所有的远程引用，然后将远程 master 分支合并到本地 master 分支。 它也会列出拉取到的所有远程引用。

这是一个经常遇到的简单例子。 如果你是 Git 的重度使用者，那么还可以通过 git remote show 看到更多的信息。

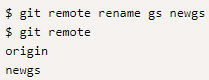
$ git remote show origin



这个命令列出了当你在特定的分支上执行 git push 会自动地推送到哪一个远程分支。 它也同样地列出了哪些远程分支不在你的本地，哪些远程分支已经从服务器上移除了，还有当你执行 git pull 时哪些分支会自动合并。

#### 远程仓库的移除与重命名

如果想要重命名引用的名字可以运行 git remote rename 去修改一个远程仓库的简写名。 例如，想要将 gs 重命名为 newgs，可以用 git remote rename 这样做：



值得注意的是这同样也会修改你的远程分支名字。 那些过去引用 gs/master 的现在会引用 newgs/master。

如果因为一些原因想要移除一个远程仓库 - 你已经从服务器上搬走了或不再想使用某一个特定的镜像了，又或者某一个贡献者不再贡献了 - 可以使用 git remote rm ：



## 升级git版本

由于centos7.4自带的git版本为1.8.3，建议升级到高版本。

### 卸载低版本的git

yum remove git

### 安装依赖包

$ yum -y install curl-devel expat-devel gettext-devel openssl-devel zlib-devel gcc perl-ExtUtils-MakeMaker autoconf

### 下载压缩包

wget <https://www.kernel.org/pub/software/scm/git/git-2.13.2.tar.gz>

### 解压安装

$ tar -zxvf git-2.13.2.tar.gz

$ cd git-2.13.2

$ ./configure --prefix=/usr/local/git all

$ make && make install

### 加入环境变量

$ gedit /etc/bashrc

最后一行加入

$ export PATH=$PATH:/usr/local/git/bin

$ source /etc/bashrc

## 初始化安装

### 前期检查

whereis git; which git等命令来检查git的安装位置。

使用命令：yum remove git卸载git

### 方式一：离线安装（推荐）

#### 下载

最新源码包下载地址：<https://git-scm.com/download>

$ wget <https://www.kernel.org/pub/software/scm/git/git-2.23.0.tar.gz>

#### 安装依赖项

$ yum -y install curl-devel expat-devel gettext-devel openssl-devel zlib-devel gcc perl-ExtUtils-MakeMaker autoconf

#### 安装

$ tar -zxf git-2.23.0.tar.gz

$ cd git-2.23.0

$ ./configure --prefix=/usr/local/git/

$ make & make install

#### 检查

$ git -version

将git指令添加到bash中

#### 将git指令添加到bash中

$ gedit /etc/profile

在最后一行加入

$ export PATH=$PATH:/usr/local/git/bin

让该配置文件立即生效

source /etc/profile

### 方式二：在线安装（不推荐，安装版本较低）

$ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel

$ yum -y install git-core

$ git --version

# yum remove git ## 移除原来的版本

## 常用Git命令



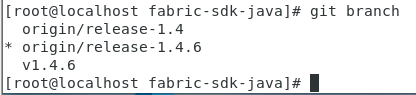
### $ git clone --branch [tags标签] [git地址] 或者 git clone -b [tags标签] [git地址]

注意这个<branchname>用精确Tags号比较好，比如：v1.4.6

比如: $ git clone --branch v1.4.6 git@github.com:hyperledger/fabric-sdk-java.git

$ git status -s(-s 表示获取简短的结构输出) #查看当前项目的当前状态"M":已修改；"??":未添加，"A":已添加到缓存，"AM":这个文件在我们将它添加到缓存之后又有改动。

$ git branch # 没有参数时，git branch 会列出你在本地的分支。



此例的意思就是，我们本地有三个分支，并且当前分支是origin/release-1.4.6。当你执行 git init 的时候，默认情况下 Git 就会为你创建 master 分支。如果我们要手动创建一个分支。执行 git branch (branchname) 即可。

### git branch



#### $ git branch #列出所有本地分支

#### $ git branch -r #列出所有远程分支

#### $ git branch -a #列出所有本地分支（绿色表示）和远程分支（红色表示）

#### git branch -vv #查看本地分支所关联的远程分支

#### git branch -m old\_branch new\_branch # Rename branch locally

#### git push origin :old\_branch # Delete the old branch of remote depository

#### git push --set-upstream origin new\_branch 或者git push -u origin new\_branch

# Push the new branch, set local branch to track the new remote$ git branch <branchname> # 新建一个分支，但依然停留在当前分支

说明：以上d)、e)、f)、g)可以删除远程和本地分支。

#### $ git branch --set-upstream master origin/next

Git也允许手动建立追踪关系。上面命令指定master分支追踪origin/next分支。

#### $ git branch -d <branchname> #删除本地分支

#### $ git push origin --delete 删除远程分支

### git checkout

git checkout命令用于切换分支或恢复工作树文件。git checkout是git最常用的命令之一，同时也是一个很危险的命令，因为这条命令会重写工作区。

更新工作树中的文件以匹配索引或指定树中的版本。如果没有给出路径 - git checkout还会更新HEAD指针，将指定的分支设置为当前分支。

HEAD指向的是现在使用中的分支的最后一次更新。通常默认指向master分支的最后一次更新。通过移动HEAD，就可以变更使用的分支。

#### git checkout [<commit>] [--] <paths>

用于拿暂存区的文件覆盖工作区的文件，或者用指定提交中的文件覆盖暂存区和工作区中对应的文件。

第一种用法在命令中包含路径 <paths>。为了避免路径<paths>和引用（或者提交）<commit> 同名而冲突，可以在<paths>前用两个连续的短线减号作为分隔。

<commit> 是可选项，如果省略则相当于从暂存区（index）检出。这和 git reset 重置命令（例如 git reset HEAD <file>）大不相同：重置的默认值是 HEAD，而检出的默认值是暂存区。因此重置一般用于重置暂存区（除非使用--hard参数，否则不重置工作区），而检出命令主要是覆盖工作区（如果<commit>不省略，也会替换暂存区中相应的文件）。

第一种用法（包含了路径 <paths> 的用法）不会改变 HEAD 头指针，主要是用于拿指定版本的文件覆盖工作区中对应的文件。如果省略<commit>，则会拿暂存区的文件覆盖工作区的文件，否则用指定提交中的文件覆盖暂存区和工作区中对应的文件。

#### git checkout <branch> #切换到指定分支，并更新工作区

用于切换本地分支。当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容， 所以多个分支不需要多个目录。

第二种用法（不使用路径<paths>的用法）则会改变 HEAD 头指针。之所以后面的参数写作<branch>，是因为只有 HEAD 切换到一个分支才可以对提交进行跟踪，否则仍然会进入“分离头指针”（Detached HEAD）的状态。在“分离头指针”状态下的提交不能被引用关联到而可能会丢失。所以用法二最主要的作用就是切换分支。

#### git checkout -b <new\_branch> [<start\_point>] #新建一个分支，并切换到该分支

在本地仓库创建新分支并立即切换到该分支下。注意，该操作不会更新工作区！！！，且会导致'detached HEAD' state。

-b 选项表示创建新分支。如果指定了 <start\_point>（可以为远程仓库的release或者tag号） ，那么新的分支将指向<start\_point> ，否则新的分支指向当前的 HEAD. 新分支和我们熟悉的 master 分支并没有什么实质的不同，都是在refs/heads命名空间下的引用。第三种用法用于分支的创建并切换。

相当于 $git branch <new\_branch> & $ git checkout <new\_branch> [<start\_point>]

**深入**

要想更深入的了解checkout，我们需要了解checkout的作用机制。该命令的主要关联目标其实是.git 文件夹下的HEAD文件，我们可以查看工程下面的.git文件夹中的HEAD文件，该文件记录了当前HEAD的信息：



可以看到当前HEAD文件指向了refs/heads路径下的master文件，该文件记录了master分支最近的一次commit id,说明当前HEAD指向了master分支。如果我们将当前分支切换到newBranch 分支，我们再看HEAD文件：



可以看到HEAD文件内容指向了newBranch分支

### git fetch

一旦远程主机的版本库有了更新（Git术语叫做commit），需要将这些更新取回本地，这时就要用到git fetch命令。

#### $ git fetch <远程主机名>

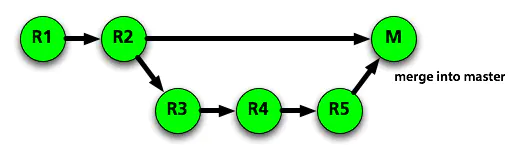
上面命令将某个远程主机的更新，全部取回本地。

#### $ git fetch <远程主机名> <分支名>

默认情况下，git fetch取回所有分支（branch）的更新。如果只想取回特定分支的更新，可以指定分支名。比如，取回origin主机的master分支。

所取回的更新，在本地主机上要用"远程主机名/分支名"的形式读取。比如origin主机的master，就要用origin/master读取。

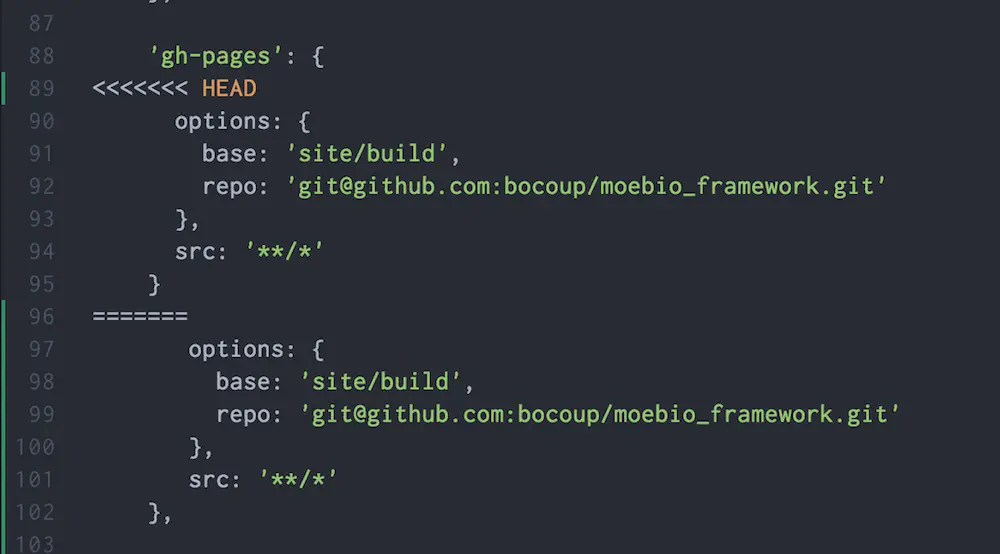
### $ git merge < branchname>#合并<branchname>分支到当前分支上。



merge命令把不同的分支合并起来。如上图，在实际开放中，我们可能从master分支中切出一个分支，然后进行开发完成需求，中间经过R3,R4,R5的commit记录，最后开发完成需要合入master中，这便用到了merge。

**git merge之前先拉一下远程仓库最新代码**

一般在merge之后，会出现conflict，需要针对冲突情况，手动解除冲突。主要是因为两个用户修改了同一文件的同一块区域。如下图所示，需要手动解除。



#### 开发分支（dev）上的代码达到上线的标准后，要合并到 master 分支

$ git checkout dev

$ git pull

$ git checkout master

$ git merge dev

$ git push -u origin master

#### 当master代码改动了，需要更新开发分支（dev）上的代码

$ git checkout master

$ git pull

$ git checkout dev

$ git merge master

$ git push -u origin dev

### $ git pull <远程主机名> <远程分支名>:<本地分支名>

取回远程主机某个分支的更新，再与本地的指定分支合并。

将远程存储库中的更改合并到当前分支中。在默认模式下，git pull是git fetch后跟git merge FETCH\_HEAD的缩写。

更准确地说，git pull使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前分支中。

$ git pull <远程主机名> <远程分支名>:<本地分支名>

比如，要取回origin主机的next分支，与本地的master分支合并，需要写成下面这样:

$ git pull origin next:master

如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略。上面命令可以简写为：

$ git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

$ git fetch origin next

$ git merge origin/next

### $ git push <远程主机名> <本地分支名>:<远程分支名>

上传本地仓库分支到远程仓库分支，实现同步。

#### $ git push origin #

上面命令表示，将当前分支推送到origin主机的对应分支。如果当前分支只有一个追踪分支，那么主机名都可以省略：$ git push

如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机：

#### $ git push -u origin <branchname>

上面命令将本地的<branchname>分支推送到origin主机对应的<branchname>分支，同时-u表示指定origin为默认主机，后面就可以不加任何参数使用git push了。

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。此外，还有一种matching方式，会推送所有有对应的远程分支的本地分支。Git 2.0版本之前，默认采用matching方法，现在改为默认采用simple方式。

#### $ git push origin :<远程分支名> # 删除远程分支，注意冒号前有空格

#### git push --force强行推送当前分支到远程仓库，即使有冲突

#### git push --all推送所有分支到远程仓库

### $ git log显示当前分支的版本历史

### $ git status显示有变更的文件

### $ git diff #显示暂存区和工作区的详细差异

执行 git diff 来查看执行 git status 的结果的详细信息。git diff 命令显示已写入缓存与已修改但尚未写入缓存的改动的区别。git diff 有两个主要的应用场景；尚未缓存的改动：git diff；查看已缓存的改动： git diff –cached；查看已缓存的与未缓存的所有改动：git diff HEAD；显示摘要而非整个 diff：git diff –stat

### git diff HEAD显示工作区与当前分支最新commit之间的差异

## git clone、git pull和git fetch的用法及区别

### git fetch 更新远程代码到本地仓库

理解 fetch 的关键, 是理解 FETCH\_HEAD，FETCH\_HEAD指的是: 某个branch在服务器上的最新状态’。这个列表保存在 .Git/FETCH\_HEAD 文件中, 其中每一行对应于远程服务器的一个分支。

当前分支指向的FETCH\_HEAD, 就是这个文件第一行对应的那个分支.

一般来说, 存在两种情况:

* 如果没有显式的指定远程分支, 则远程分支的master将作为默认的FETCH\_HEAD
* 如果指定了远程分支, 就将这个远程分支作为FETCH\_HEAD

**git fetch更新本地仓库（版本库）的两种用法：**

**# 方法一**

$ git remote show [remote-name] # 查看远程仓库(e.g., origin)的信息

$ git fetch origin master #获取远程origin仓库master分支的最新版本到本地版本库，不会自动merge

$ git log -p master origin/master #比较本地的仓库和远程仓库的区别

$ git merge origin/master #把远程master分支的代码合并到当前分支

**# 方法二（得先checkout到master分支下）**

$ git fetch origin master:temp #从远程的origin仓库的master分支下载到本地并新建一个分支temp

$ git diff temp #比较master分支和temp分支的不同

$ git merge temp #合并temp分支到当前master分支

$ git branch -d temp #删除temp

**# 方法三**

**#** git pull origin master

上述命令其实相当于git fetch 和 git merge。在实际使用中，git fetch更安全一些，因为在merge前，我们可以查看更新情况，然后再决定是否合并。

### git pull

git pull是拉取远程分支更新到本地仓库的操作。比如远程仓库里的学习资料有了新内容，需要把新内容下载下来的时候，就可以使用git pull命令。事实上，git pull是相当于从远程仓库获取最新版本，然后再与本地分支merge（合并）。

　　即：git pull = git fetch + git merge

注：git fetch不会进行合并，执行后需要手动执行git merge合并，而git pull拉取远程分之后直接与本地分支进行合并。更准确地说，git pull是使用给定的参数运行git fetch，并调用git merge将检索到的分支头合并到当前分支中。

**用法：**

$ git pull <远程主机名> <远程分支名>:<本地分支名>

$ git pull <远程主机名> <远程tag名>:<本地tag名>

举例：将远程主机origin的master分支拉取过来，与本地的branchtest分支合并。

$ git pull origin master:branchtest

如果将冒号和后面的branchtest去掉，则表示将远程origin仓库的master分支拉取下来与本地**当前分支**合并。

以上的git pull操作如果用git fetch来表示：

$ git fetch origin master:brantest

$ git merge brantest

**相比起来，git fetch更安全也更符合实际要求**，因为可以在merge前，我们可以查看更新情况，根据实际情况再决定是否合并。

### git clone

git clone顾名思义就是将其他仓库克隆到本地，包括被clone仓库的版本变化。举个例子，你当前目录比方说是在e:/course/中，此时若想下载远程仓库，本地无需git init,直接git clone url（url是你远程仓库的地址，直接复制就可以了）。执行git clone等待clone结束，e:/course/目录下自动会有一个.git的隐藏文件夹（如果看不见，请尝试设置隐藏文件夹可见），因为是clone来的，所以.git文件夹里存放着与远程仓库一模一样的版本库记录。clone操作是一个从无到有的克隆操作，再次强调不需要git init初始化。

$ git clone <版本库的网址> <本地目录名>

## git remote update、git fetch、git pull的区别

git remote update will update all of your branches set to track remote ones（更新版本库中的所有分支）, but not merge any changes in. git fetch will update only the branch you're on（只更新版本库HEAD指针所在的当前分支）, but not merge any changes in. git pull will update and merge any remote changes of the current branch you're on。

## git本地版本回退与远端版本回退(回滚)

### 预备知识：

一个commit对应这一个版本，有一个commit id，40位的16进制数字，通过SHA1计算得到，不同的文件计算出来的SHA1值不同(有很小的几率相同，可忽略)，这样每一个提交都有其独特的id。每提交一个新版本，实际上Git就会把它们自动串成一条时间线。

在Git中，HEAD表示当前版本，也就是e620a6ff0940a8dff…，HEAD^表示上一个版本，HEAD^^表示上上一个版本，往上100个版本可以写成HEAD加连续100个^，也可以写成：HEAD~100。

### 从远程版本回退

#### git log：该命令显示从最近到最远的提交日志。

#### git log --pretty=oneline # 将只会显示提交的commit id号和对应的注释。

$ git log --pretty=oneline --abbrev-commit #只显示commit id的缩写

#### git reset --hard commit\_id 或则是 git reset --hard HEAD^

hard选项，表示彻底将**工作区、暂存区和版本库**记录恢复到指定的版本库

使用git reset --hard 进行版本回退之后，在本地查看**README.md**，里面已经变为版本2对应的内容了。

### 从本地版本回退

如果你在本地做了错误提交，那么回退版本的方法很简单

#### $ git reflog #先找到要回退的版本的commit id：

#### $ git reset --hard a7e1d279

a7e1d279就是你要回退的版本的commit id的前面几位。

### 如果你的错误提交已经推送到自己的远程分支了，那么就需要回滚远程分支了。

#### 首先要回退本地分支：

$ git reflog

$ git reset --hard Obfafd

#### 紧接着强制推送到远程分支：

$ git push -f origin master # 这里假设只有一个master分支

origin就是一个名字，它是在你clone一个托管在Github上代码库时，git为你默认创建的指向这个远程代码库的标签，origin指向的是repository，master只是这个repository中默认创建的第一个branch。

**注意：本地分支回滚后，版本将落后远程分支，必须使用强制推送覆盖远程分支，否则无法推送到远程分支。（貌似还需要远程仓库设置一下允许强制覆盖，自行百度）**

## git强制覆盖本地代码（与git远程仓库保持一致）

git强制覆盖：

    git fetch --all #拉取所有更新，不同步；

    git reset --hard origin/master #本地代码同步线上master分支的最新版本(会覆盖本地所有与远程仓库上同名的文件)；

## tag操作

tag是对历史提交的一个id的引用，如果理解这句话就明白了tag的含义。

### git查看tag

$ git tag #查看所有tag

$ git tag -l 'v1.4.\*' #查看v1.4系列tag

### git show <tagname>查看tag信息

$ git show v1.4.6

### git切换到tag

git clone整个仓库后，使用以下命令就可以取得对应tag的代码:

#### $ git checkout tag\_name (e.g., git checkout v1.4.0 # 会更新代码)

此时git可能会提示你当前处于“detached HEAD” 状态。因为tag相当于一个快照，不能修改它的代码。需要在tag代码基础上做修改，并创建一个分支。

该步骤可能会出先如下错误：



可以执行：$ git remote update将远程所有分支更新到**本地仓库，**然后再checkout

$ git fetch 和g $git pull貌似也行，没测试过！！！详情请参考“git remote update、git fetch、git pull的区别”章节。

#### $ git checkout -b branch\_name tag\_name

上面命令的意思切换到指定的tag（该tag\_name在远程仓库必须存在)，并创建一个新的本地分支，命名为（branch\_name）。以fabric的版本为例：

$ git checkout -b tag-v1.4.0 v1.4.0

其中tag-v1.4.0为新的本地分支名，v1.4.0为github上fabric项目自身的tag命名。

#### $ git checkout -B branch\_name tag\_name

这个命令，可以强制创建新的分支，为什么加-B呢？如果当前仓库中，已经存在一个跟你新建分支同名的分支，那么使用普通的git checkout -b 这个命令，是会报错的，且同名分支无法创建。如果使用-B参数，那么就可以强制创建新的分支，并会覆盖掉原来的分支。

git checkout -B tag-v1.4.0 v1.4.0 强制创建一个基于指定的tag的分支。

### 给本地仓库当前分支打轻量级tag标签

注意：标签是打在最新提交的commit上的。

#### 首先，切换到需要打标签的分支上：

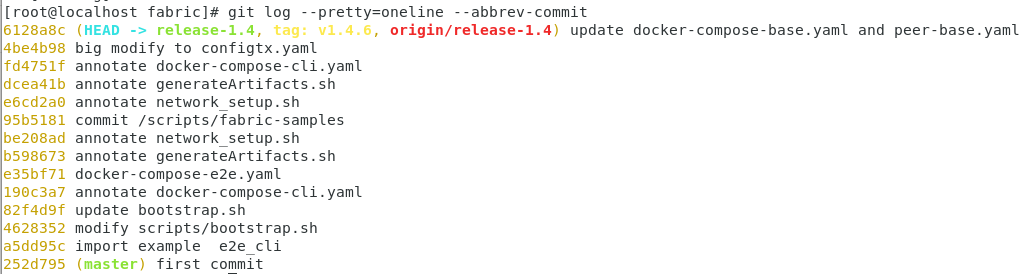
#### 然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.4.6

### 根据历史记录，给之前的commit打标签

默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？方法是找到历史提交的commit id，然后打上就可以了：

$ git log --pretty=oneline --abbrev-commit



比如要对first commit这次提交打标签，它对应的commit id是252d795，敲入命令：

$ git tag v1.4.6 252d795

### 给本地仓库的当前分支的commit打带注释的tag标签

创建带有说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v1.4.6 -m " first commit to master " 252d795

用命令git show <tagname>可以看到说明文字：

$ git show v1.4.6

### 把本地仓库分支tag推送到远程服务器

到这里我们就给相关分支的某些提交版本添加了tag，但是git tag命令是对本地仓库分支加的标签，为了能把标签同步到远程服务器，我们需要做如下操作:

默认情况下，git push并不会把tag标签传送到远端服务器上，只有通过显式命令才能分享标签到远端仓库。

#### 存在的问题

**问题1：执行完$** git push origin v1.4.0后远程tag没有更新，这是因为tag是针对最新提交的commit，虽然commit了，但是新的commit没有跟老的tag关联的原因。

**解决办法：**删除本地和远程仓库原来的tag，重新给修改后的内容打tag，然后push到远程仓库。

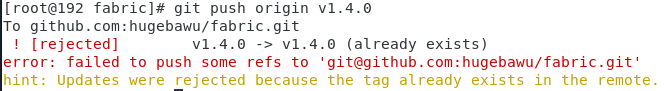
1. $ git checkout tag-v1.4.0

2. $ git tag -d <tag name> # 删除本地tag

git tag -d v1.4.0

3. 删除远程tag

直接push会导致以下错误：



所以先删除远程tag。

$ git push origin :refs/tags/v1.4.0

4. $ git push origin v1.4.0

**问题2**：执行操作没有报错，但是远程仓库就是没有更新。

看起来没有报错，但是却不是正确的返回结果。这是因为创建的新的分支tag-v1.4.6分支没有在release-1.4上，所以将其需要merge然后push到release-1.4上。

切换到release-1.4分支，合并新的分支，然后在push就没有问题了，指令如下：

$ git checkout release-1.4

$ git merge tag-v1.4.0

$ git push -u origin release-1.4

**详细原理解释搜索：“Git 分支 - 分支的新建与合并”**

#### push单个tag，命令格式为：

$ git push origin [tagname]

e.g., $ git push origin v1.4.0 # 将本地当前分支的标签版本v1.4.0推送到远端服务器

#### push所有tag，命令格式为：

$ git push [origin] --tags

1) $ git push origin --tags 或者：

2)$ git push --tags # 当远程有多个服务的时候远程服务名称是必须的，而如果远程只有一个远程服务则远程服务名称可以省略。

## 本地代码提交到远程仓库

将本地代码上传到远程仓库的时候，打开命令行窗口，进入到本地代码的文件夹。

### $ #清理旧的.git文件夹

rm -rf .git;

注意：其它.gitattributes和.gitignore等.git开头的文件夹别清理，非常有用。

### $ git init （生成.git文件夹的时候才需要）

#Git 使用 git init 命令来初始化一个 Git 仓库，Git 的很多命令都需要在 Git 的仓库中运行，所以 git init 是使用 Git 的第一个命令。

在执行完成 git init 命令后，Git 仓库会生成一个 .git 目录，该目录包含了资源的所有元数据，其他的项目目录保持不变（不像 SVN 会在每个子目录生成 .svn 目录，Git 只在仓库的根目录生成 .git 目录）。

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

### $ echo "# fabric" >> README.md #新建README.md文件并输入

### $ git add README.md

但这一步是添加README.md文件，初始化git，这样在本地文件夹下会生成.git文件夹。具体这个文件是做什么的，可以自行百度，不过一句话概括的话，是将本地仓库与远程仓库关联起来。

### $ git add . #将当前项目的所有文件添加到缓存区 #

### $ git commit -m "first commit"

将缓存区内容添加到仓库中

（提示出错则需要配置用户名和邮箱地址 $ git config --global user.name hugebawu

$ git config --global user.email drbjhu@163.com）

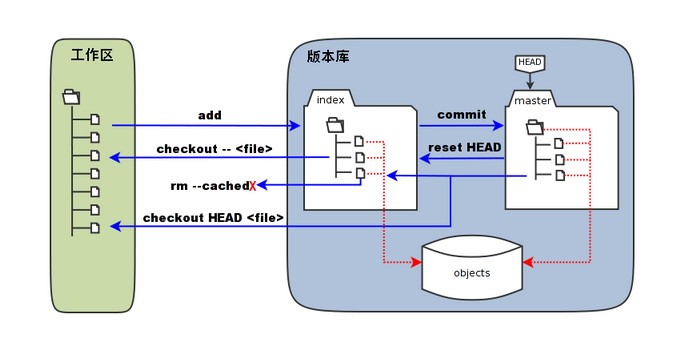
到commit这一步，只是完成了本地提交。这里为什么有一个add一个commit的过程，有必要解释一下。在git中有一个暂存区的概念，add就是将代码从工作区提交到暂存区；commit的过程就是将代码从暂存区提交到**版本库**。git add . 是提交全部文件到暂存区，git commit -m后面相当于给本次提交加一个注释。

这里解释一下git为什么有暂存区的概念，这是和SVN所不同的。git是分为**工作区**和版本库的，工作区就是我们本地看到的工作目录，版本库就是你工作目录中的**.git文件**。

工作区：就是你在电脑里能看到的目录。

暂存区：英文叫**stage, 或index**。一般存放在 ".git目录下" 下的index文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）。

版本库：工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。



图中左侧为工作区，右侧为版本库。在版本库中标记为 "index" 的区域是暂存区（stage, index），标记为 "master" 的是 master 分支所代表的目录树。

图中我们可以看出此时 "HEAD" 实际是指向 master 分支的一个"游标"。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。

图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下，里面包含了创建的各种对象及内容。

当对工作区修改（或新增）的文件执行 "git add" 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。

当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。

当执行 "git reset HEAD" 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。

当执行 "git rm --cached <file>" 命令时，会直接从暂存区删除文件，工作区则不做出改变。

当执行 "git checkout ." 或者 "git checkout -- <file>" 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。

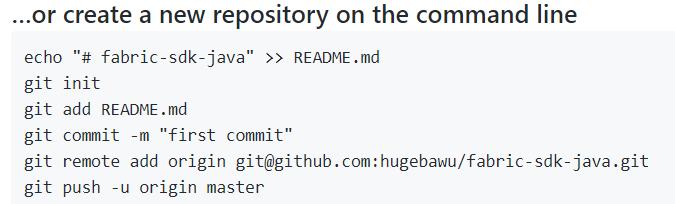
当执行 "git checkout HEAD ." 或者 "git checkout HEAD <file>" 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

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

版本库中存放了很多东西，比如stage,就是我们上面提到的暂存区，还有自动创建的master分支，还有指向master的head指针…….我们不能手动修改版本库内的任何数据，否则可能会造成不可预期的错误。所以我们上面，将工作区的代码提交到版本库是分为两个步骤的，第一步是通过add命令将文件添加到暂存区stage中；第二步将暂存区中的数据提交到版本库中，用commit命令，其实就是将stage中的内容提交到当前分支。我们在创建git版本库时，会为我们自动创建一个master主分支(#git branch -a来查看)，当然我们还可以手动创建其他分支，当前分支是什么，我们commit的时候就将stage的内容提交到该分支上。

### 先在github个人网页手动新建fabric-sdk-java仓库，然后执行命令（第一次执行才需要）：

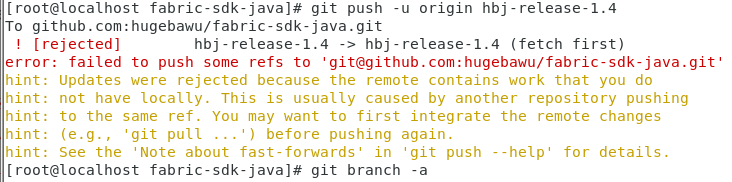
**$ git remote add origin git@github.com:hugebawu/fabric-sdk-java.git**



### $ git push -u origin <branchname 比如”master”, ”release-1.1” >

说明：将本地仓库push到远程仓库 #最后一个字段填本地想要提交的分支名

**注意1）**这一步需要保证远程分支具有的工作本地都有，否则会出现如下错误：



所以需要先执行pull操作，如: git pull origin hbj-release-1.4

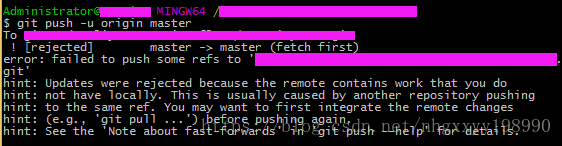
**注意2）**这一步必须保证branchname分支在本地存在，否则报错！！！：



可以通过git checkout -b <new branch name>在本地仓库创建新分支并切换。

**注意3）：**由于新建的远程仓库是空的，所以要加上-u这个参数，等远程仓库里面有了内容之后，下次再从本地库上传内容的时候不需要加-u。

最后将版本库中的代码提交到远程分支上去。到这里，将本地代码提交到远程仓库的步骤就全部完成了。我在提交的时候也遇到一些问题，拿出来我们一起分享下



出现错误的原因是git服务器中的README.md文件不在本地代码目录中，可以通过如下命令进行代码合并

git pull --rebase origin master

执行上面的代码就可以使本地和git服务器同步了。其实如果有时间，可以好好研究一下这个.git和README.md文件。.git文件里面存放的内容上面简单介绍了一下，README.md文件要研究起来内容还是蛮多的，这里推荐一个博客感觉介绍的挺好的，感兴趣的源儿们可以看一下原来Github上的README.md文件这么有意思——Markdown语言详解

好了 这篇博客就介绍到这里，顺便推荐一下廖雪峰关于git的文章，介绍的非常好，非常适合初学研究，整体看下可以形成一个宏观的知识体系。好的，先这样，后续再更新。

# Maven学习

## Maven安装

### 下载maven

$ wget <https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.5.0/apache-maven-3.5.0-bin.tar.gz>

$ wget <http://apache.mirrors.pair.com/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz>

$ tar -zxvf apache-maven-3.6.3-bin.tar.gz -C /usr/local

### 配置环境变量

$ gedit /etc/bashrc

export MAVEN\_HOME=/usr/local/apache-maven-3.6.3

export PATH=$PATH:$MAVEN\_HOME/bin

$ source /etc/bashrc

### 是否安装成功

mvn -v

## Maven的使用

进入项目根目录

### #mvn compile

compile是maven工程的编译命令，作用是将src/main/java下的文件编译为class文件输出到target目录下

### #mvn test

这一步的正确执行需要按照搭建好fabric环境，相关的容器得跑起来。

test是maven工程的测试命令，会执行src/test/java下的单元测试类。

### #mvn clean

clean是maven工程的清理命令，执行 clean会删除根目录下生成的target目录的内容。 而project->clean选项将工程中的.class文件删除，同时重新编译(build)工程，类似于jbuild中的rebuild。

### #mvn package

package是maven工程的打包命令，对于java工程执行package打成jar包，对于web工程打成war包。

### $mvn install

最大的作用就是将项目本身编译并打包，并且其他项目引用本项目的jar包时不用去私服上下载jar包，直接从本地就可以拿到刚刚编译打包好的项目的jar包，很灵活，避免每次都需要重新往私服发布jar包的痛苦。

# Linux下Eclipse安装与配置

## Eclips安装到linux

下载地址：

<https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2019-12/R/eclipse-jee-2019-12-R-linux-gtk-x86_64.tar.gz>

### 解压到/usr/local目录

#tar -zvxf eclipse-jee-2019-09-R-linux-gtk-x86\_64.tar.gz -C **/usr/local**

### 创建软链接

使用符号链接eclipse目录，将eclipse 启动命令软链接到/usr/bin/eclipse。 eclipse与/usr之间有空格 命令的意思是，在/usr/bin/eclipse 目录下创建一个同步链接，而源文件在/usr/local/eclipse/eclipse目录下

#ln -s /usr/local/eclipse/eclipse /usr/bin/eclipse

### 在系统菜单栏创建eclipse启动选项

#gedit /usr/share/applications/eclipse.desktop

编辑器中输入

[Desktop Entry]

Encoding=UTF-8

Name=Eclipse

Comment=Eclipse 2019

Exec=/usr/bin/eclipse

Icon=/usr/local/eclipse/icon.xpm

Categories=Application;Development;Java;IDE

Version=1.0

Type=Application

Terminal=0

### 创建eclipse桌面快捷键

进入/usr/share/applications，把 对应的 eclipse.desktop 文件复制到桌面就OK！

#cp /usr/share/applications/eclipse.desktop /root/Desktop/

### eclipse启动不了办法

#kill -9 `ps -ef|grep eclipse|awk '{print $2}'` #杀死进程

#rm -rf /usr/local/eclipse

#rm /usr/bin/eclipse

重新从第(1)开始

### Eclipse的环境配置

参考链接: <https://blog.csdn.net/qq_39135287/article/details/82080373>

#### 安装Chekcstyle

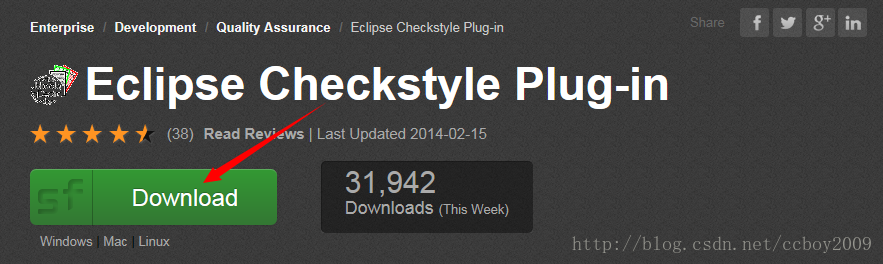
CheckStyle是SourceForge下的一个项目，提供了一个帮助JAVA开发人员遵守某些编码规范的工具。它能够自动化代码规范检查过程，从而使得开发人员从这项重要但枯燥的任务中解脱出来。它可以根据设置好的编码规则来检查代码。比如符合规范的变量命名，方法体的最大行数，重复代码检查等等。

如果你的项目经理开会时说：“我希望我们写出来的代码就象一个人写的!”，那么用Checkstyle绝对是正确选择！

**离线安装（推荐）**

下载checkstyle离线安装包；

<http://sourceforge.net/projects/eclipse-cs/>



下载完成后将压缩包中features文件夹和plugins文件夹的复制到eclipse中的根目录中进行覆盖。

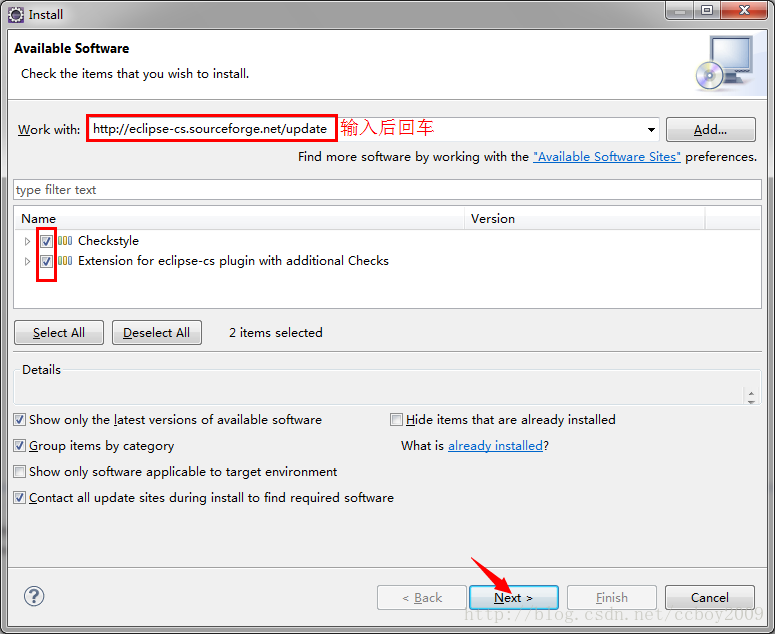
打开eclipse，Window-->Preferences,如果列表中出现了checkstyle的菜单，则表示安装成功了。

**在线安装**

注意在安装过程中保持网络畅通，如果网络不稳定可能会导致安装失败。

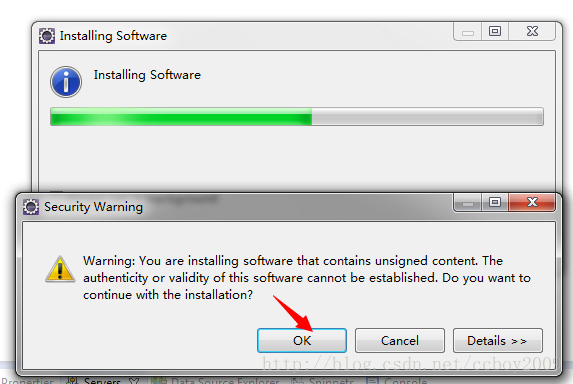
Help --> Install New Software

CheckStyle插件地址：<http://eclipse-cs.sourceforge.net/update>



Next-->next-->选择 I accept... Finish。Checkstyle插件就开始安装。

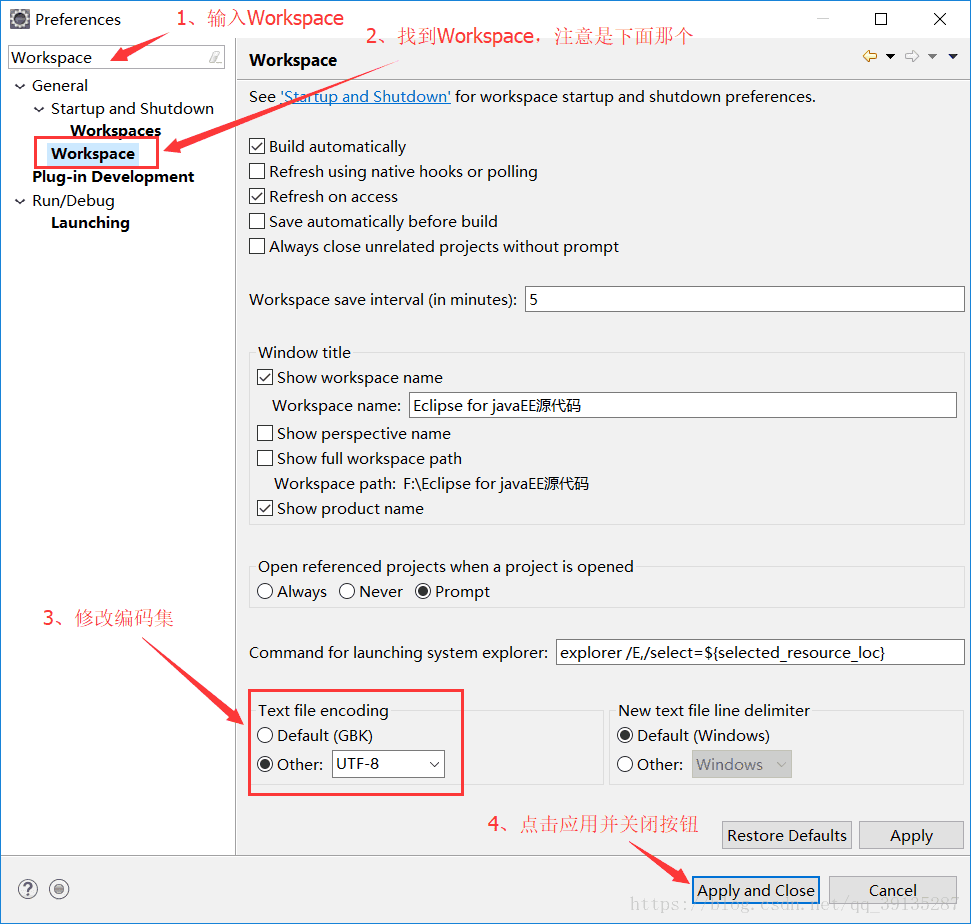
在安装过程中，可能会出现警告提示，选择ok即可。



安装完成后，根据提示重启eclipse，checkstyle即安装完成。

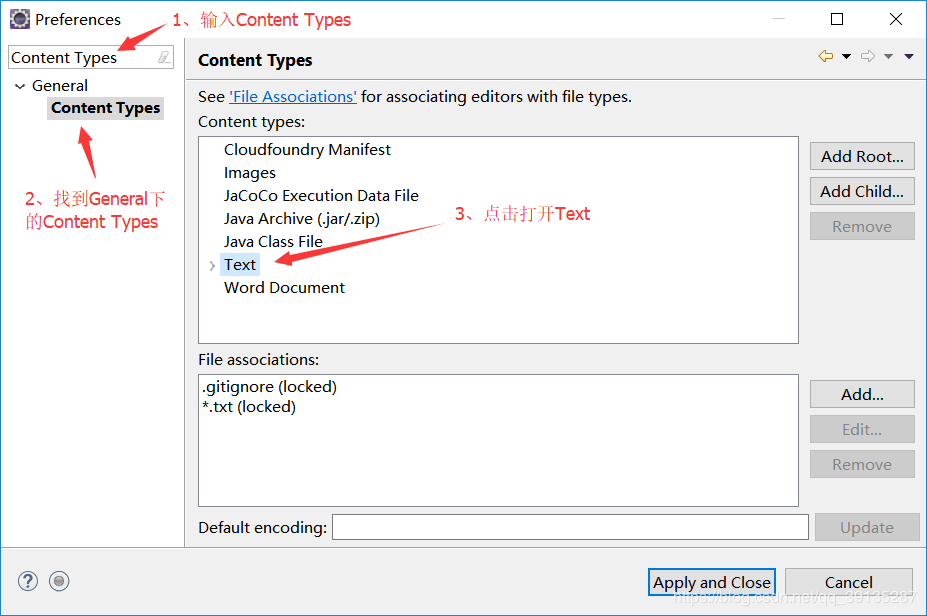
#### 修改Eclipse默认的工作空间编码集，默认编码集为GBK，修改成UTF-8

Window → Preferences🡪输入Workspace，找到第二个Workspace，修改编码集为 UTF-8,最后点击Apply and Close 按钮。

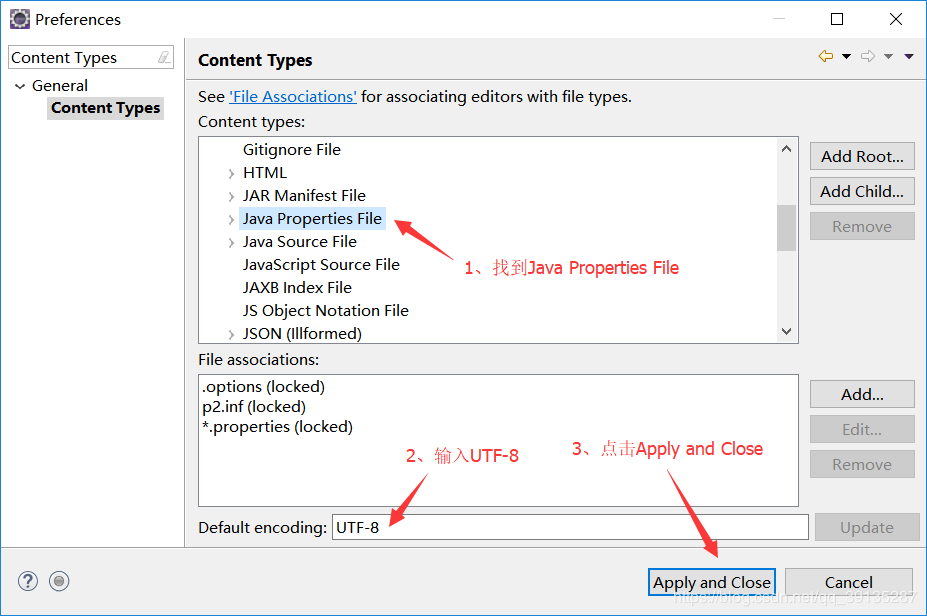


#### 修改Eclipse的资源文件编码集，修改成UTF-8

Window → Preference🡪 Content Types，General🡪 Content Types，并点击打开Content Types：中的Text。

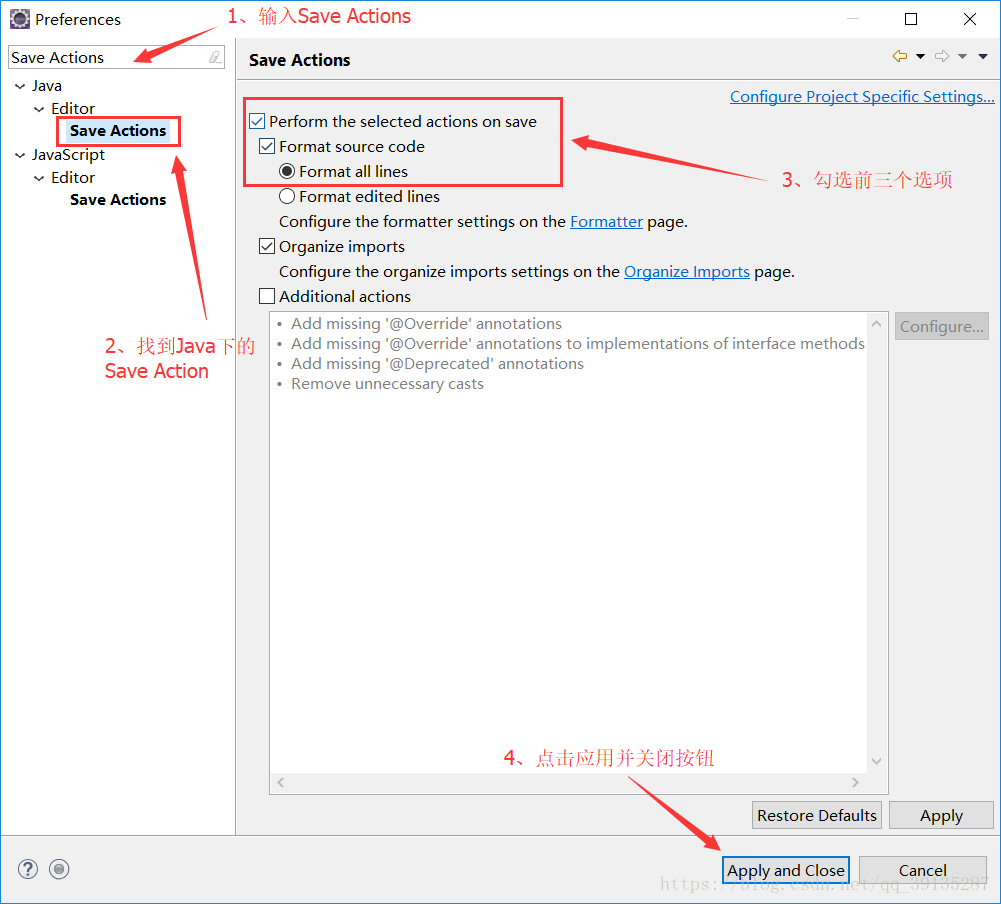


找到Text下的Java Properties File、Java Source File 和 XML，在下面的Default encoding输入框中输入UTF-8，并点击Apply and Close 按钮。



#### 设置 ctrl + s 保存时，自动格式化所有行

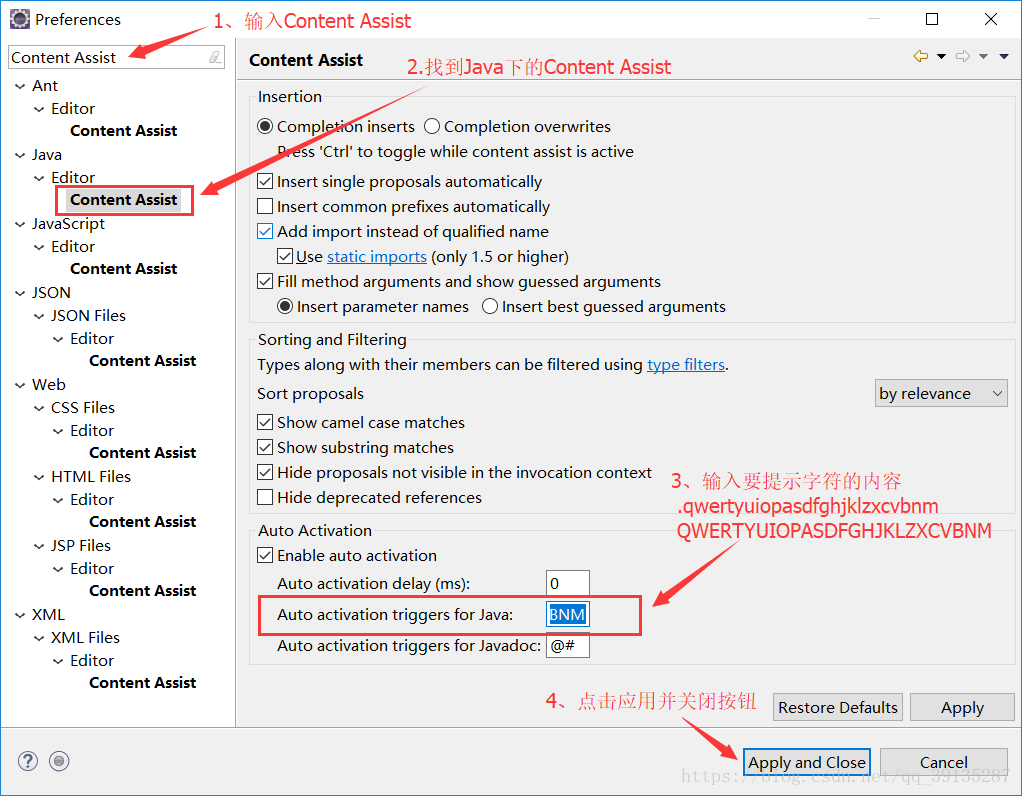
找到 Window → Preferences，在左上角的输入框中输入Save Actions，找到Java → Editor → Save Actions，把前三个选项都勾选上，最后点击Apply and Close 按钮。操作如下图：



#### 设置内容自动提示

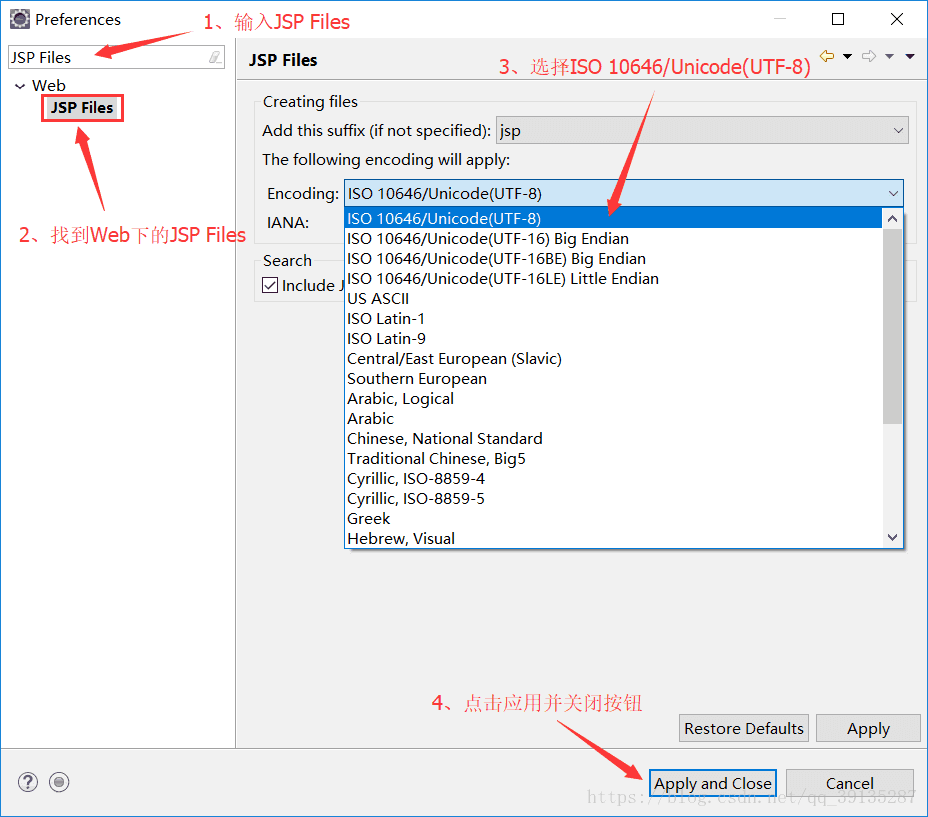
Eclipse默认 按下点（即 .）的时候，才会出现内容自动提示，现在设置除了按下点之外，任何一个字母的大小写都会有内容自动提示。

找到 Window → Preferences，在左上角的输入框中输入Content Assist，找到Java → Editor → Content Assist，在Auto activation triggers for java对应的输入框中输入 .qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM，最后点击Apply and Close 按钮。操作如下图：



#### 修改jsp默认的编码集，默认编码集为ISO Latin-1 ，修改成ISO 10646/Unicode(UTF-8)

找到 Window → Preferences，在左上角的输入框中输入JSP Files，找到Web → JSP Files，在Encoding下拉选择框中选择 ISO 10646/Unicode(UTF-8)，最后点击Apply and Close 按钮。操作如下图：



#### Eclipse代码模板的构建

当你对某一段代码写的比较厌烦的时候，你可以构建代码模板，通过模板名称来创建出你的模板内容，例如：当你输入try\_catch\_finally，并按下回车键，就会出现如下代码片段。

try {

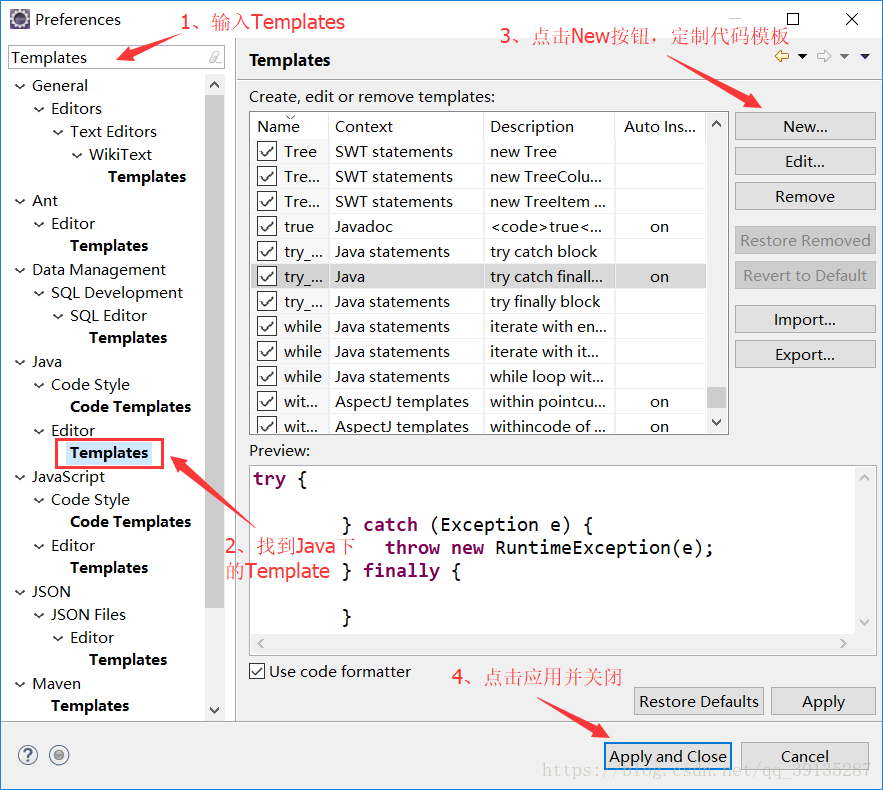
} catch (Exception e) {

throw new RuntimeException(e);

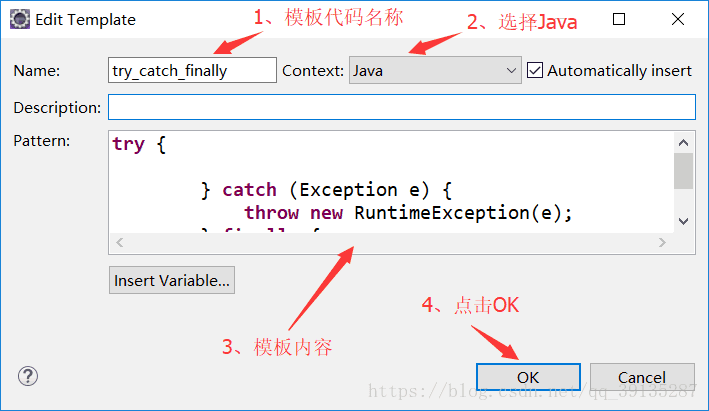
} finally{

}

找到 Window → Preferences，在左上角的输入框中输入Templates，找到Java → Editor → Templates，点击New按钮，自定义代码模板，最后点击Apply and Close 按钮。操作如下图：



代码模板构建操作如下图：



#### Eclipse快捷键配置

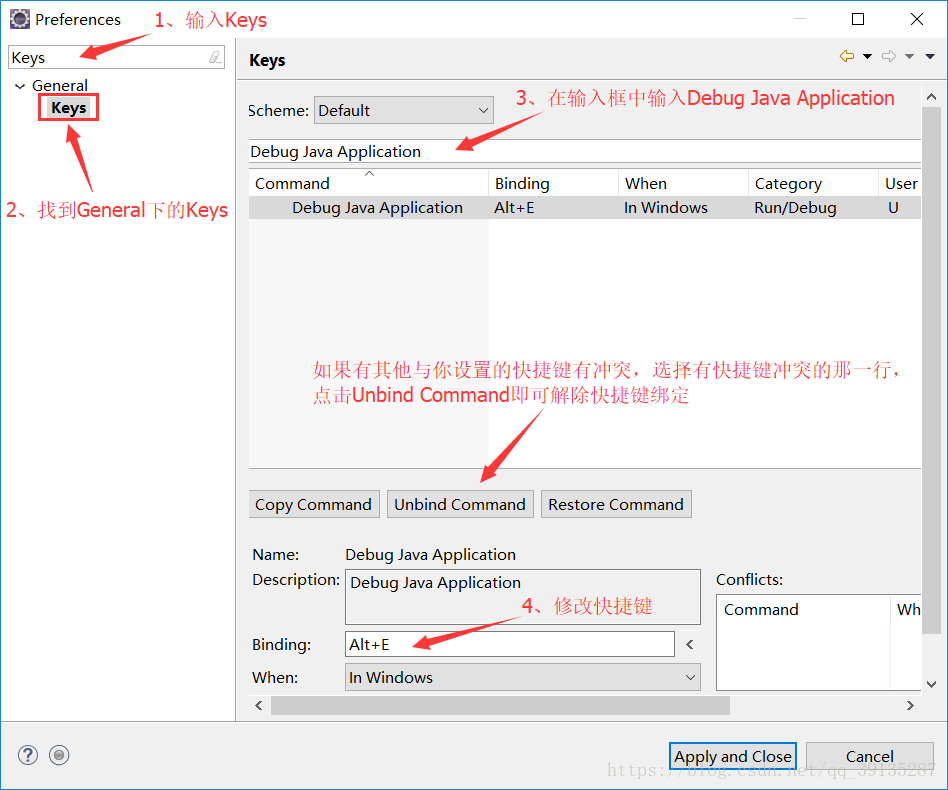
找到 Window → Preferences，在左上角的输入框中输入Keys，找到General → Keys，在Keys视窗的输入框中

输入Debug Java Application，找到它修改它的Binding为Alt+D

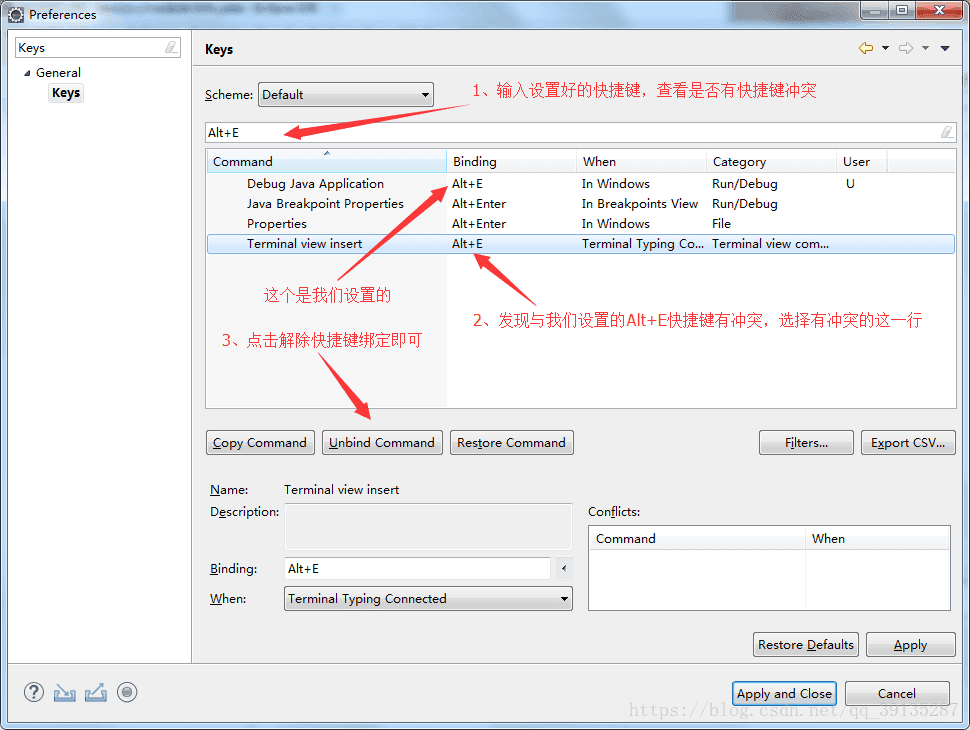
输入Debug JUnit Test，找到它修改它的Binding为Alt+J

输入Debug on Server，找到它修改它的Binding为Alt+E

操作如下图：



修改完之后，在输入框中输入你修改的快捷键，例如：在输入框中输入Alt+E，看是否会有快捷键冲突，如果有冲突，点击冲突的那一行选项，再点击Unbind Command即可解除快捷键绑定即可。



#### Eclipse配置方法注释模板

找到 Window → Preferences，在左上角的输入框中输入Code Templates，找到Java → Code Style → Code Templates，在Code Templates视窗中选择Comments,配置注释模板（即 alt+shift+j 手动添加注释，提示：1、将光标放到类名上，按下快捷键 alt+shift+j 会添加类的注释。2、将光标放到方法名上，按下快捷键 alt+shift+j 会添加方法的注释）

选择Comments → Methods 配置方法的注释模板（即 手动添加注释），双击Methods，添加如下代码到你的Pattern中：

/\*\*

\* @Title: ${enclosing\_method}

\* @Description: ${todo}(这里用一句话描述这个方法的作用)

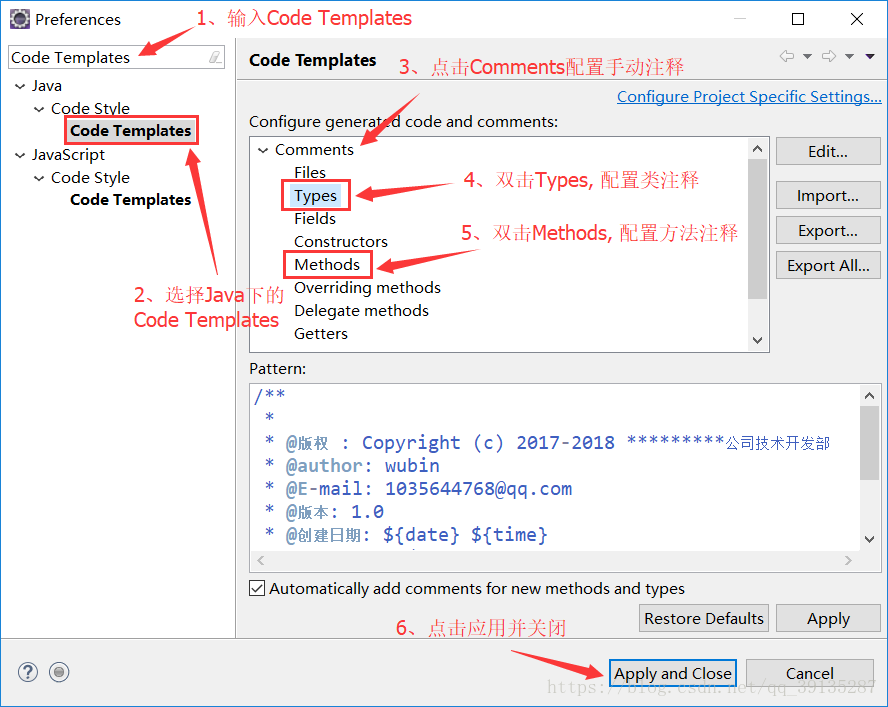
\* ${tags} 參數描述

\* @return ${return\_type} 返回类型

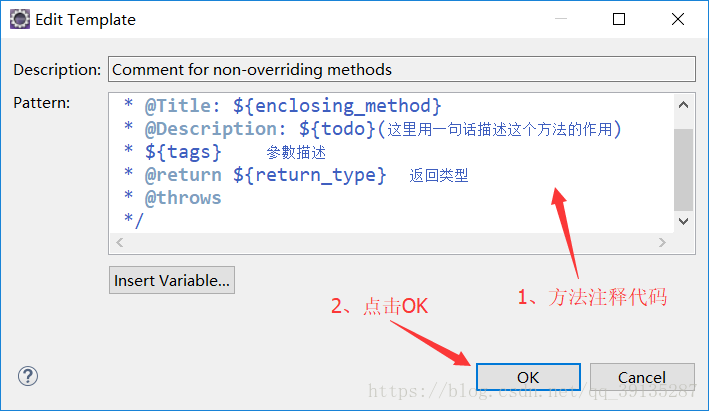
\* @throws

\*/

操作如下图 ：

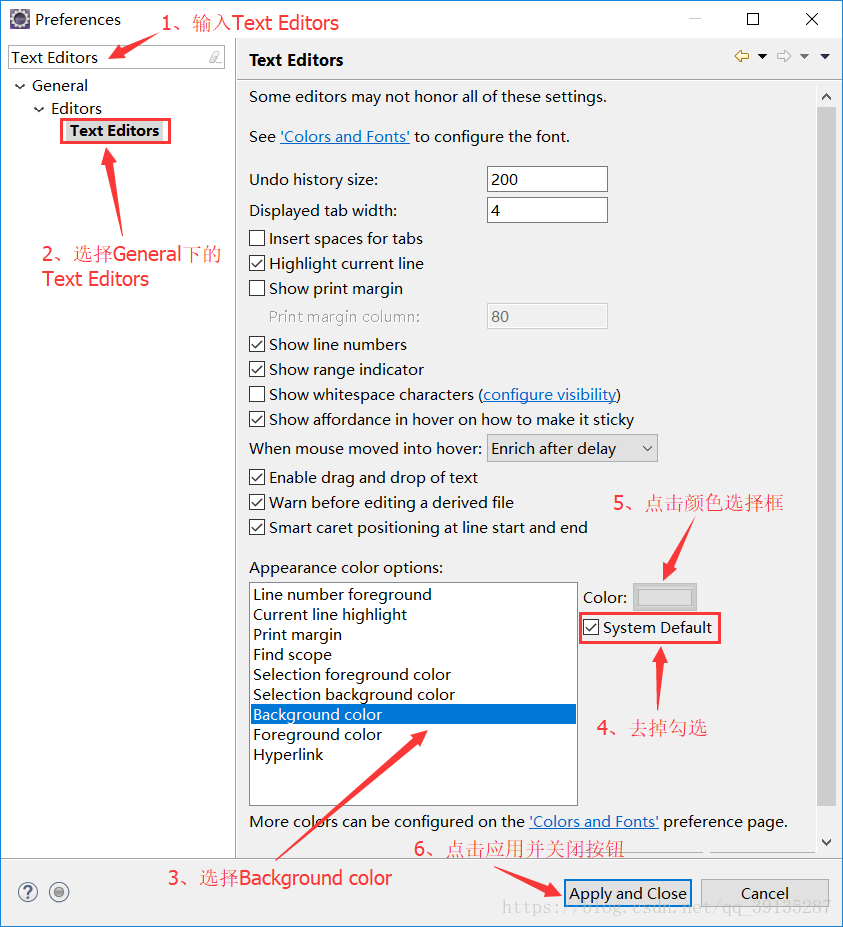


方法注释模板配置：

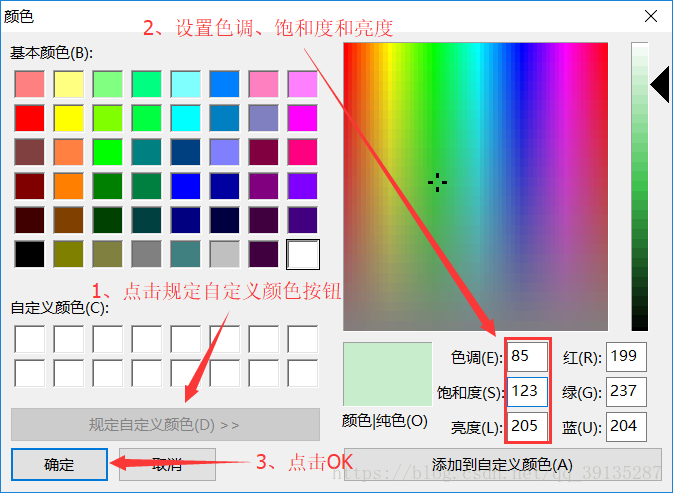


#### Eclipse背景颜色配置

找到 Window → Preferences，在左上角的输入框中输入Text Editors，找到General → Editors → Text Editors，在Text Editors视窗中的Appearance color options中选择 Background color，把右边的System Default的勾选去掉，点击Color旁边的颜色选择框，操作如下图 ：

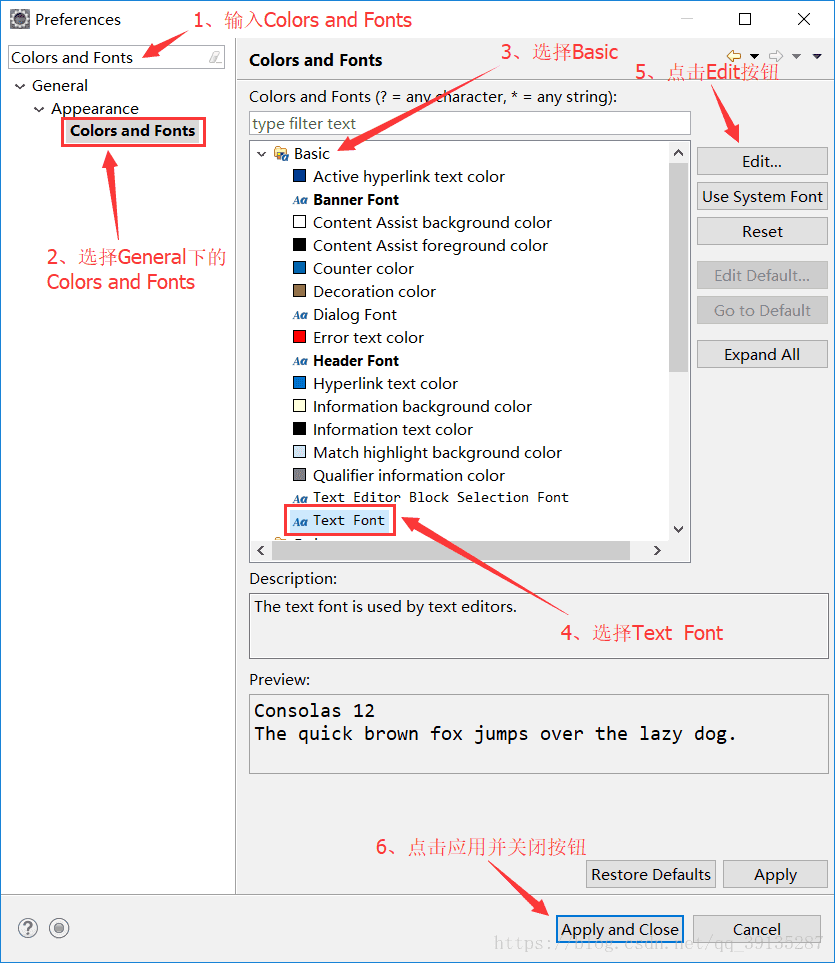


在弹出的颜色选择框中 点击规定自定义颜色(D)按钮，设置色调(E)：85，饱和度(S)：123，亮度(L)：205即可，对应的红(R) 绿(G) 蓝(U) 会作相应的变化，在点击确定即可 操作如下图：

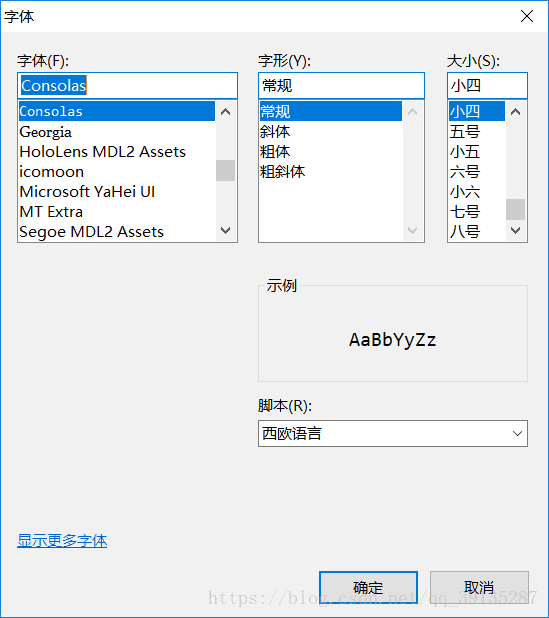


#### Eclipse字体大小设置

找到 Window → Preferences，在左上角的输入框中输入Colors and Fonts，找到General → Appearance → Colors and Fonts，在Colors and Fonts视窗中的Basic 中选择 Text Font，点击右边的Edit，选择相应的字体大小即可，操作如下图 ：



字体大小和风格可以根据个人的爱好随意设置，如下是我自己的设置：



#### Eclipse去掉类名或者接口名前自动添加上的包名

 当我们输入List并点击回车时，输出了  java.util.List<E>，Eclipse把前面的包名java.util也给带上了，而我们希望前面的包名是通过import 方式导入，这是我们就需要去掉前自动添加上的包名。

一、找到 Window → Preferences，在左上角的输入框中输入Content Assist，找到Java → Editor → Content Assist，在Content Assist视窗中勾选Add import instead of qualified name选项即可，操作如下图 ：

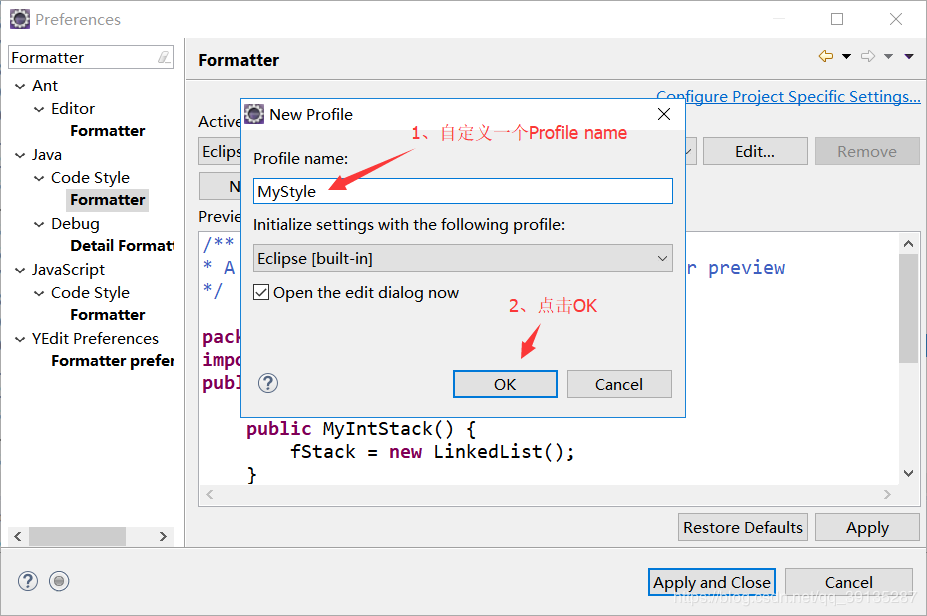


#### Eclipse设置不格式化注释

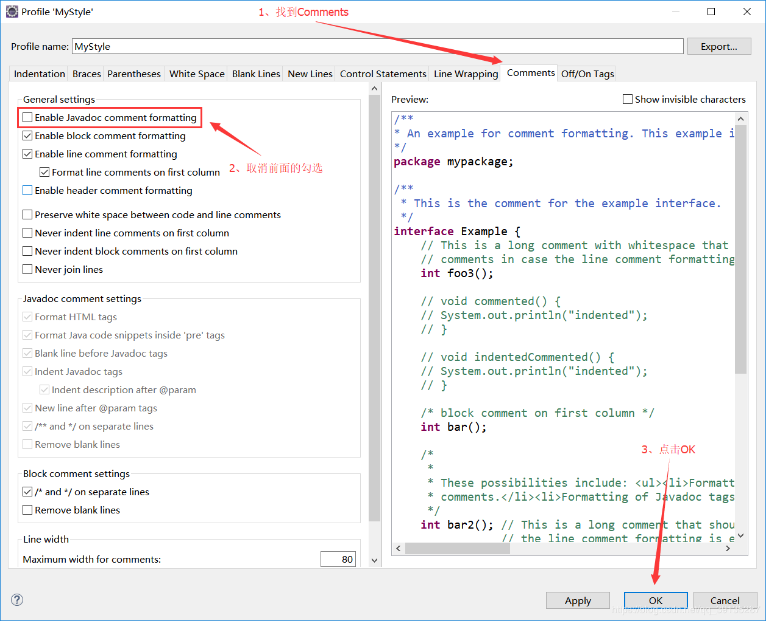
找到 Window → Preferences，在左上角的输入框中输入Formatter，找到Java → Code Style → Formatter



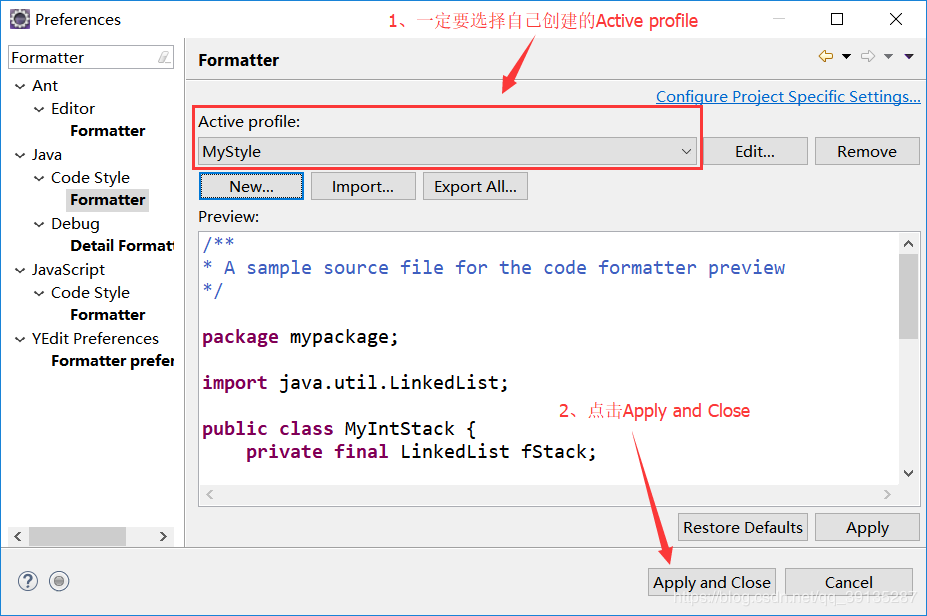
之后点击New按钮，新建一个属于自己的Active profile（名字自定），记住不要使用eclipse自带的Active profile



之后会弹出一个编辑界面，选中Comments，取消勾选“Enable Javadoc comment formatting”，最后点击OK按钮



查看Active profile是否是自己创建的Active profile，最后点击Apply and Close



Eclipse的配置就暂时到这了。

## 安装jdk

### 卸载旧版本

#yum remove java

### 首先到oracle官网下载jdk，

<https://download.oracle.com/otn/java/jdk/8u241-b07/1f5b5a70bf22433b84d0e960903adac8/jdk-8u241-linux-x64.tar.gz>

#tar -zxvf jdk-8u241-linux-x64.tar.gz -C /usr/local

### 配置java环境变量

#gedit /etc/bashrc

export JAVA\_HOME=/usr/local/jdk1.8.0\_241

export PATH=$PATH:$JAVA\_HOME/bin

export CLASSPATH=$JAVA\_HOME/jre/lib/rt.jar:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

#source /etc/bashrc

### 查看是否安装成功

#java -version

## eclipse配置maven

点击window–>preferences–>maven–>Installations

Add按钮，填入/usr/local/apache-maven-3.6.2（如果已有内嵌版本，则不用再配置）

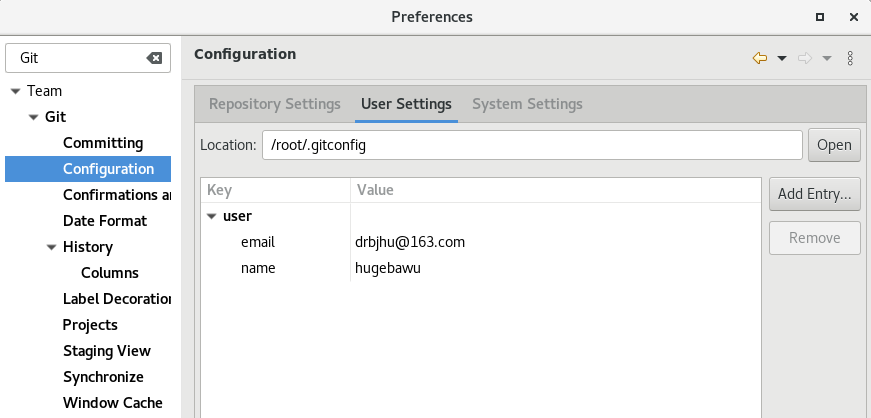
将maven安装目录的settings.xml拷贝到/root/.m2/下

#cp /usr/local/apache-maven-3.6.2/conf/settings.xml /root/.m2/

## eclipse配置git

新版eclipse自带git，不用安装了。

Windows🡪Preferences🡪Team🡪Git🡪Configuration



点击Add Entry，在弹出框里面输入key和value的值

名字的key ：user.name ； value：是你的github用户名。

邮箱的key：user.email ; value:你的登陆GitHub邮箱账号。

以后你使用git提交代码时候，eclipse会自动提取这些信息，和代码一起发送到git远程仓库。

## 配置ssh-keygen

#ssh-keygen #生成公私钥命令

1) 查看公钥：

$ cd ~/.ssh

$ cat id\_rsa.pub

如果没有的话直接抓到第3)步骤

2) 然后将其复制到github网站的个人settings中即可。

3) 生成新公钥方式

$ ssh-keygen

## 导入java项目

### 注意！！！：

1. 注意workspace路径路径

2. 以下两种方式，都需要在将该代码的**原始版本下载到本地之后，得先**提交到自己的github远程仓库，用于维护自己的对该版本的修改、注释等，方便后续快速使用！！！参考“Git学习”章节操作。

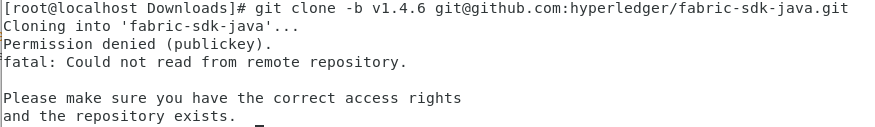
### 方法一，先把项目下载到本地，然后以Maven项目方式导入（推荐，考虑项目配置问题）

#### 获取项目源码

$ git clone [git@github.com:hyperledger/fabric-sdk-java.git](mailto:git@github.com:hyperledger/fabric-sdk-java.git) (ssh 方式)

这种ssh方式需要将linux系统的公钥配置到github.com上去，否则会报如下错误。配置方式参考”配置ssh-keygen”很多服务器都是需要认证的，ssh认证是其中的一种。在客户端生成公钥，把生成的公钥添加到服务器，你以后连接服务器就不用每次都输入用户名和密码了。

很多git服务器都是用ssh认证方式，你需要把你生成的公钥发送给代码仓库管理员，让他给你添加到服务器上，你就可以通过ssh自由地拉取和提交代码了。

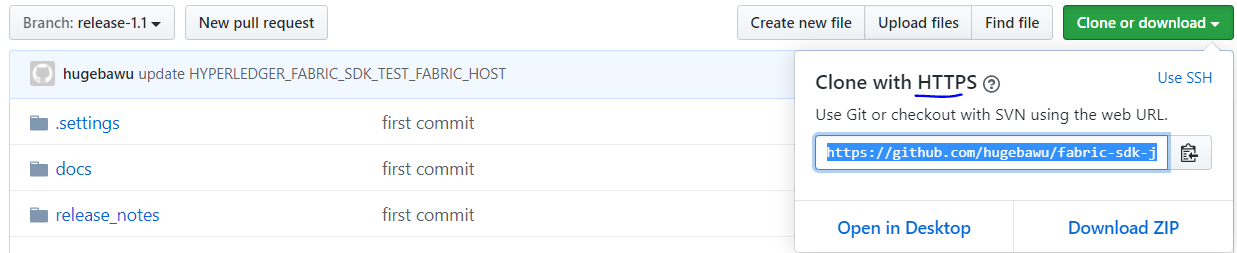


#### 导入

File > Import > Maven > Existing Maven Project. Point to the location where you cloned the SDK

### 方法二：直接从git云端下载项目到本地

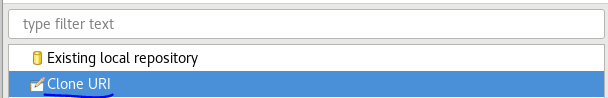
#### 从git.hub网站克隆.git结尾的网址



这里我使用https方式，很奇怪为什么ssh方式不能成功。

#### 从Eclipse中选择File-》import-》Git-》Projects from Git

选择next-》选择Clone URI



#### 选择next，（如果导入自己github仓库的项目的话，则注意填上在github上申请的账号密码：hugebawu 1962111mm，否则第一次从git官方导入的话，默认User：git）

#### 选择next，进入分支选择，如果是大型项目，可能会有多个分支。这里可选择的是release-1.1或release-1.4。

#### 选择next，配置本地项目存放路径/root/eclipse-workspace/fabric-sdk-java

#### 单击Finish，等待下载完成。然后选择Import existing projects，然后再单击Finish即可在Eclipse中看到导入的项目了。（项目复制到了eclipse的工作空间中）

然后等待eclipse对源码的第一次编译。

## Eclipse常用快捷键

Ctrl+Shift+T 打开类型

Ctrl+Shift+R 打开资源(查找目标文件)

**搜索**

Ctrl+H 全局搜索

Ctrl+G 工作区中的声明

Ctrl+Shifi+G 工作区中的引用

Ctrl+T 查看某一个类的继承类或者实现类

**文件**

Ctrl+Shift+S 全部保存

Ctrl+Shift+F4 全部关闭

Alt+Enter 属性

Ctrl+N 新建

源代码

Ctrl+Shift+Y 将所选区域字母设置为小写

Ctrl+Shift+X 将所选区域字母设置为大写

Alt+Shift+J 方法添加注释

运行

Alt+R Run Junit Test(windows>>Preferences>>Keys中自定义配置)

Ctrl+F11 运行上次启动

# Fabric-sdk-java项目配置

## 注意

### 先根据docs/EclipseSetup.md 来配置。

### 必须详细阅读README.md

可知要运行End2EndIT.java中的测试案例，必须先在src/test/fixture/sdkintegration中执行：

$ ./fabric.sh restart

### 运行End2EndIT.java，记得将src/test/java加入build path，不然不能被识别成Junit测试程序

### 此外，注意仔细看错误提示内容，对症下药！！！

## 常见错误及解决办法

### 错误1 Step 4

: Open up Eclipse to import the project. Go to File > Import > Maven > Existing Maven Project. Point to the location where you cloned the SDK in step 2.

\* If you see the error: \*\* Failed to read the project description file (.project) for 'fabric-sdk-java-master'.  The file has been changed on disk, and it now contains invalid information.  The project will not function properly until the description file is restored to a valid state. \*\*

.project文件出现问题或者.project文件丢失

解决办法：Remove the comments on top of .project file and .classpath and try importing the project again. 也就是把.project和.classpath文件删除，重新导入project，Maven -> Update Project会重新生成这两个文件。

Hit Finish.

### 错误2 Step 5

: Hit Finish. \* If you see error: \*\*No marketplace entries found to handle maven-antrun-plugin:1.4:run in Eclipse. Please see Help for more information.\*\*

解决办法：Continue to hit Finish. We will resolve this later.

### step 6: Download dependencies

Navigate to you fabric-sdk-java folder, and execute the following command: \*\*mvn install -DskipTests\*\*

### 错误3 Step 7

: After running the command in step 6, you might see the following error in Eclipse（在pom.xml可以看到）:

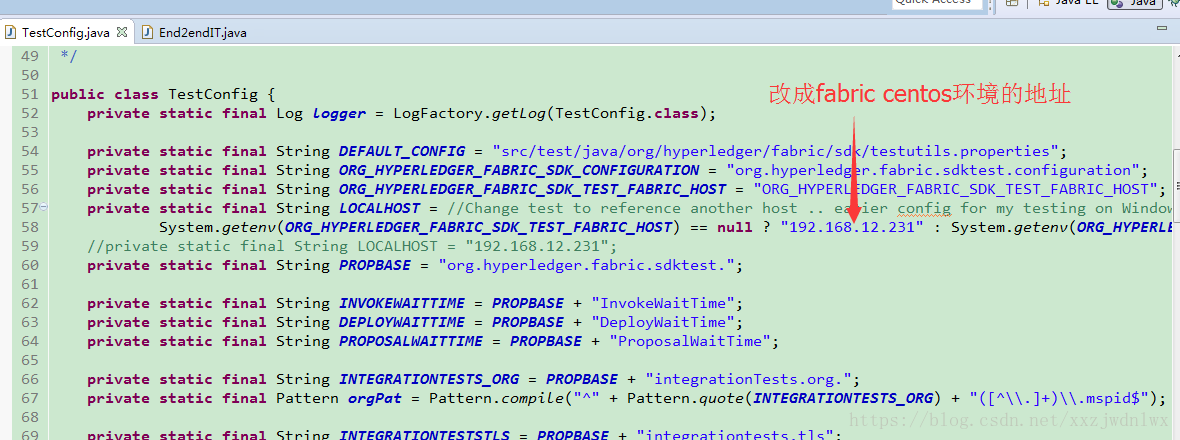
Plugin execution not covered by lifecycle configuration: org.apache.maven.plugins:maven-antrun-plugin:1.4:run (execution: default, phase: generate-test-resources).

解决办法：

Right click on the error and choose: \*\*Mark goal run as ignored in eclipse.\*\*

### 修改TestConfig.java中服务的地址

将localhost改为自己fabric centos 环境服务器的地址



重新编译项目

#mvn install

### Step 8: Enable Checkstyle

Go to Project > Properties > Checkstyle. Make sure the “Checkstyle active for this project”, “files outside source directories”, and “derived (generated) files” are all checked.

### Step 9: Hit “Local Check Configurations”

then select “New”. Choose “External Configuration File”, hit “Browse” and select “checkstyle-config.xml” from the fabric-sdk-java root directory.

### Step 10: Hit "Apply and Close"

and the checkstyle will be applied.

先按照 docs/EclipseSetup.md 来操作，操作完成后，还会遇到问题。

### POM错误：（该错误貌似只在release1.1中有）

com.google.protobuf:protoc:exe:${os.detected.classifier}:3.0.0

问题就出在 ${os.detected.classifier}，需要我们手动把他替换掉。

随便建一个class，main中写：

System.out.println(System.getProperty("os.name"));

System.out.println(System.getProperty("os.arch"));

如果报错：Implicit super constructor Object() is undefined for default constructor.

解决办法：把java的类库加载进去，在工程上右键选择属性->Java Build Path的Libraries->Add Library选择JRE System Library->点击Next->选择Execution environment并选择版本或workspace default jre->点击Finish。

得到：

Linux

amd64

打开网页：

<https://github.com/trustin/os-maven-plugin>

查看os.detected.name和os.detected.arch分别对应的值

linux和x86\_64

在 pom.xml 的 properties 中新增：

<os.detected.classifier>linux-x86\_64</os.detected.classifier>

### POM 的 plugins 部分标签错误（该错误貌似只在release1.1中有）

在 plugins 上使用 <pluginManagement> 标签将其包起来

### java 代码报错 找不到 fabric 相关类（该错误貌似只在release1.1中有）

查看 **README**，version根据错误提示决定。

pom.xml 中添加依赖：

<dependency>

<groupId>org.hyperledger.fabric-sdk-java</groupId>

<artifactId>fabric-sdk-java</artifactId>

<version>1.4.8</version>

</dependency>

导入这个有可能在项目的maven dependencies中加载不进来fabric-sdk-java的jar包，从而导致项目中的类import不了jar包中的类，这是因为项目的POM文件的groupId与你引入的依赖的groupId一致了，此时把项目的groupId改个其它名字即可。

还需要修改 pom.xml 的 <groupId> 值，现有值后面随便加点什么就可以，是为了保证其值和刚添加的依赖的 groupId 不相同，以免无法导入。

### 选择 Maven -> Update Project 就可以了

### Junit测试运行 End2endIT.java

这里需要运行样例End2end，否则会报错。

## Linux下 Release1.4配置及常见错误解决办法

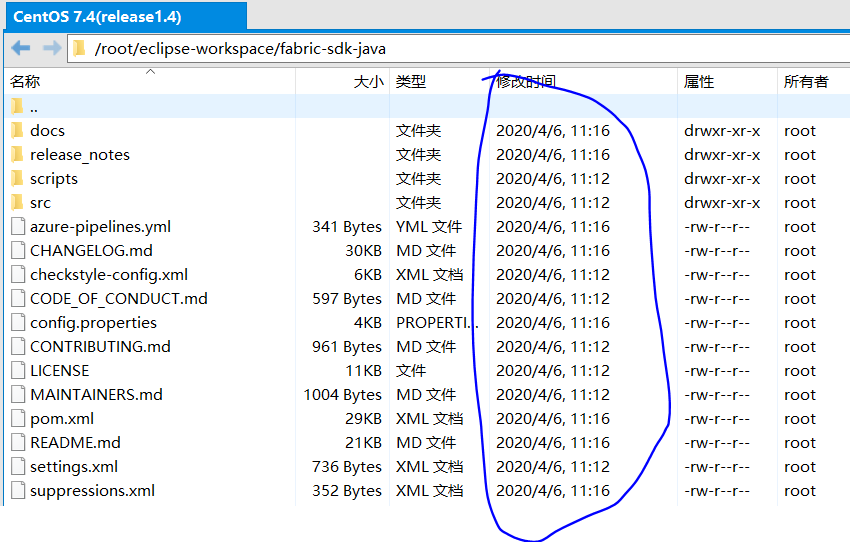
### 导入项目

$ git clone [git@github.com:hyperledger/fabric-sdk-java.git](mailto:git@github.com:hyperledger/fabric-sdk-java.git)

切换到版本release-1.4

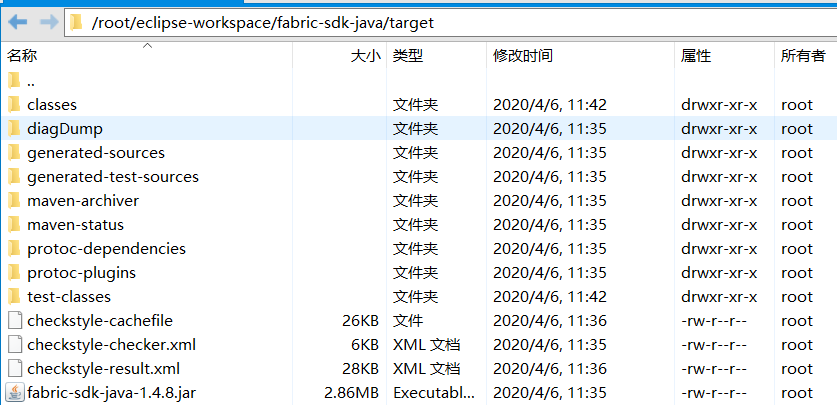
$ git checkout release-1.4

可在xftp6 中的“修改时间”看出已经切换成功。



### 先按照 docs/EclipseSetup.md 来配置，操作完成后，还会遇到问题。

特别注意的是：Step 6 \*mvn install -DskipTests\*会生成新的文件target，并对项目进行编译，特别是生成fabric-sdk-java-1.4.8.jar包。\*mvn install\*会进行单元测试，而\*mvn install -DskipTests\*不会。



必须先执行\*mvn install -DskipTests\*，后配置Step8-10的checkstyle，否则会出现如下错误：



具体原因不清楚，暂时可以忽略，不影响功能。

### Maven -> Update Project

出现错误：

Missing:

----------

1) com.google.protobuf:protoc:exe:${os.detected.classifier}:3.0.0

Try downloading the file manually from the project website.

Then, install it using the command:

mvn install:install-file -DgroupId=com.google.protobuf -DartifactId=protoc -Dversion=3.0.0 -Dclassifier=${os.detected.classifier} -Dpackaging=exe -Dfile=/path/to/file

Alternatively, if you host your own repository you can deploy the file there:

mvn deploy:deploy-file -DgroupId=com.google.protobuf -DartifactId=protoc –

等等

搜索” com.google.protobuf:protoc:exe:${os.detected.classifier}:3.0.0”可知这个原因由插件os-maven-plugin在eclipse中的使用带来的，解决方式如下：

### Issues with Eclipse m2e or other IDEs

参考网页: <https://github.com/trustin/os-maven-plugin#issues-with-eclipse-m2e-or-other-ides>

If you are using IntelliJ IDEA, you should not have any problem.

If you are using Eclipse, you need to install an additional Eclipse plugin because m2e(eclipse中的maven插件) does not evaluate the extension specified in a pom.xml. Download os-maven-plugin-1.6.1.jar and put it into the <ECLIPSE\_HOME>/plugins directory(i.e., /usr/local/eclipse/plugins).

(As you might have noticed, os-maven-plugin is a Maven extension, a Maven plugin, and an Eclipse plugin.)

根据pom.xml中os-maven-plugin的版本手动下载需要的插件(e.g., os-maven-plugin- 1.4.1.Final.jar)，将其放到elclipse根目录下的plugins目录即可。

至此，错误全部消除。Thank God！！！

### Fabric Java SDK安装测试代码

package org.hyperledger;

import org.hyperledger.fabric.sdk.HFClient;

import org.hyperledger.fabric.sdk.security.CryptoSuite;

public class App {

public static void main(String[] args) throws Exception {

// TODO Auto-generated method stub

HFClient client = HFClient.createNewInstance();

client.setCryptoSuite(CryptoSuite.Factory.getCryptoSuite());

System.out.println("fabric-sdk-java installed successfully.");

}

}

## 学会Junit测试，参考极客学院的《JUnit 入门教程》

## 跑通Fabric-sdk-java的End2endIT.java的JUnite测试（已跑通）

$ ./fabric.sh restart # 启动fabric环境

选中End2endIT.java，Run as Junit Test

其fabric网络配置相关文件在：

/root/eclipse-workspace/fabric-sdk-java/src/test/fixture/sdkintegration/

# Fabric Java chaincode开发

## 注意

1）由于笔者熟悉java，而不是go、node.js语言、因此选择使用eclipse+ fabric-chaincode-java开发chaincode。

2）详细阅读README.md

## 需要先配置好fabric网络环境，参考“Fabric环境部署”章节

## 下载fabric-chaincode-java源码

$ cd /root/eclipse-workspace下

$ git clone [git@github.com:hyperledger/fabric-chaincode-java.git](mailto:git@github.com:hyperledger/fabric-chaincode-java.git)

## Gradle介绍

Gradle官网说法：

### 建造任何东西

用Java，C ++，Python或您选择的语言编写。打包在任何平台上部署。并依靠Gradle无与伦比的多功能性来构建这一切。

### 自动化一切

    使用Gradle丰富的API和成熟的插件和集成生态系统来实现雄心勃勃的自动化。建模，整合和系统化您的软件从端到端的交付。

### 提供更快

    用优雅，快速的构建扩展开发。从编译避免到高级缓存等，我们坚持不懈地追求性能，让您的团队能够持续交付。

### 多种开发工具的使用

    包括Android Studio，Eclipse，IDEA，NetBeans

### 可以将基于maven的项目迁移到基于Gradle的项目中！

## gradle安装

手动方式类似于maven的安装

Gradle可在所有主要操作系统上运行，并且只需要安装Java JDK或JRE版本7或更高版本。在此，只介绍在linux下的安装，提供两种方式，sdkman安装和手动安装。



### 使用sdkman安装（没测试过，不推荐）

前提条件：1、linux机器能连网 2、已安装sdkman，否则使用手动下载安装方式。

$ sdk install gradle

$ sdk install gradle 4.10.2

### 手动安装

Gradle Distribution Archive有两种类型： "binary-only "和"complete"， "binary-only "存档包含Gradle软件，只有"complete "存档附带二进制文件，文档和源代码。

<https://gradle.org/releases/>

<https://gradle.org/releases/>可以选择各种版本下载，zip包，下载后解压到/usr/local

$ wget <https://services.gradle.org/distributions/gradle-4.10.2-bin.zip>

$ unzip -d /usr/local/ gradle-4.10.2-bin.zip

配置PATH环境变量:

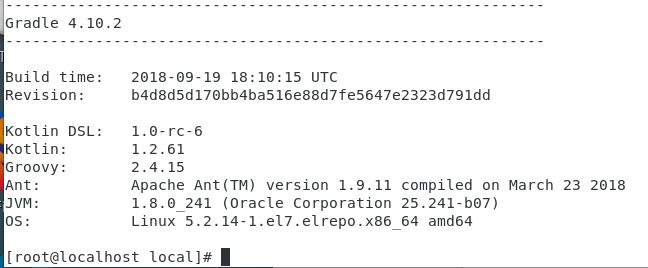
$ gedit /etc/bashrc

$ export PATH=$PATH:/usr/local/gradle-4.10.2/bin

$ source /etc/bashrc

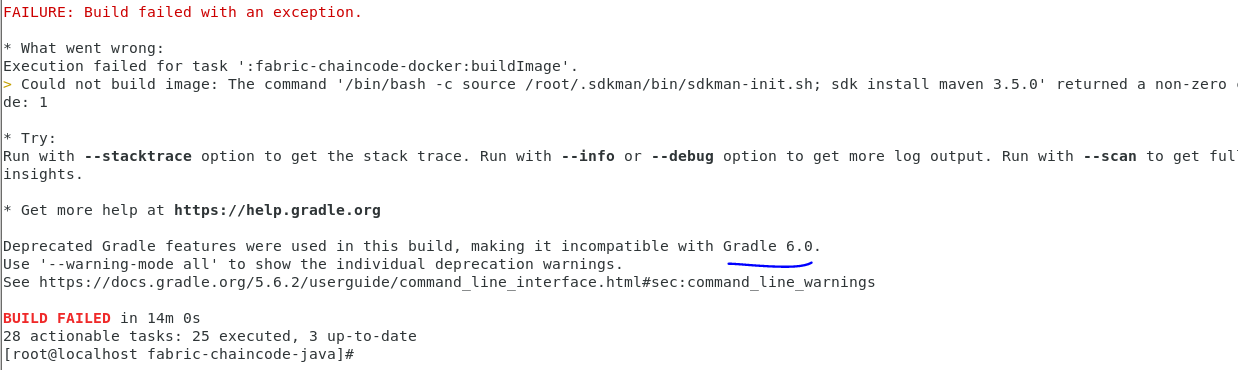
**检查是否安装成功**

$ gradle -v

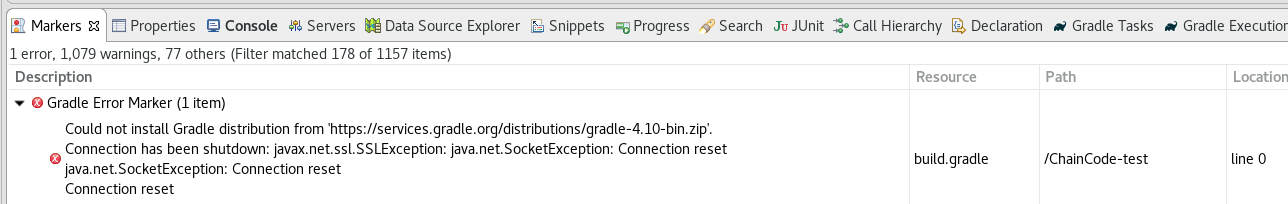


### 测试gradle 版本是否与fabirc-chaincode-java匹配

笔者第一次安装用的4.10版本，根据READM.md中的指令$ ./gradlew clean build classes(或者./gradlew clean build install)编译项目，出现错误如下：



以及eclipse中出现错误：

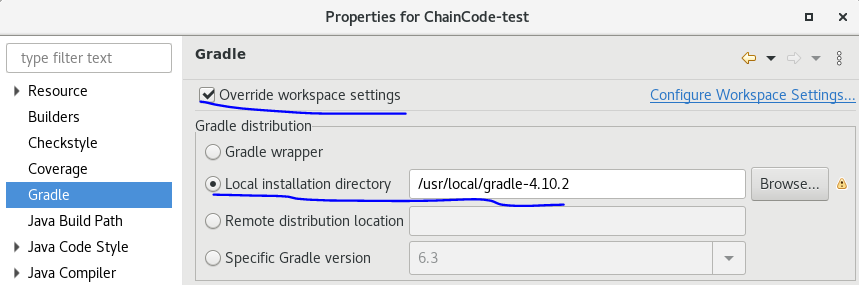


这个是gradle版本选择错误原因。目前不知道这个版本怎么快速确定。

1）版本可以根据《Fabric区块链开发详解》的环境中gradle版本看出来的。



2）或者通过在项目properties中手动测试出来。将gradle版本换成4.10.2后消除错误。这两个地方改成这样！！！



记得在eclipse中分别Maven🡪updata project和Grandle-refresh Grandle project后eclipse中的错误显示才能消失！！！

## 安装REST API客户端（官方文档用的是SoapUI）（暂时不知道如何使用）

“Blockchain chaincode for Java developers——How to write chaincode for Hyperledger Fabric v0.6 using the Java language”这篇文章要求装的。

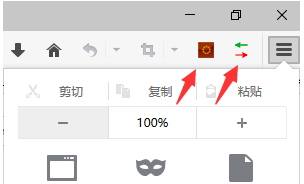
笔者使用firefox浏览器自带的RESTClient插件，具体安装方法可以自行百度。

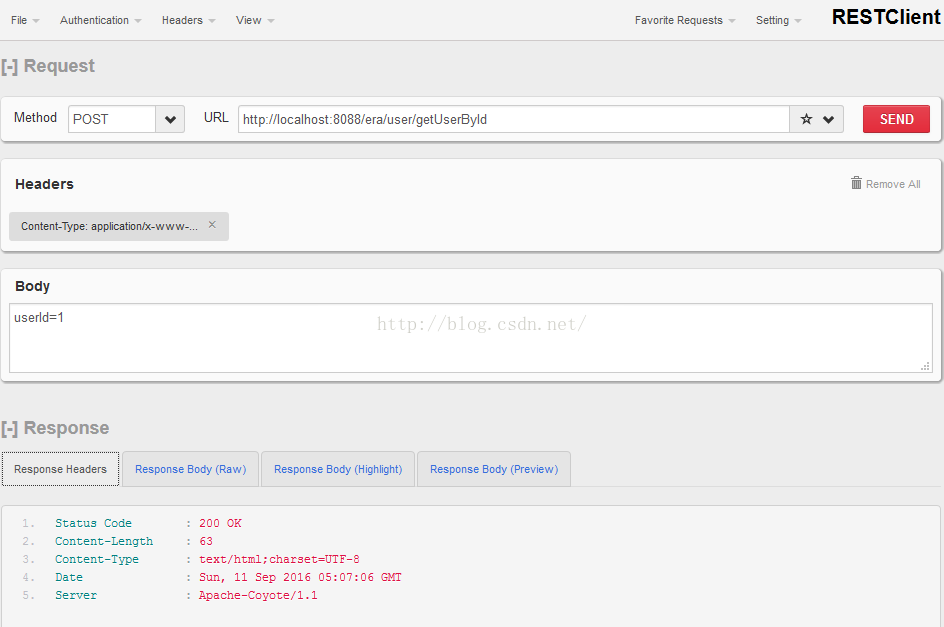
<https://blog.csdn.net/aqsunkai/article/details/52503222>

我们写好一个接口后，需要进行测试。有时我们会写一个html表单提交，无疑增加了工作量，尤其是当参数比较多或者传json或xml数据时，效率更是大大降低。我们可以使用基于FireFox的RESTClient或HttpRequester插件来进行测试。

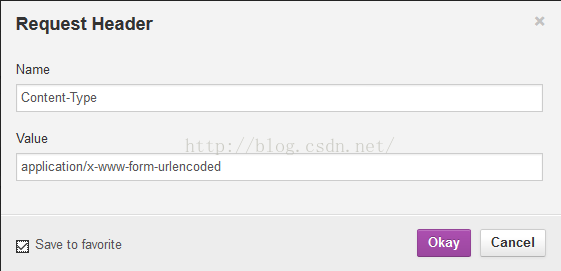
打开火狐浏览器，在右上角的菜单中有附加组件，进入其中，直接搜索这两个插件，安装后重启即可，安装完后在浏览器右上方生成了插件的图标。

下面以RESTClient为例。HttpRequester没找到。仍准图标





发送get请求时参数在url中；当发送post请求传参时需要设置头信息，点击上图中的Headers-Custom Header进行设置



## Linux中Eclipse安装与配置gradle

仔细阅读：

<https://github.com/eclipse/buildship/blob/master/docs/user/Installation.md>

### 方式一（推荐）Eclipse Marketplace搜索安装：

eclipse-->help-->Eclipse Marketplace,然后搜索buildship

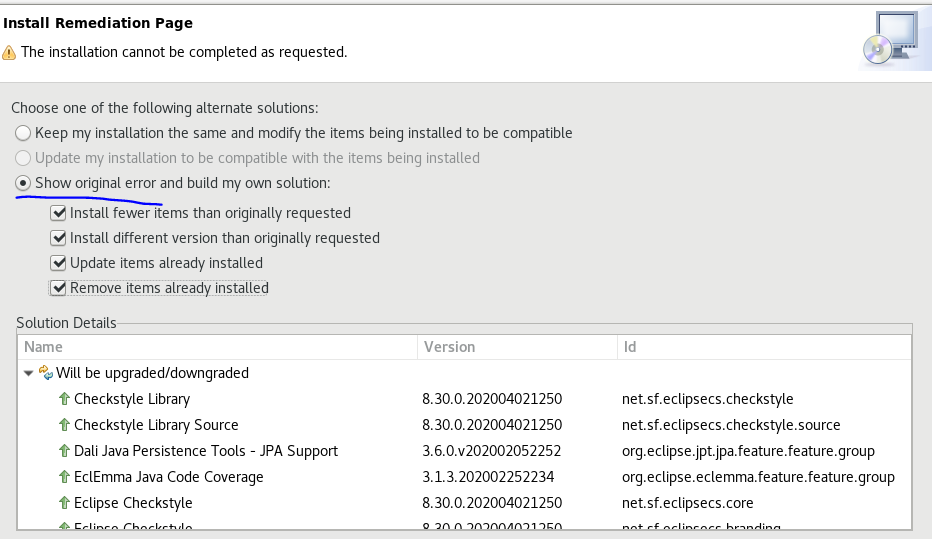
### 方式二：

1）eclipse help》Install new software，输入

http://dist.springsource.com/release/TOOLS/gradle 回车。

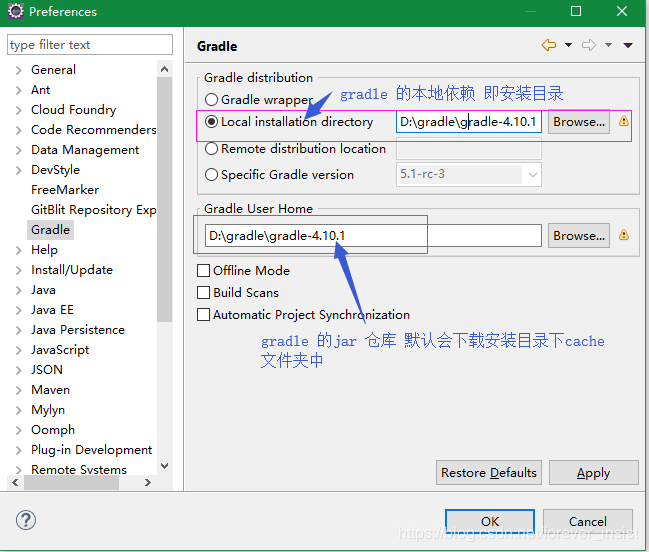
2）选择Extensions/Gradle Integration和Uncategorized点击下一步

3）选择第三项，子项全选



4）Next 选择“I accept …”,等待安装完成

### 配置



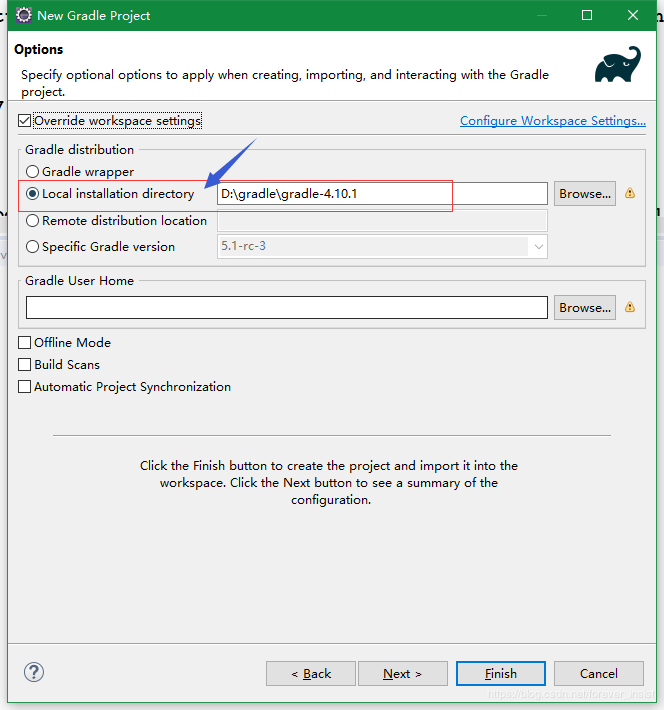
这里的Gradle User Home路径使用默认的就好。Linux下为：/root/.gradle。是一个用于gradle存放jar包的本地仓库。

## 开发自己的java chaincode

可参考fabric-v0.6对应的java chaicode开发中文版文档：

<https://www.ibm.com/developerworks/cn/java/j-chaincode-for-java-developers/index.html>

### 新建或者导入gradle项目



红圈处填写你的gradle目录，然后点击finish。此时就会生成名为 ChainCode-test的gradle项目。（时间比较长，耐心等待！！！）

gradle 插件为我们提供了一个常用的Gradle task列表，便于我们加快我们开发速度。

### 导入jar包

#### 使用build.gradle自动导入jar包

打开build.gradle文件，参考chaincode任一范例中的build.gradle文件修改此文件的内容。

参考：中的build.gradle文件

<https://github.com/hyperledger/fabric-samples/blob/v1.4.6/chaincode/chaincode_example02/java/build.gradle>

此时要注意把build.gradle文件中mainClassName字段改为你相应的目录。

比如在dependencies中加入

dependencies {

compile group: 'org.hyperledger.fabric-chaincode-java', name: 'fabric-chaincode-shim', version: '1.4.6'

}

则会自动下载fabric-chaincode-shim-1.4.6相关jar包。

然后进行右键 Gradle🡪Refresh操作，此时项目会自动下载相关的jar包。

#### 手动导入jar包

需要导入其他jar包的时候：在自己的 Gradle 项目里建立一个名为 libs （这个名字可以自己定义，不一定非要叫这个名字）的文件夹，把自己本地的 jar 包拷贝到这个文件夹中。

然后在build.gradle配置如下：

dependencies { compile fileTree(dir:'libs',include:['\*.jar'])}

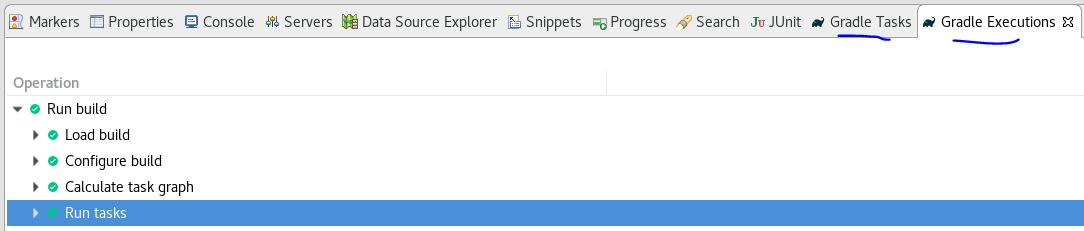
然后在此进行进行Refresh刷新操作。

### 编写chaincodejava代码

比如：在src/test/java下新建CounterCC文件夹，然后新建立CounterChaincodest.java。该代码来自“Fabric区块链Java开发详解\repo\c2\counter-cc”。

### 编译调试

在Gradle Tasks面板选中要编译的项目，然后展开build，右键单击Run Gradle Tasks进行编译；若显示如下，则编译成功：



此时在你的gradle项目中会出现build文件夹，此时可直接在此文件夹下运行gradle -b build.gradle run

命令运行程序，也可以进入build-->distributions文件夹，会看到如下两个压缩包：

解压任一压缩包，并进入ChainCode-test-->bin目录，此时可看见可执行文件：

在此目录下打开终端窗口，输入./ChainCode-test命令执行程序。（注意：此时fabric网络需要开启，就会在network窗口中看见链码注册到区块链网络的相关信息）

# 跑《Fabric区块链Java开发详解》中的solo-network

## 注意

### 参考汇智网的教程《Fabric区块链Java开发详解》。相关文档已经复制到文档：“Fabric区块链Java开发详解.docx”。

### 依赖v1.4.0环境

从Fabric区块链Java开发详解\repo\c2\counter-cc\build.gradle中的：

“compile group: 'org.hyperledger.fabric-chaincode-java', name: 'fabric-chaincode-shim', version: '1.4.0'”

可以看出其使用的是fabric-chaincode-java的v1.4.0版本。因此fabric和fabric-sdk-java也应该配置成v1.4.0版本。分别参考“Fabric环境部署”和“Fabric-sdk-java项目配置”章节配置好。

### 步骤2-9的fabric网络配置相关代码在：

<https://github.com/hugebawu/fabric/tree/release-v1.4.0/examples/solo-network>

对应的fabric-sdk-java代码在（执行有错误）：

<https://github.com/hugebawu/fabric-sdk-java/tree/v1.40/src/test/java/org/hyperledger/fabric/counterApp>

### 该例子没有启用任何容器

### 注意先删除artifact和data文件夹

### 学会将目录下所有文件格式由dos转换为unix风格

$ find . -type f -exec dos2unix {} \

解释：

$ find . -type f # 将目前目录其其下子目录中所有一般文件列出

$ -exec dos2unix {} \ # execute dos2unix on each file found

## 生成本地msp

### 生成ca和user私钥、证书。

$ cd mini-ca

$ ./ generate-identity.sh

### 将相关私钥、证书复制到msp中

$ cd ..

$ ./init-msp.sh

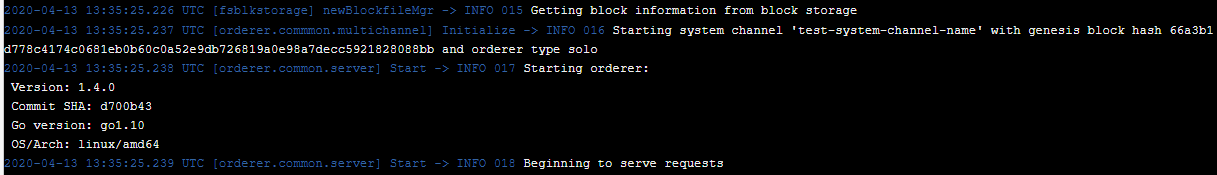
## 终端1生成系统通道创世块

$ bash create-systemChannel.sh

## 终端1启动order节点

$ bash start-orderer.sh

“Fabric区块链Java开发详解”提供的实验平台的显示结果：



## 终端2启动peer节点

$ bash start-orderer.sh

## 终端3创建以及加入通道

$ bash create-channel.sh

$ bash join-channel.sh

### 可能错误1：

Error: error getting endorser client for channel: endorser client failed to connect to 0.0.0.0:7051: failed to create new connection: context deadline exceeded

出错原因：未知。

解决办法：

将core.yaml中的如下内容的IP由0.0.0.0:7051改成127.0.0.1:7051即可。

# When used as peer config, this represents the endpoint to other peers

# in the same organization. For peers in other organization, see

# gossip.externalEndpoint for more info.

# When used as CLI config, this means the peer's endpoint to interact with

address: 127.0.0.1:7051

### ordererr终端（终端1）显示的错误：

2020-04-13 21:29:38.069 CST [orderer.commmon.multichannel] newChain -> INFO 01b Created and starting new chain ch1

2020-04-13 21:29:38.072 CST [common.deliver] Handle -> WARN 01c Error reading from 127.0.0.1:56546: rpc error: code = Canceled desc = context canceled

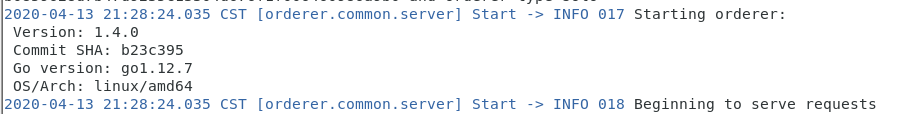
2020-04-13 21:29:38.072 CST [comm.grpc.server] 1 -> INFO 01d streaming call completed {"grpc.start\_time": "2020-04-13T21:29:38.053+08:00", "grpc.service": "orderer.AtomicBroadcast", "grpc.method": "Deliver", "grpc.peer\_address": "127.0.0.1:56546", "error": "rpc error: code = Canceled desc = context canceled", "grpc.code": "Canceled", "grpc.call\_duration": "18.52727ms"}

### 终端3显示的错误：

**2020-04-13 21:30:00.688 CST [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized**

**Error: proposal failed (err: rpc error: code = Unavailable desc = transport is closing)**

**而终端1在orderer节点启动后显示的日志如下：**



**与“Fabric区块链Java开发详解”提供的实验平台的显示结果相比，go语言版本不一样。难道是它的原因？**

**目前网页搜索不出解决办法，又不能改go语言版本为go1.1.0，因为**<https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html>中的Docs » Getting Started » Prerequisites说Go version 1.12.x is required。

## 终端4启动链码

$ cd counter-cc

$ ./start.sh

## 终端3安装、实例化、query、invoke链码

$ ./install-chaincode.sh

遇到错误，修改脚本中的countercc所在路径即可！！！

$ ./instantiate-chaincode.sh

$ ./query-chaincode.sh

$ ./invoke-chaincode.sh

---------------------以上终端1-4成功跑通！！！------------------------

## 终端5使用fabric-sdk-java开发Fabric应用，在应用中访问链码

在终端1-4跑通额基础上：

$ cd /opt/gopath/src/github.com/hyperledger/fabric/examples/counter-dapp

$ ./start.sh

第一次运行出错，提示证书验证不通过。本以为根据第一步重新生成msp文件夹即可解决错误，但是仍然提示错误。先放着吧，回头再看。

## 使用《Fabric区块链Java开发详解》提供的wiz工具实现以上步骤4-9（已跑通）

Fabric的环境配置相当繁琐，为了提高开发效率，课程提供了一个便利的开发工具箱WIZ： wiz提供了一个简单的Fabric应用开发模板，以及一组用于提高效率的脚本。

### 将wiz工具加入系统路径

wiz工具被放在fabric/example/tools中，将tools文件夹加入系统路径，则wiz可直接调用。

$ gedit /etc/bashrc

加上: export PATH=$PATH:$FABRIC\_HOME/examples/tools

$ source /etc/bashrc

$ find . -name "\*.sh" | xargs chmod a+x # 还得给wiz和所有.sh赋予可执行权限。

### 初始化项目目录（终端1）

在fabric/examples下创建一个新目录，然后执行wiz init命令初始化目录结构：

$ mkdir -p test-wiz && cd test-wiz

$ wiz init

wiz项目的目录结构组织如下：

• network：fabric网络定义与配置数据目录

• chaincode：链码源代码目录

• application：应用源代码目录

• script：脚本目录

### 启动网络（终端1）

在wiz项目目录内，使用wiz net start命令启动网络，在初次启动网络前 需要使用wiz net reset命令初始化网络配置。例如，在1#终端执行如下命令：

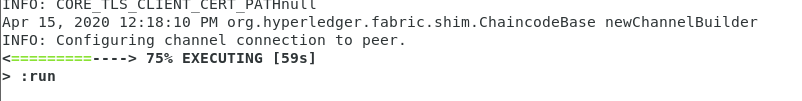
test-wiz$ wiz net reset

test-wiz$ wiz net start

### 启动链码（终端2）

在wiz项目目录内，使用wiz cc start命令启动链码。例如，在2#终端 执行如下命令：

test-wiz$ wiz cc start



等待终端2启动完成才能执行终端3的步骤。

### 启动通道环境（终端3）

在wiz项目目录内，使用wiz ch start命令启动通道并进行必要的环境初始化，包括将当前节点加入通道、安装并激活链码。例如，在3#终端执行如下命令：

test-wiz$ wiz ch start

### 启动管理终端（终端4）

在wiz项目目录内，使用wiz admin命令进入管理终端，管理终端配置了必要的环境变量，因此可以直接以管理员身份执行peer命令。例如，在4#终端执行如下命令进入管理终端，并查看当前节点加入的通道：

test-wiz$ wiz admin

admin@org1> peer channel list

### 启动应用（终端5）

在wiz项目目录内，使用wiz app start启动项目应用。例如，在5#终端执行如下命令：

test-wiz$ wiz app start