<mvc:****

**SpringMVC4**

**课程讲义**

**北京动力节点教育培训公司**

**制作人：老郭**

**Git&GitHub**

**2015年12月**

**Git&GitHub**

# 本机的github仓库：

github的token密码：ghp\_pnvjMlGQhMccm5ySECCiPrSkEBOSYR1Whnvp（提交代码用它，不能用sk812588）

GitHub：01kane0001 sk812588

# 第一章 Git概述

## 1-1版本控制工具简介

### 1-1-1什么是版本控制工具

版本控制工具提供完备的版本管理功能，用于存储、[追踪](https://baike.baidu.com/item/%E8%BF%BD%E8%B8%AA/44928" \t "_blank)目录（文件夹）和文件的修改历史，是软件开发者的必备工具，是软件公司的基础设施。

版本控制最主要的功能就是追踪文件的变更。它将什么时候、什么人更改了文件的什么内容等信息忠实地了已录下来。[软件开发](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91/3448966" \t "_blank)往往是多人协同作业，版本控制可以有效地解决版本的同步以及不同开发者之间的开发通信问题，提高协同开发的效率。

### 1-1-2版本控制工具的作用

（1）协同开发

团队协作共同完成同一个项目

（2）版本管理

以不断提升项目版本的方式逐步完成项目。

（3）数据备份

开发中以版本控制的形式保存每一个历史版本。

（4）权限控制

对团队开发人员进行不同的权限分配。

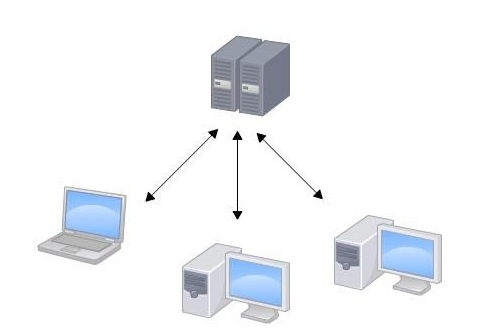
（5）分支管理

允许开发团队在工作过程中多条生产线同时推进任务，进一步提高效率。

### 1-1-3市场上常见的版本控制工具及分类

**a. 集中式版本控制工具**

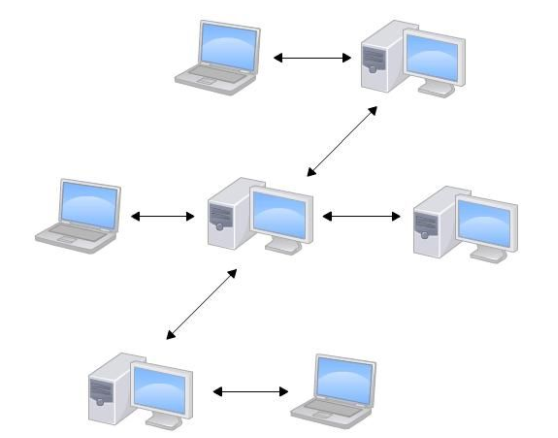
CVS、**SVN**、VSS……

****

不同的客户端直接跟服务器端交互工作，可以进行提交操作、克隆操作等。不同客户端之间是相互独立的。

**b. 分布式版本控制工具**

**Git**、Mercurial、Bazaar、Darcs……



每个用户都有自己在本地的仓库，用来管理版本控制，每个客户端也同时是一个服务器。同时也可以用一个专门的服务器的仓库用来管理。

## 1-2 Git简介

### 1-2-1 Git概述

Git是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。Git起初是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

### 1-2-2 Git历史简介

Git创建历史解读：

时间 事件

1991年 Linus在1991年创建了开源的Linux，从此，[Linux 系 统](https://www.baidu.com/s?wd=Linux%E7%B3%BB%E7%BB%9F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)不断发展，已经成为最大的服务器系统软件了

2002年以前 世界各地的志愿者把源代码文件通过diff的方式发给 Linus（Linus坚定地反对CVS和SVN，这些集中式的版本 控制系统不但速度慢，而且必须联网才能使用。有一些 商用的版本控制系统，虽然比CVS、SVN好用，但那是 付费的，和Linux的开源精神不符）

2002年 Linux系统已经发展了十年了，代码库之大让Linus很难 继续通过手工方式管理了。Linus选择了一个商业的版本 控制系统BitKeeper，BitKeeper的东家BitMover公司出 于人道主义精神，授权Linux社区免费使用这个版本控 制系统

2005年 开发Samba（Samba是在Linux和UNIX系统上实现SMB 协议的一个免费软件）的Andrew试图破解BitKeeper的 协议，被BitMover公司发现了，于是BitMover公司在愤 怒中收回了Linux社区对于BitKeeper的免费使用权

同是2005年 Linus在没有版本控制系统的窘境下，自己用C语言写了 一个分布式版本控制系统，这就是Git, 值得一提的是， Linus仅仅使用两周时间就完成了Git的创建

Git创建完毕的一个月后，Linux系统的源码正式由Git 管理

2008年 GitHub网站上线了，它为开源项目免费提供Git存储， 无数开源项目开始迁移至GitHub，包括jQuery，PHP， Ruby等等

如今，GitHub+Git的开发模式已经被全世界所认可，已然成为了全世界应用最广泛的分布式版本控制管理工具。

### 1-2-3 Git官网及Logo

官网地址：https://git-scm.com/

Logo：

### 1-2-4 Git的优点

（1）适合[分布式开发](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E5%BC%80%E5%8F%91" \t "_blank)，强调个体。

（2）公共服务器压力和数据量都不会太大。

（3）速度快、灵活。

（4）相对容易的解决冲突。

（5）大部分操作在本地完成，不需要联网。

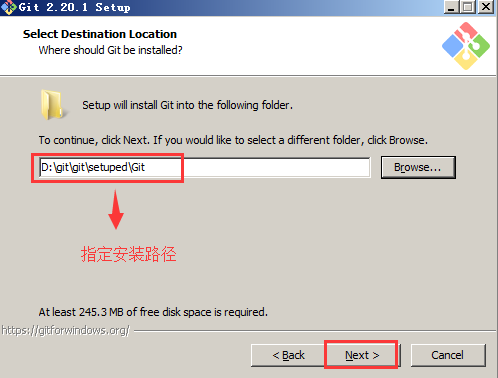
（6）以快照流的方式工作

# 第二章 Git应用

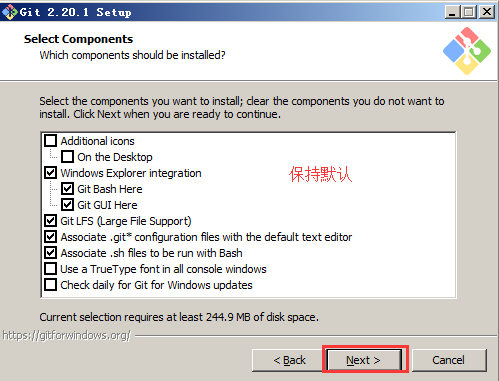
## 2-1 Git安装

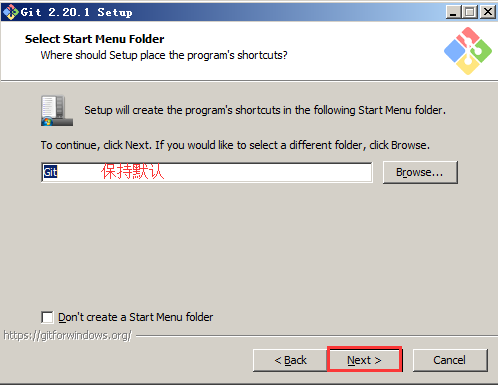
双击安装文件

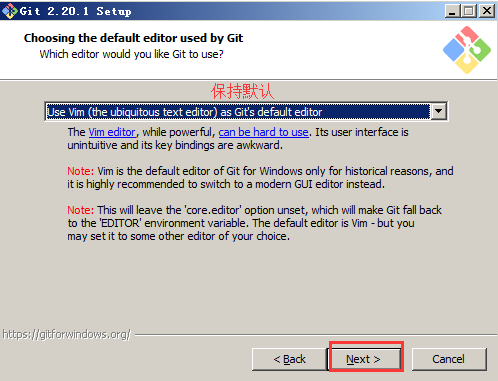


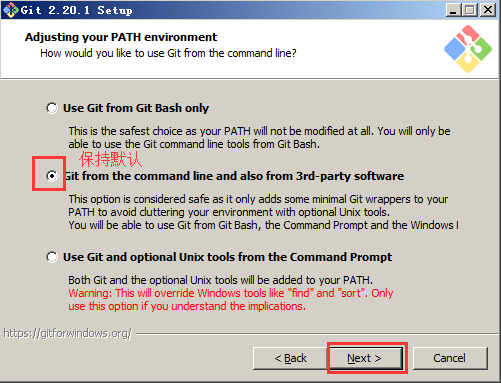


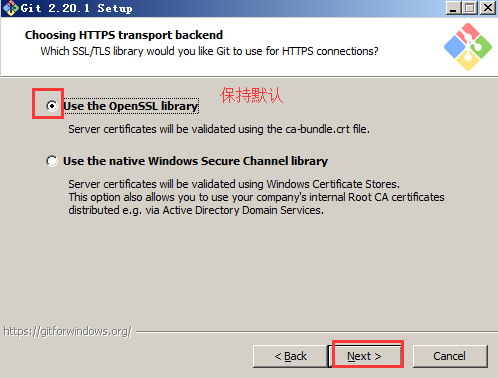
注意安装路径不能有中文和空格

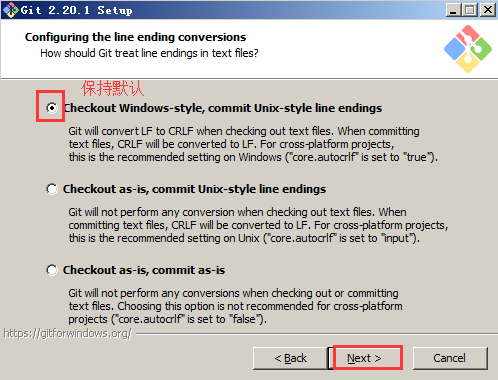


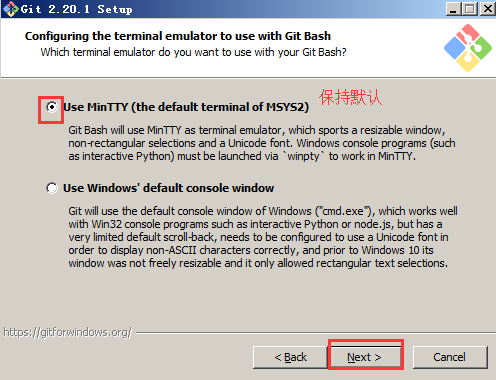


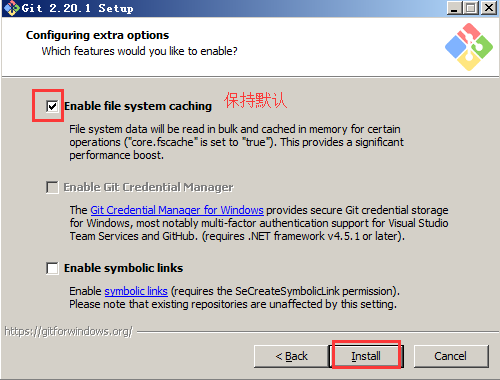


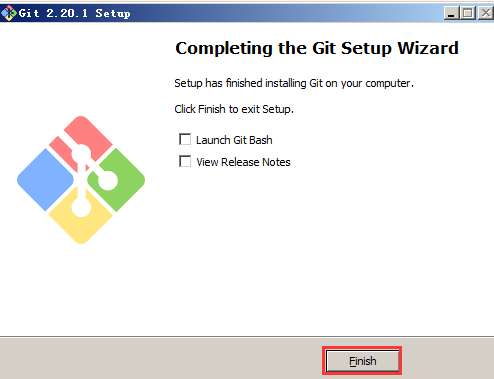




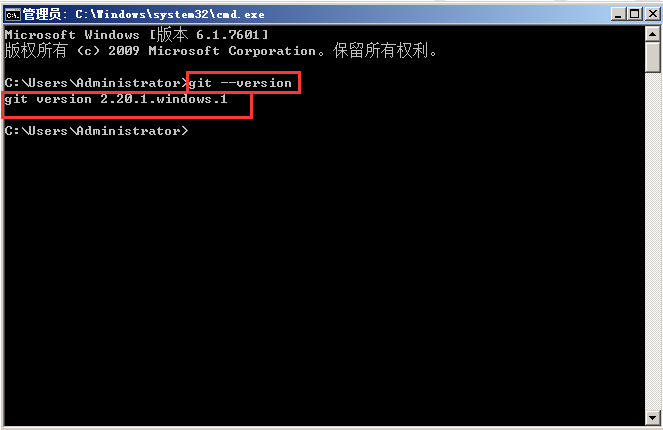






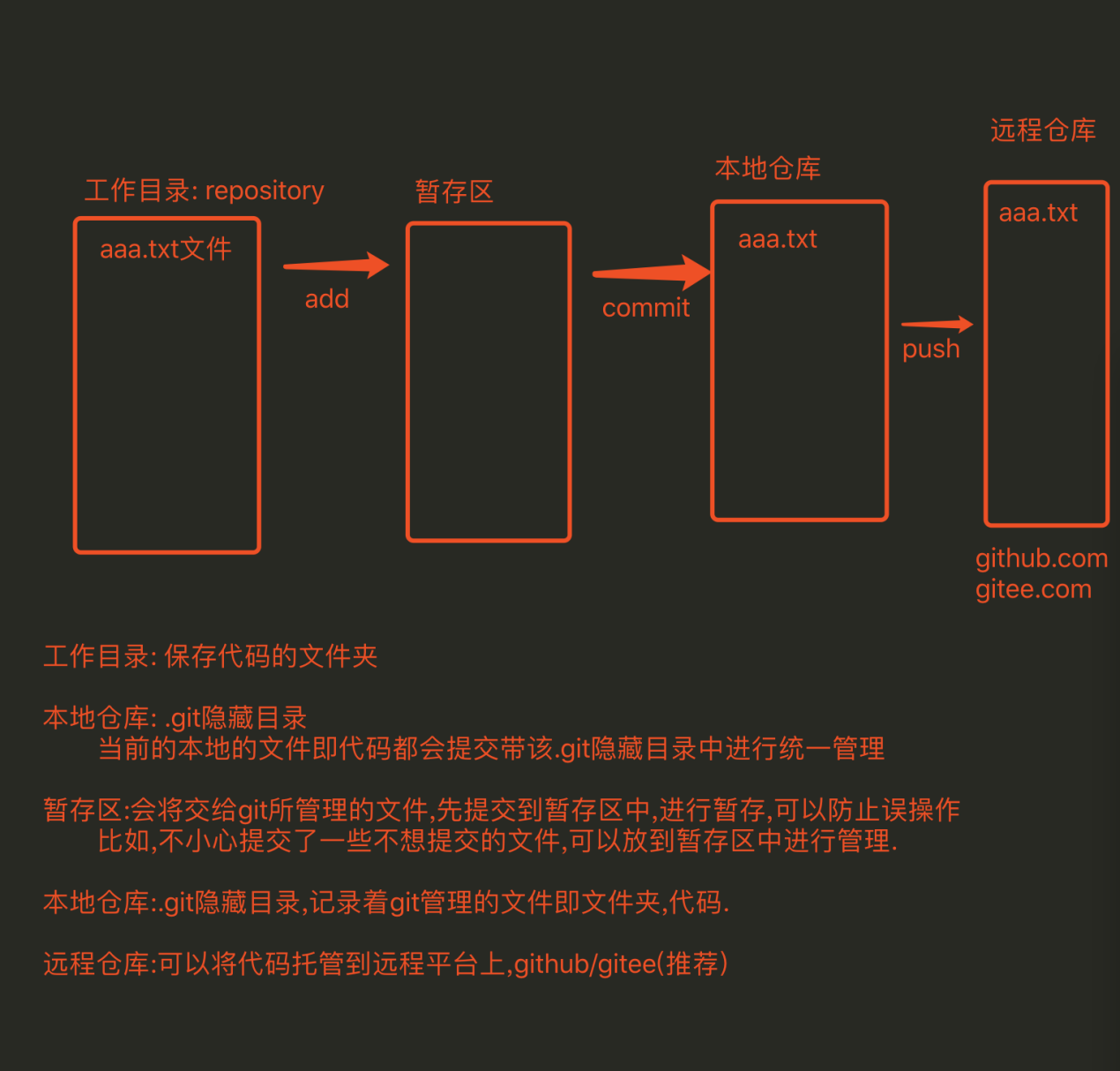


启动cmd命令行工具，以查看版本号的形式验证Git是否已经安装成功



## \*\*2-2 Git准备工作

### \*工作流程图：



### 2-2-1 创建.git目录git init

首先找到一个任意的路径当做本地库目录

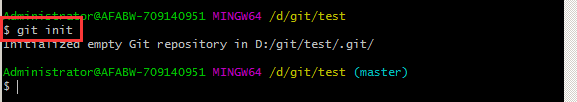
例如：目录D:\git\test

点击右键 --> Git Bash Here

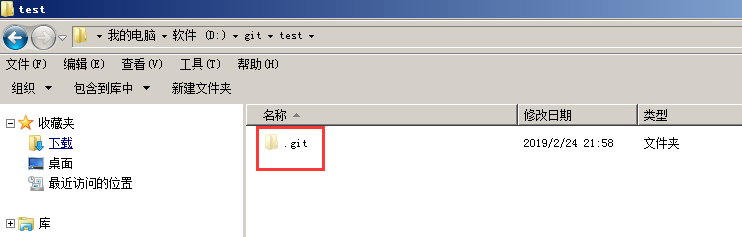


初始化本地库

命令：git init



执行完该命令后，在当前目录会出现 文件夹 .git



**.git目录的出现**表示**本地库初始化成功**（表示**这个.git目录**就是我们的**本地库**了）

注意：.git目录存放的是**和本地库相关的文件，不要修改或者删除**

Git目录说明

工作目录：任意目录下，我们开发代码的目录

暂存区域：.git目录下，作用：有个后悔（返回撤销）的余地

本地仓库：.git目录下，Git存储项目的仓库

### 2-2-2 设置签名git config

新的远程仓库创建成功后，想要将本地仓库的文件上传到远程仓库，首先需要让远程仓库与本地仓库互相识别。

（1）设置签名的作用：**区分**不同开发人员的身份

注意：为Git设置签名**与远程库（代码托管中心）的账号密码**没有任何关系

（2）设置签名命令： git config

本地库级别设置签名方式：

git config user.name zs

git config user.email zs@bjpowernode.com

信息保存位置：./.git/config 文件

系统用户级别设置签名方式：如果**去掉 --global** 参数只对当前仓库有效。

git config --global user.name zs

git config --global user.email zs@bjpowernode.com

~/.gitconfig 文件

eg：

　git config -global，有了这个参数，表示**这台机器上所有的git仓库**都会使用这个配置

　git是分布式版本控制系统，需要填写**用户名和邮箱**作为一个标示

此时打开【C:\Users\Administrator】，会发现Administrator文件夹下多出了“.ssh”的文件夹进入该文件夹后会有两个文件“id\_rsa”、“id\_rsa.pub”选择名称为id\_rsa.pub文件，右击鼠标，打开，并复制里面的内容（不建议用记事本和word打开文件，可以下载软件notepad++，用该软件打开id\_rsa.pub）

优先级按照就近原则：**项目级别优先**于系统用户级别，二者都有时采用项目级别的签名。

## 2-3 Git版本管理

### 2-3-1 文件版本控制方式



将工作目录的代码先提交到暂存区，然后再由暂存区提交到本地仓库

### 2-3-2 文件版本基础管理

#### 一、新建远程仓库+本地仓库：

1. [创建.git目录+设置签名【见前面】](#_2-2-1 创建.git目录git init)
2. [关联远程仓库【见下面】](#_关联远程仓库)

###### 关联远程仓库

打开本地客户端，进入本地仓库（**git@github.com:用户名/库名.git==直接在github中复制链接即可**）  
git remote add origin git@github.com:用户名/库名.git　　//使用git协议，关联远程仓库，推荐此方法  
git remote add origin https://github.com/用户名/库名.git　//或使用https，使用此方法后**每次**git push时需要输入github密码

此时远程仓库与本地仓库可互相识别，可以愉快的上传本地文件到远程仓库了

###### git pull报错，没有correct access rights和Permission denied (publickey)

git@github.com: Permission denied (publickey). fatal: Could not read from remote repository. Please make sure you have the correct access rights and the repository exists.

解决方法：

1. ssh-keygen -t rsa -C “raiceg@163.com”　　//生成本地秘钥）
2. 登录github，点击右上角的头像，选择“settings”，进入设置后，选择“SSH and GPG keys”。点击【New SSH key】，将刚刚复制过来的字串符粘贴到key里（title为名称，可不填），点击Add SSH key

#### 直接拉取现成的远程仓库：git clone url

1）[创建.git目录+设置签名【见前面】](#_2-2-1 创建.git目录git init)

2）git clone url 或者git clone url 别名

如果你想获得一份已经存在了的 Git 仓库的拷贝，这时就要用到 git clone 命令。

git clone 拷贝一个 Git 仓库（包含.git文件夹）到本地（相当于一个仓库），让自己能够查看该项目，或者进行修改。

这将在当前目录下创建一个名为 “libgit2” 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。 如果你想在克隆远程仓库的时候,自定义本地仓库的名字,

你可以使用如下命令:

$ git clone https://github.com/libgit2/libgit2 **mylibgit**

　　这将在本地创建的仓库名字变为 mylibgit。

Git 支持多种数据传输协议。 可使用 https:// 协议，也可以使用 git:// 协议或者使用SSH 传输协议。

#### 已经拉取过代码了：

直接从（2）开始，跳过关联远程仓库这个步骤。

#### git pull

第1步：**一定先**同步远程仓库代码：git pull

提交代码第1步：git pull 同步远程仓库代码到本地

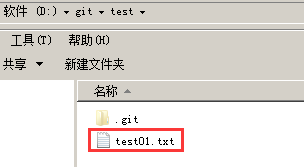
git add / git commit代码之前**首先git pull**，需先**从服务器上面拉取代码**，**以防**覆盖别人代码；如果有冲突，1）**先备份自己的代码**，2）git checkout下**远程库里最新的的代码**，**将自己的代码合并进去**，**3）然后再提交代码**。

#### **git status：**查看工作区、暂存区状态

测试：

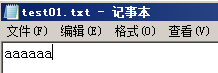
（1）在一个已经初始化的目录下（例如test目录下）

创建文件test01.txt



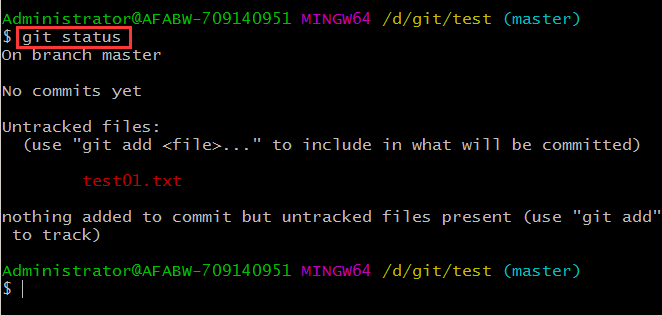
（2）在里面写点初始内容

例如：



（3）通过Git Bash Here打开Git的命令行工具

执行命令git status，查看当前状态：查看效果



文件标识**红色**表示文件**还没有被添加到Git中**进行管理，就是你修改的文件。

接下来将文件添加到暂存区

#### git add . 或git add xxx

情形一： git add .  就可以**把你的内容全部添加**到本地git缓存区中

情形二：如果你git status 查看了当前状态**发现有部分文件你不想提交**，那么就使用git add xxx(上图中的红色文字的文件链接)  就可以提交部分文件到本地git缓存区。

命令：git add 文件名1 文件名2 ...

**git add [file name]：**将工作区的信息（变化）添加到本地暂存区



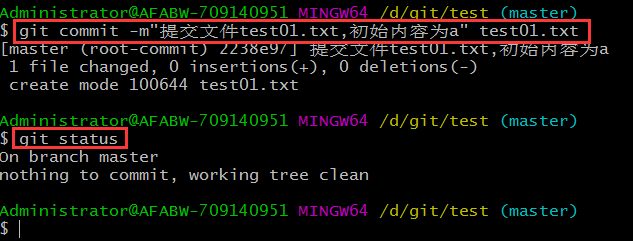
再次观察效果，文件变成绿色标识。表示新提交的。

#### git commit -m "msg" [file name]

**git commit -m "msg" [file name]：**将暂存区的内容提交到本地库，推送代码到本地git库

情况一：提交暂存区到本地仓库中:git commit -m "提交代码备注"

情况二：提交暂存区的**指定文件**到仓库区：命令：git commit **文件名** -m "提交代码备注"

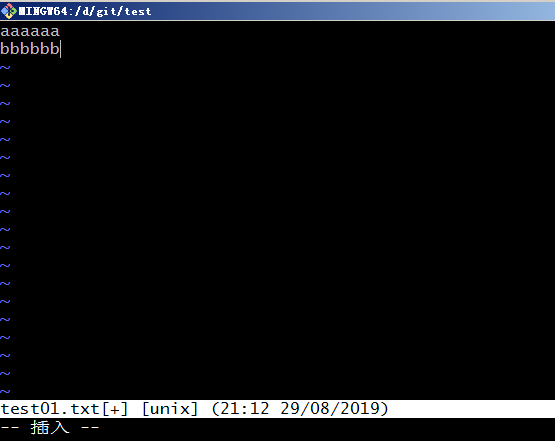


另：如果你觉得 **git add 提交缓存的流程**太过繁琐，Git 也允许你用 -a 选项跳过这一步。命令格式如下git commit -a：

a 参数设置修改文件后**不需要执行 git add 命令**，直接来提交

将文件提交到本地库之后，查看状态，没有任何文件展现，说明工作区，暂存区，本地库中的文件信息处于同步（相同）状态。

将本地库中的文件进行修改，添加bbb





查看状态，文件又变红了，说明工作区有新内容，待添加到暂存区

重复首次操作，继续add，然后继续commit

最后查看状态，确保工作区，暂存区，本地库再次同步。

#### git push提交本地代码到远程仓库：

提交代码第5步：git push <远程主机名> <远程分支名>  把当前提交到git本地仓库的代码推送到远程主机的**某个远程分支**之上。

命令：git push

git push -u origin master

#### git log：查看本地库更新历史记录

****

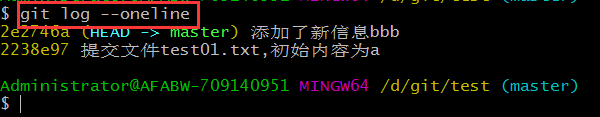
观察到我们一共为本地库，有提交过两次记录的记载

我们可以观察到两次提交，分别对应两个40位的哈希值，这就是我们提交后记录的版本号

这种形式查看历史日志是最原始的查看方式

接下来我们也可以通过其他两种方式来查看历史

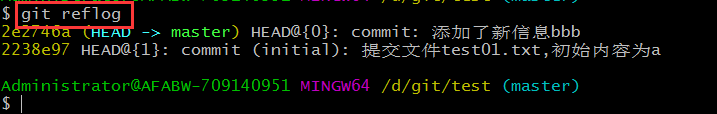
###### git log --oneline：查看本地库更新历史记录（简化版）

****

这样做可以很简单的展现历史信息

这种形式展现的哈希值为简化形式，通过也简化了提交的日期时间以及作者。

#### **git reflog：**查看本地库更新历史记录（展现HEAD指针）



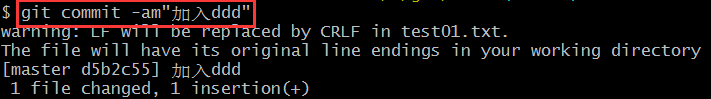
这种形式可以展现HEAD指针编号位置，我们以后可以通过该位置定位到我们想要的版本历史。

### 2-3-2 版本前进后退管理

准备工作，在原有aaa和bbb的基础上

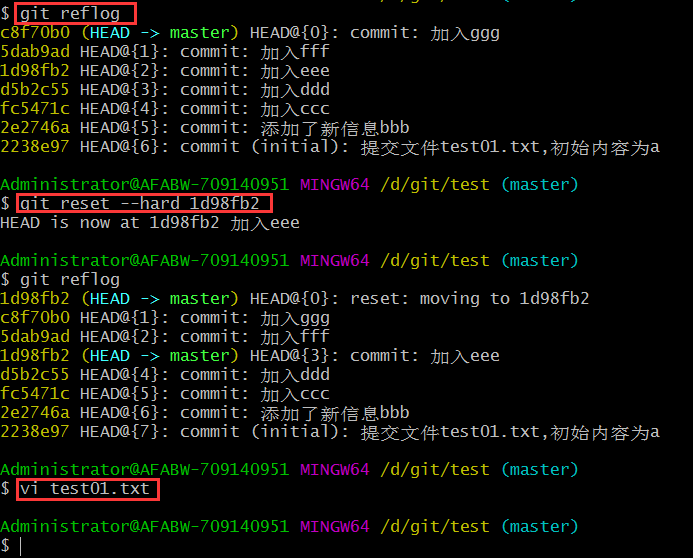
新增历史记录ccc、ddd、eee、fff、ggg

可以使用git commit -am"msg"进行**添加和提交**一次性操作



准备工作后，开始前进和后退的操作

**git reset --hard [局部索引值]：基于索引值的操作**

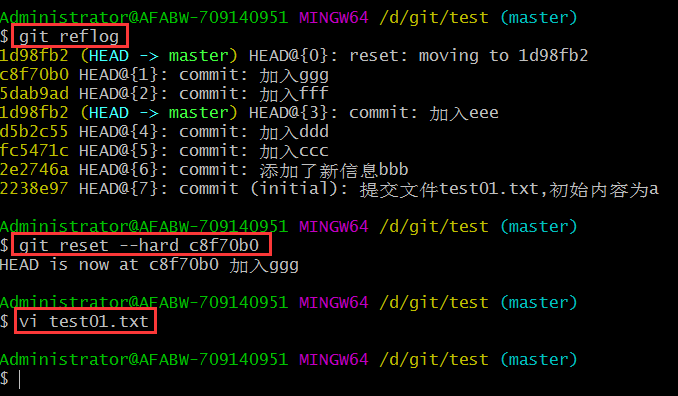


通过以上操作，先以观察哈希值的方式查看历史

通过git reset --hard eee的哈希 回到历史

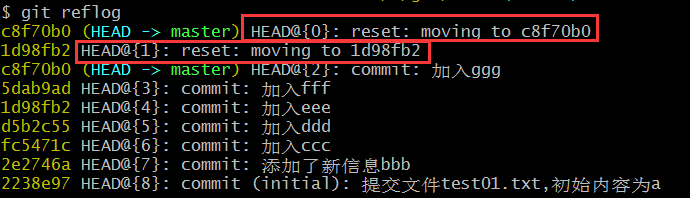
最后观察vi test01.txt 文件是回到了eee的历史版本

同时可以通过以上的方式，回到最新历史版本ggg



观察又回到了ggg的版本

在次观察历史



观察到前进和回退的操作也会被忠实的记录下来

git reset --hard HEAD^ :表示后退操作

一个^表示后退一步，N个^表示后退N步

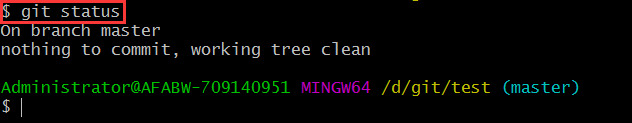


观察以上操作，我们通过后退的方式，后退到了fff

在进行以下操作前，可以先回到最新版本ggg

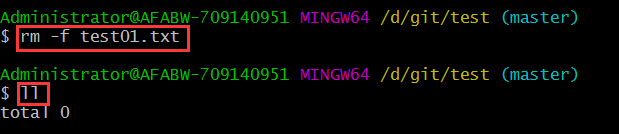
### 2-3-3 删除文件并找回的方式

前提：删除前，**文件存在时的状态**提交到了本地库

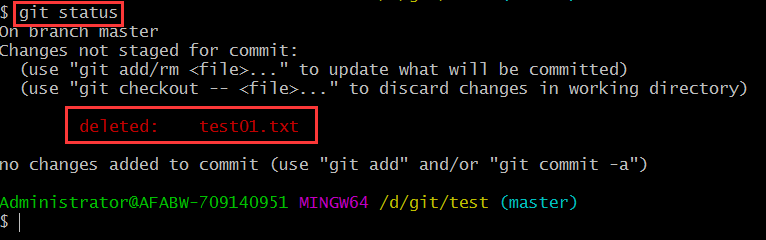


通过以上方式查看状态，表示工作区、暂存区和本地库处于同步状态。

接下来，将test01.txt文件删除



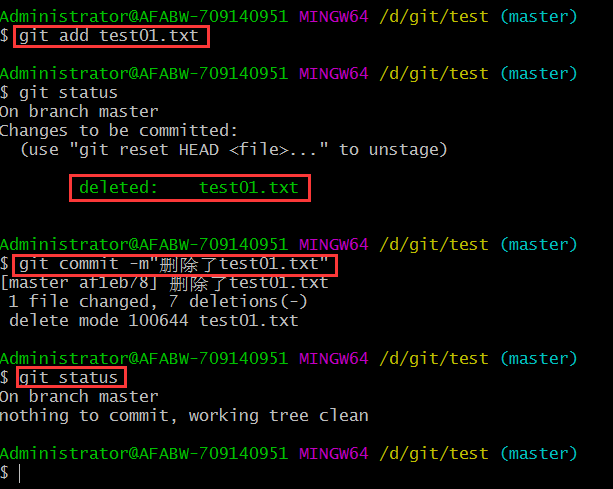
查看test文件夹状态



通过以上观察，表示工作区中少了文件，和本地库不同步了

此时根据提示，我们可以将这次的删除操作添加到暂存区，并提交的本地库。

注意：删除文件后的git操作，和添加文件后的git操作方式是一致的，也是先add，再commit



我们现在相当于彻底将这个文件删除了，最新提交到本地库的历史也是处于将这个文件删除的状态了。

现在我们要将该文件找回！

由于我们以前提交过的每一个历史版本都被忠实的记录下来的，所以我们现在回到历史就可以将文件找回了。（**文件是被删除了，但是历史永远不会被删除**）

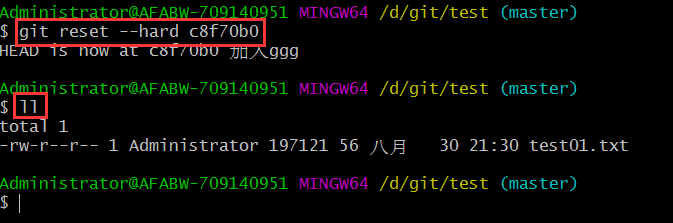
##### 1）git reflog：查看历史版本



通过以上历史版本我们观察到，最后一个版本af1eb78是删除了文件，但是我们可以看到最新的加入ggg的版本为c8f70b0，此时还是有test01.txt这个文件的，我们可以回到c8f70b0这个版本

##### 2）git reset --hard [版本号]

通过我们以上学习过的 git reset --hard [版本号] 就可以实现了



通过以上操作，观察test文件夹中又有test01.txt这个文件了

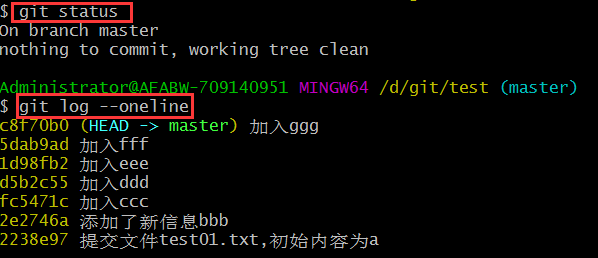
### 2-3-4 比较文件的方式

##### 1）git diff 文件名

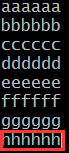
git diff 文件名 : 将**工作区中**的文件和**暂存区**进行比较

测试前，首先确保工作区，暂存区，本地库处于同步状态

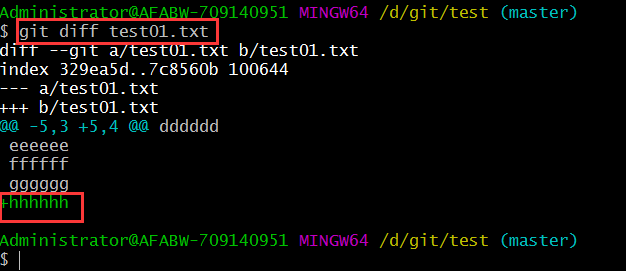
观察当前最新版本（例如以下最新版本为ggg）



工作区新增hhh



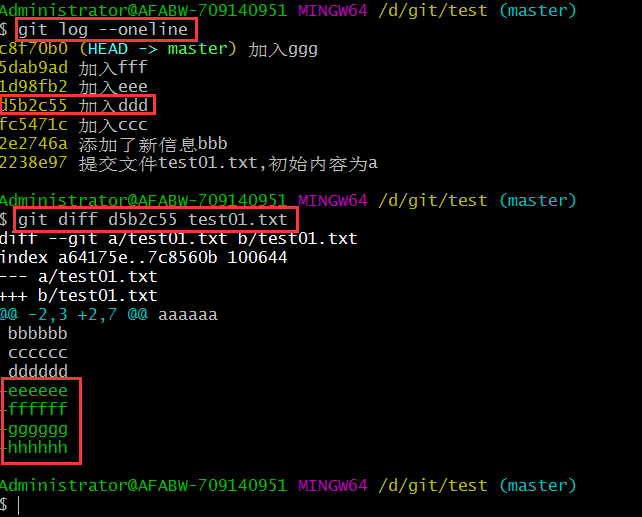
接下来比较工作区中的该文件和暂存区的差别



通过观察，看出工作区比暂存区多了hhh

还可以与本地库的**历史版本**进行比较

git diff 本地库中历史版本 文件名

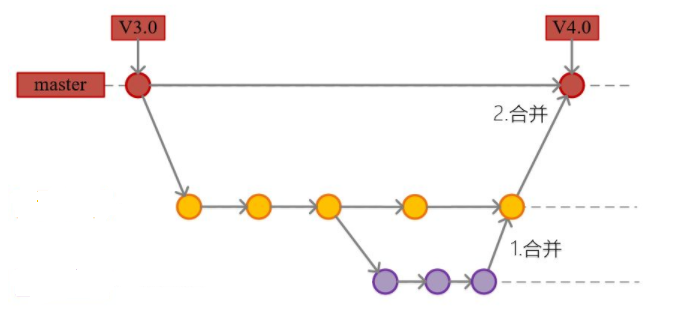


## \*\*2-4 Git分支管理

### 2-4-1 什么是分支管理

在版本控制过程中，使用多条线同时推进多个任务就是分支管理

如下图所示：



### 2-4-2 分支管理的好处

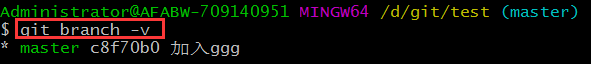
（1）同时并行推进多个功能开发，提高开发效率

（2）各个分支在开发过程中，如果某一个分支开发失败，不会对其他分支有任

何影响。失败的分支删除重新开始即可。

### 2-4-3 分支管理操作方式

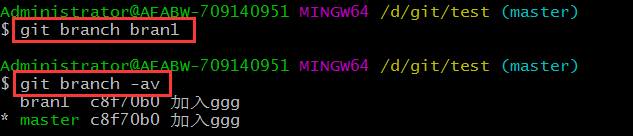
#### 1）git brach -v 查看分支



观察到我们现在只有一个默认的分支（系统提供的**主干分支**），**就是master**

#### 2）git branch 分支名 创建分支

我们现在来创建一个分支bran1



创建完毕后，再次查看分支，表示现在共有两个分支master和bran1

注意：由于barn1这个分支是由master这个分支创建出来的，所以默认指向同一个指针ggg，即分支的起点也是ggg

master**绿色高亮**表示当前操作，使用的仍然是master分支

#### 3）git checkout分支名，切换分支



切换完再次查看分支，bran1分支绿色高亮显示，表示我们现在操作的是bran1这个分支

### 2-4-4 分支合并以及解决冲突

#### （1）合并分支git merge 源分支

1）**首先**我们**切换到bran1分支上，在test1.txt上新增信息iii，然后add和commit**

在bran1分支上，我们看到了iii

如果切换回master分支上，我们是看不到iii的（iii是bran1做的操作，所以看不到）

此时master分支如果想要看到iii，就必须要和bran1分支进行合并

分支的合并不会删除某一个分支，而是对于信息进行对称整合

2）我们切换到master分支上

执行命令 git merge bran1 就会将bran1中的信息进行整合



这时我们再观察cat test01.txt就会有最新的iii了

#### （2）合并分支过程中冲突的解决方式

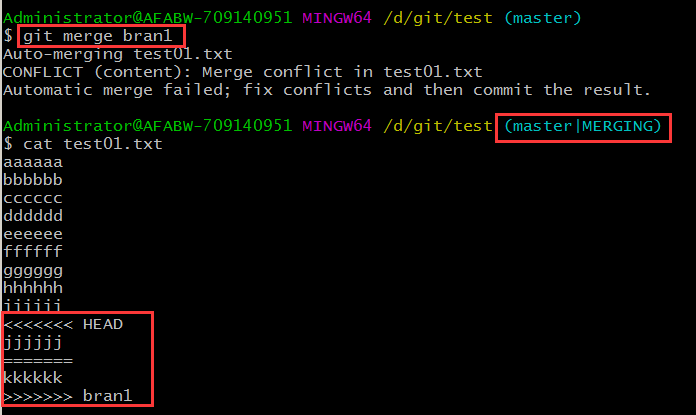
此时master分支和bran1分支上都是最新的iii

1）在master分支上**新增jjj**，执行add和commit

2）然后切换到bran1分支上，**新增kkk**，执行add和commit

3）然后切换到master，合并分支，例如我们还是使用master合并bran1分支

此时就会产生冲突

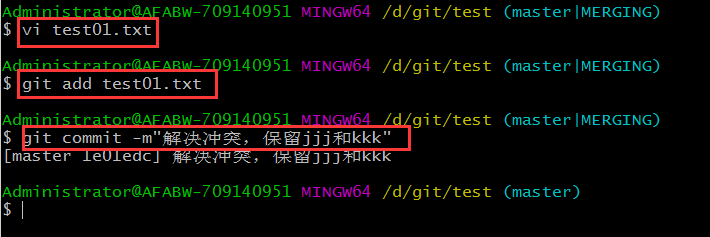


冲突的解决方式：

首先编辑test01.txt文件，保留有用信息（协商解决），**删除（作为冲突标识的）特殊符号**

执行git add test01.txt

执行 git commit -m"日志信息"



这样就成功的解决了冲突

# 第三章 GitHub应用

## 3-1 GitHub概述

### 3-1-1 GitHub简介

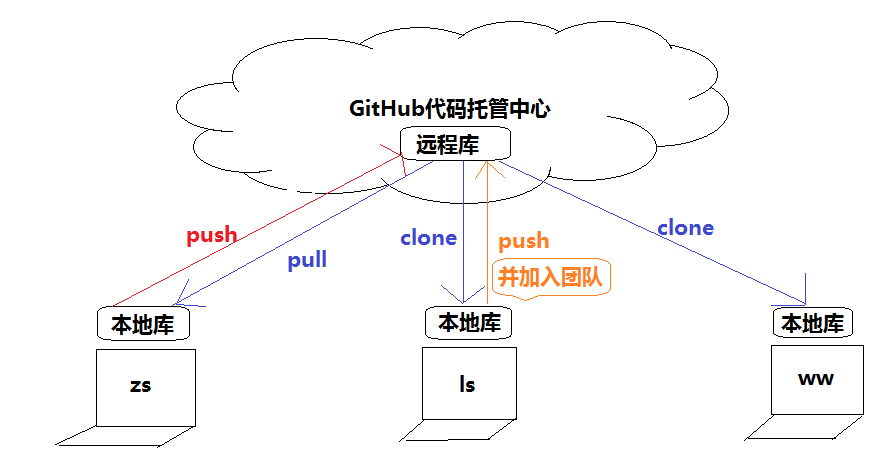
GitHub是一个面向[开源](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90/20720669" \t "_blank)及私有[软件](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6/12053" \t "_blank)项目的托管平台，因为只支持Git 作为唯一的版本库格式进行托管，故名GitHub。

2018年6月4日，微软宣布，通过75亿美元的股票交易收购代码托管平台GitHub。

2018年10月26日，微软以75亿美元收购GitHub交易已完成。同年10月29日微软开发者服务副总裁奈特·弗里德曼(Nat Friedman)将成为GitHub的新一任CEO。

### 3-1-2 GitHub流程概述

团队协作开发方式如下图所示：



### 3-1-3 GitHub官网

https://github.com/

官网首页就是注册页面

## 3-2 GitHub网站应用

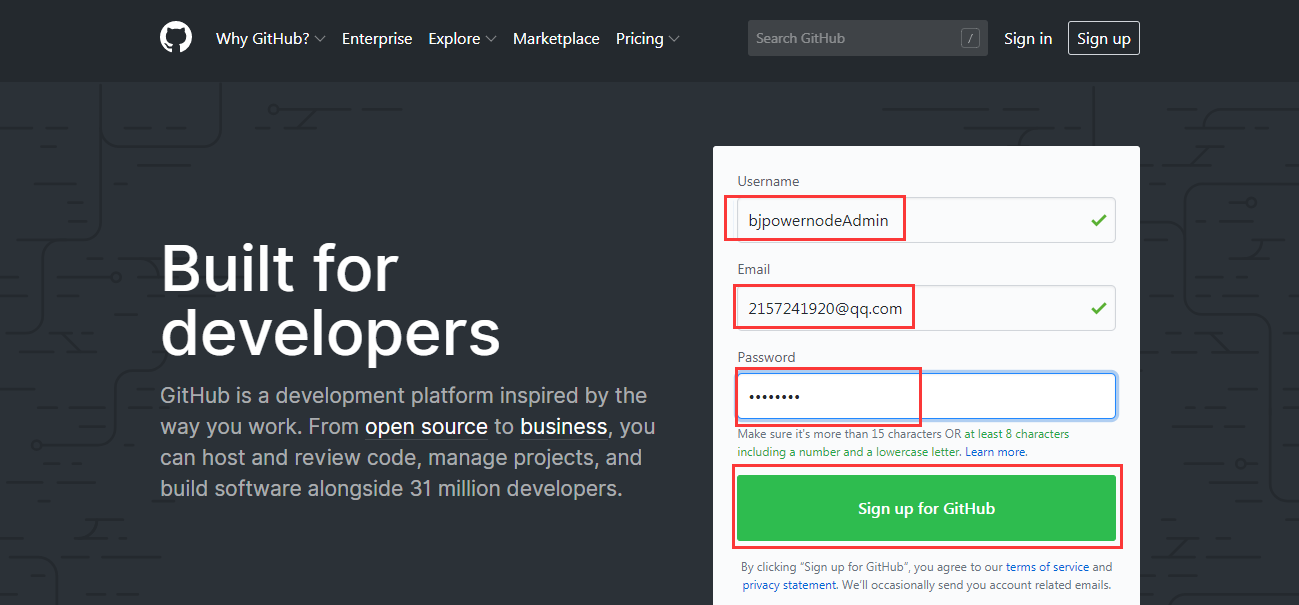
### 3-2-1 GitHub注册流程

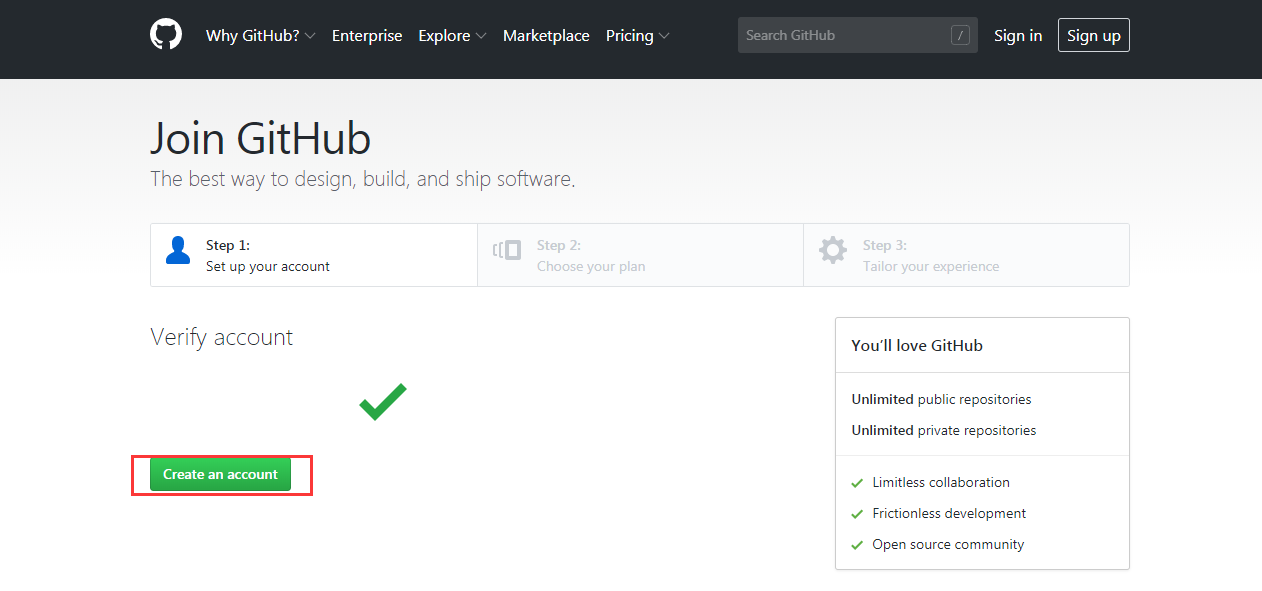
注意：GitHub网站界面更新较为频繁，所以以下实战截屏可能不代表当前体验，根据实际界面实际操作即可。

打开官网

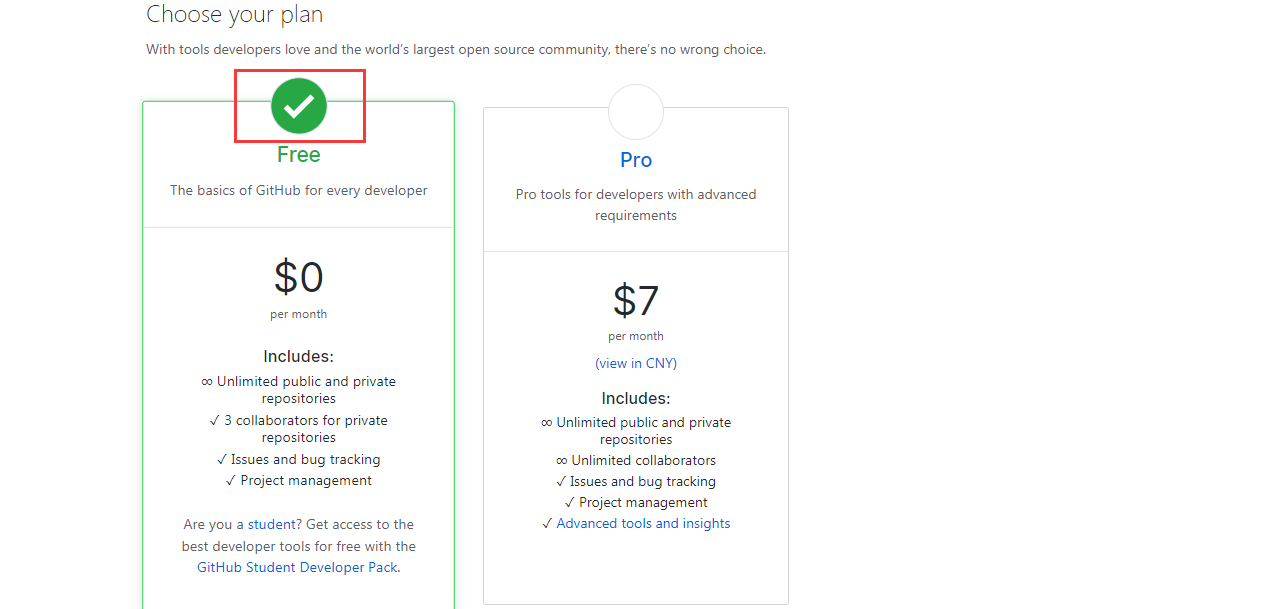
填写注册信息

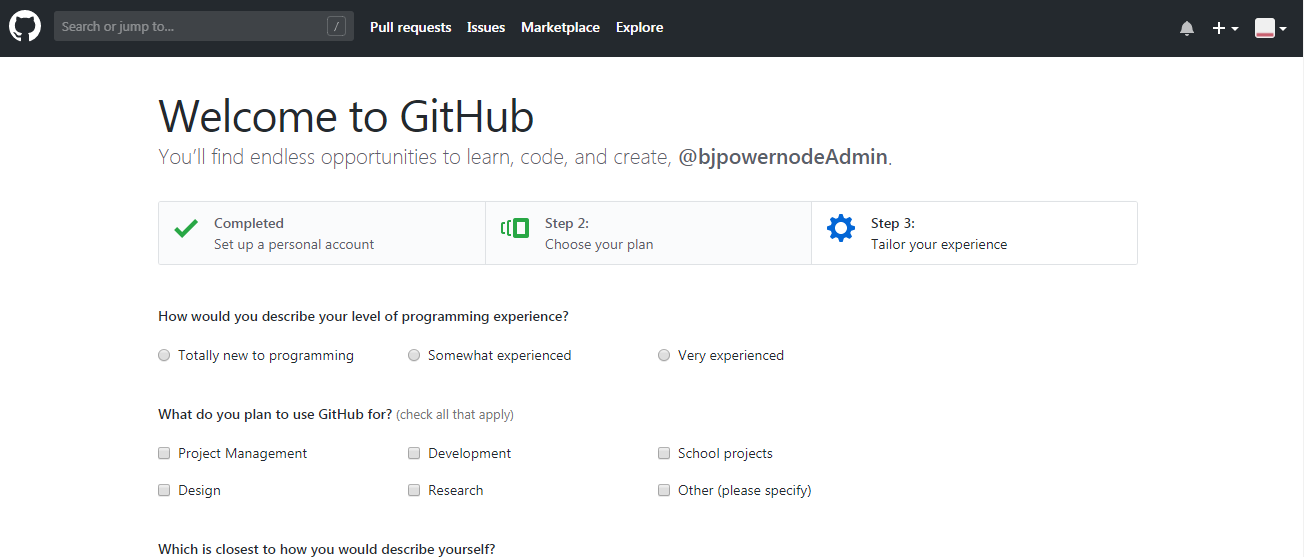
**注意：此处必须使用真实邮箱**







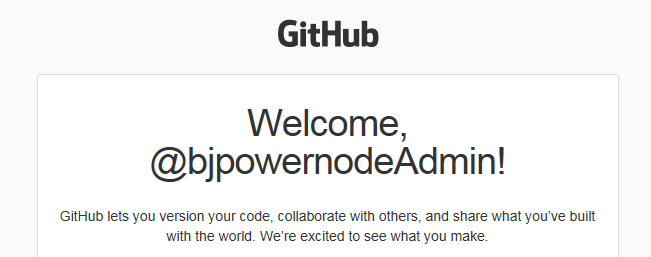




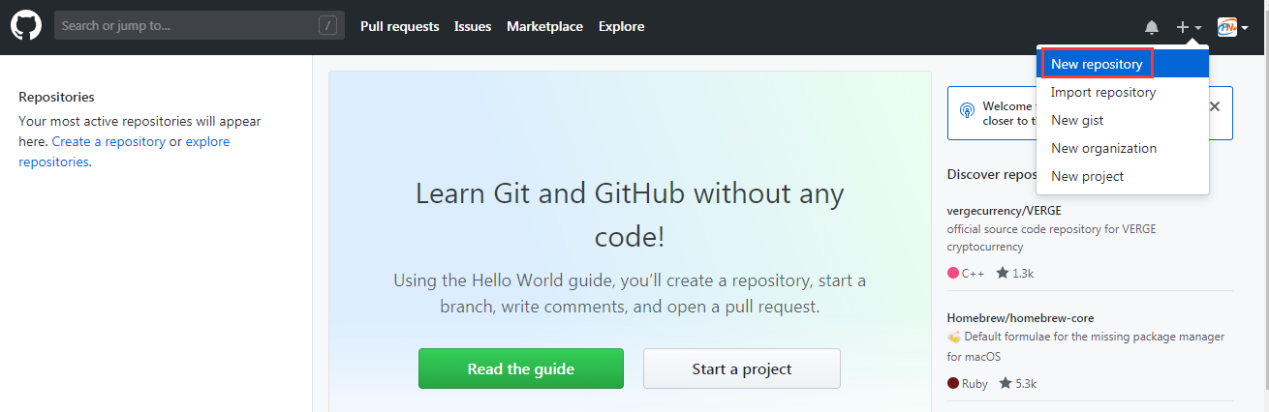
注意：注册完毕后，会收到GitHub为我们发送的邮件

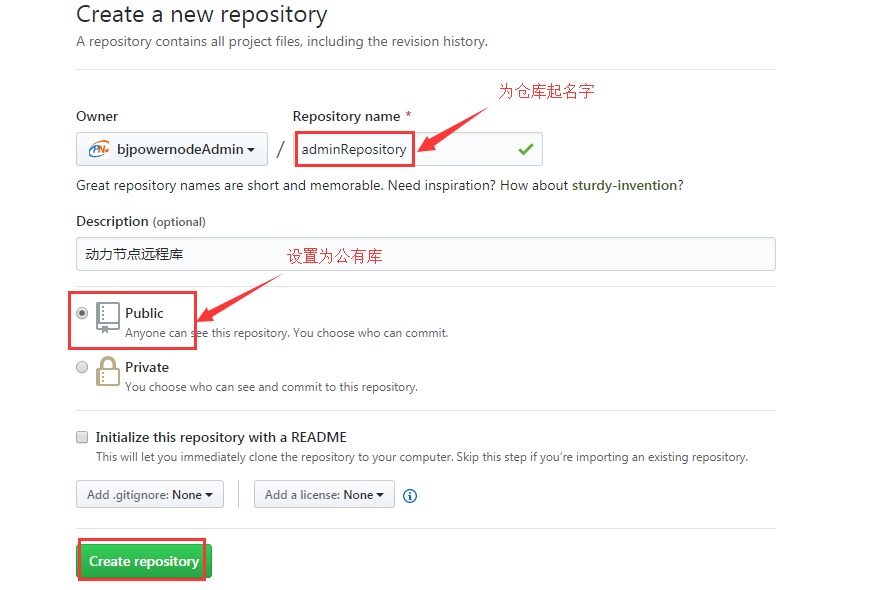
我们打开邮件，点击Verify email address

收到欢迎邮件表示完成注册

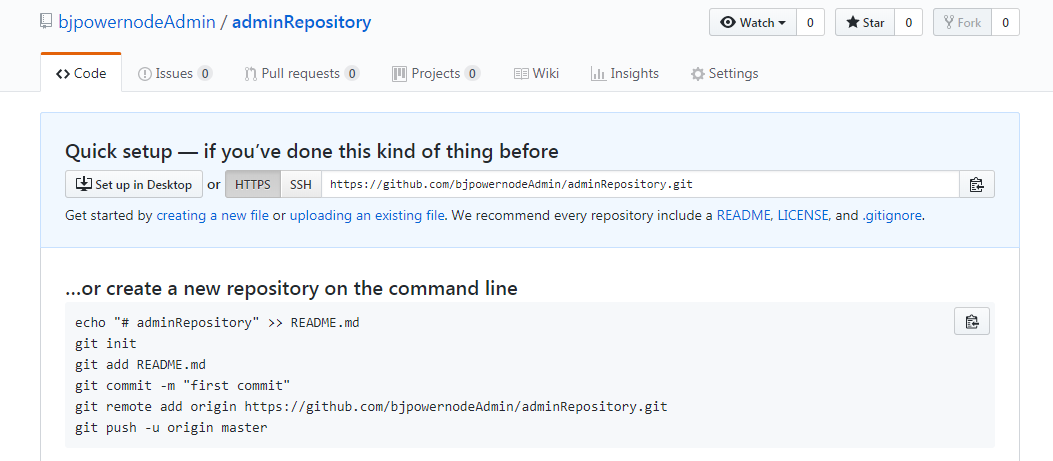


### \*\*3-2-2 创建远程库

****

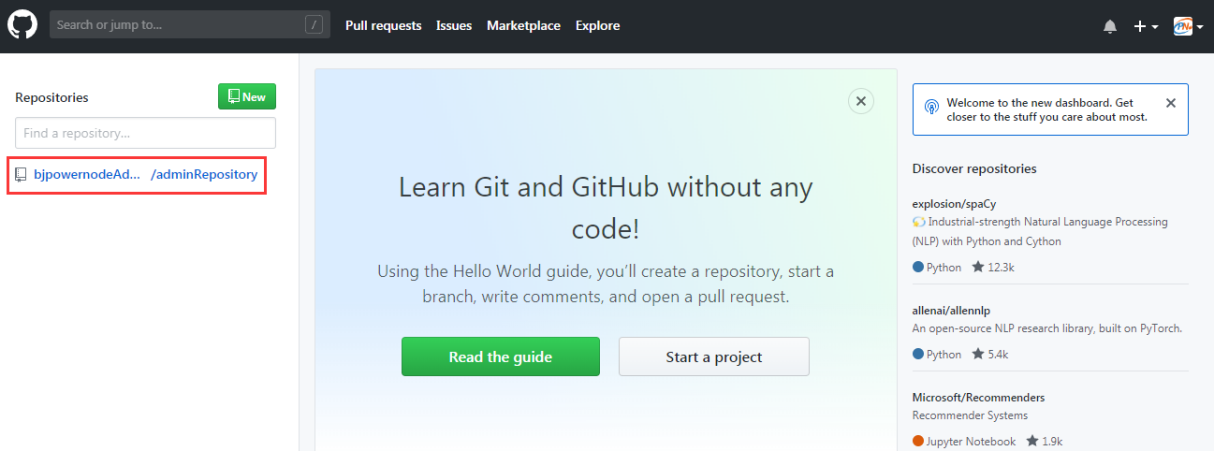


看到以下远程库信息页面证明远程库已经创建成功



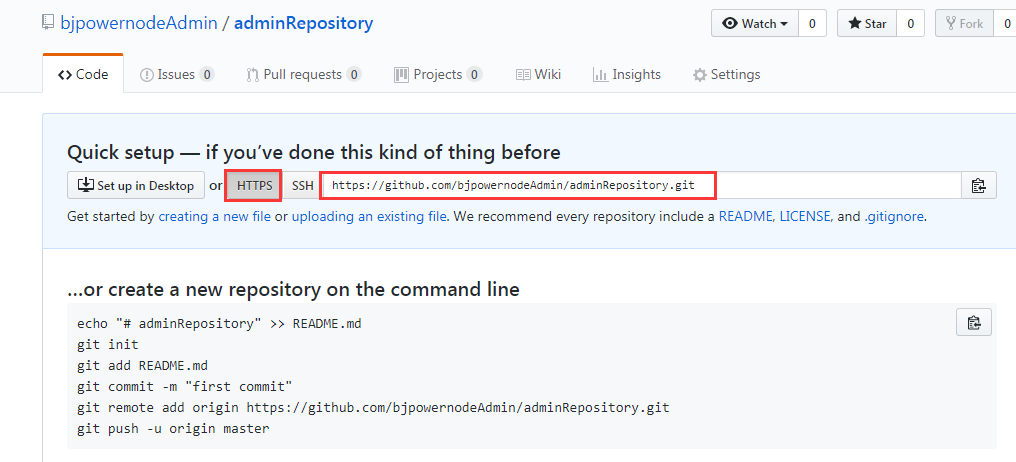
如果是刚登录进来，欢迎页面的左侧也会看到远程库的菜单项

点击左侧菜单中的远程库，也会进入到该远程库信息页面

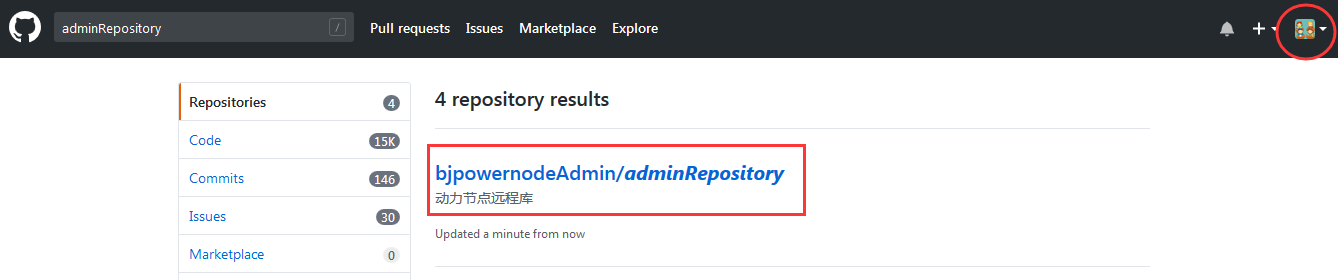


观察如下详细信息页面

点击HTTP，会得到访问远程库的地址

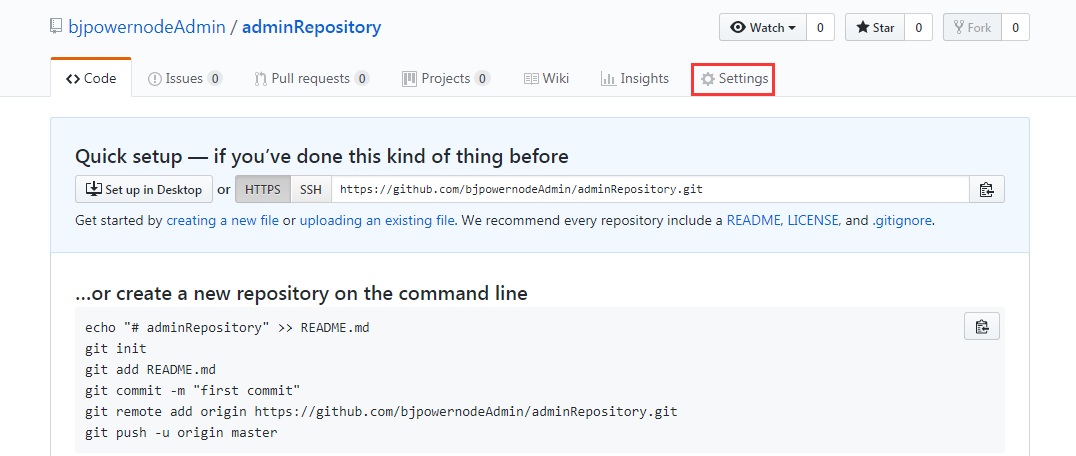


使用另一个账号登录，可以搜索到别人创建的远程库

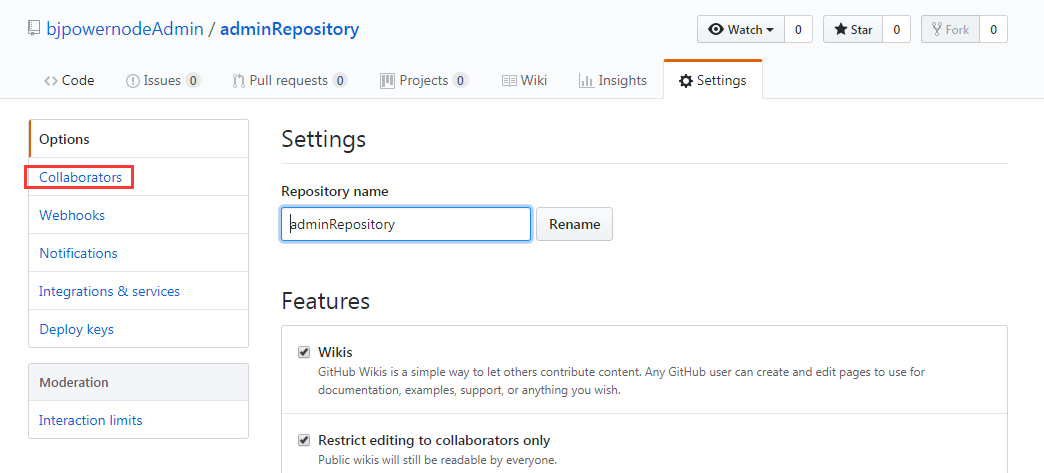


### \*\*3-2-3邀请其他GitHub用户加入开发团队

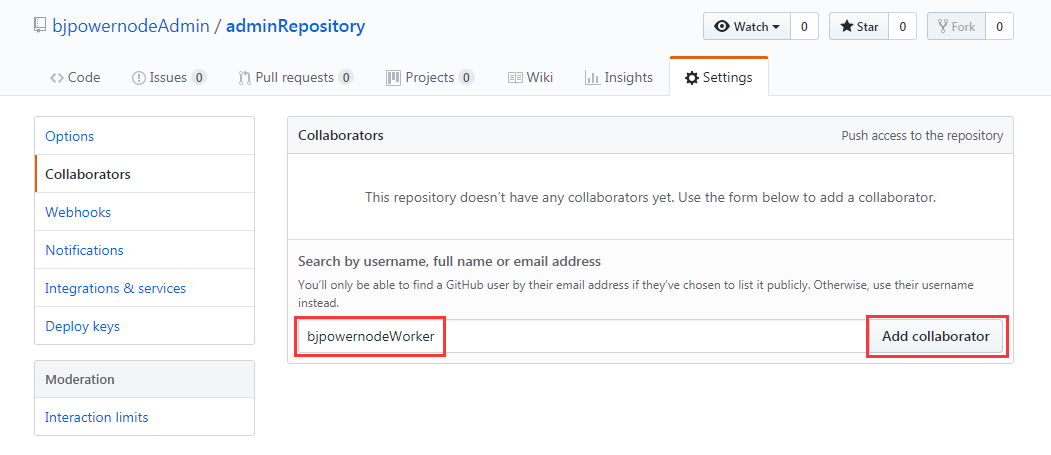
打开远程库信息页面，点击settings

****

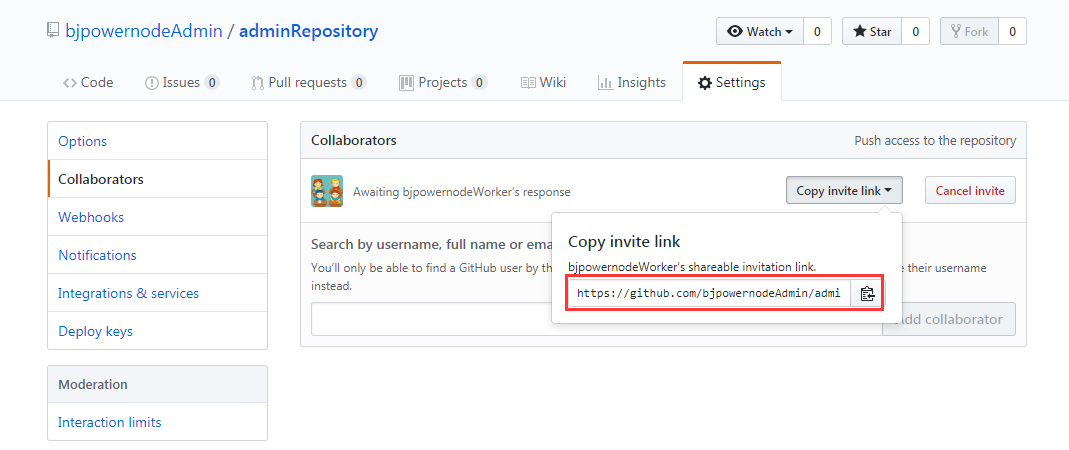
点击Collaborators



输入被邀请人账号，点击Add collaborator



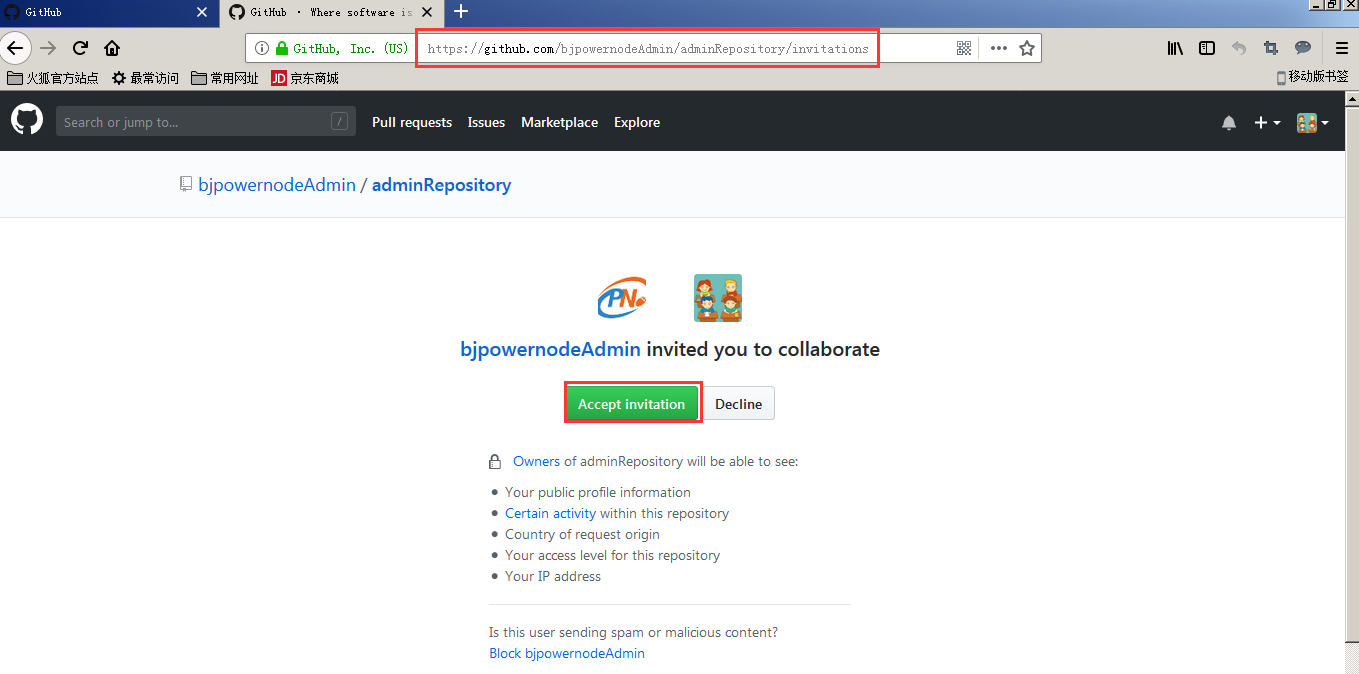
复制邀请链接



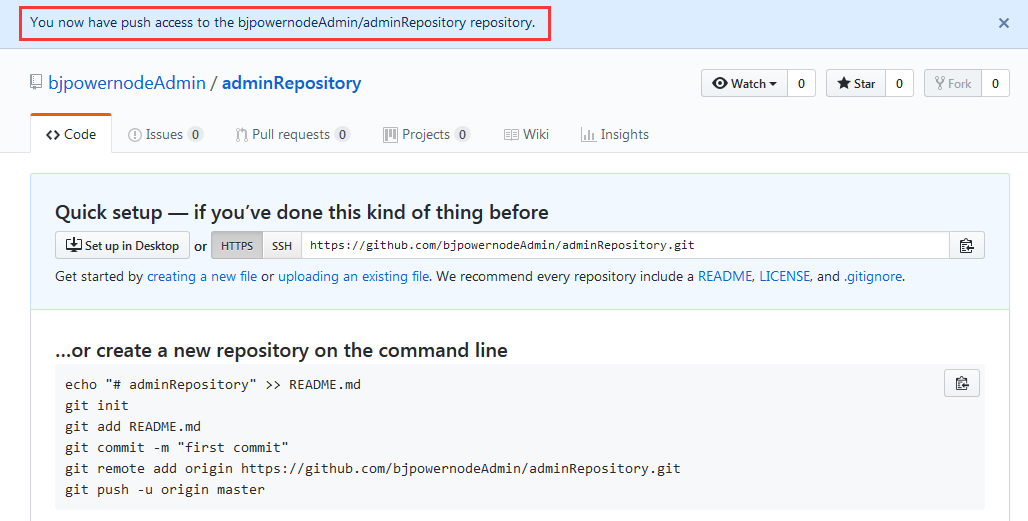
登录被邀请账号

打开浏览器新窗口，在地址栏访问刚刚复制的链接

点击Accept invitaion 接受邀请



变为团队成员后可以推送代码到远程库了

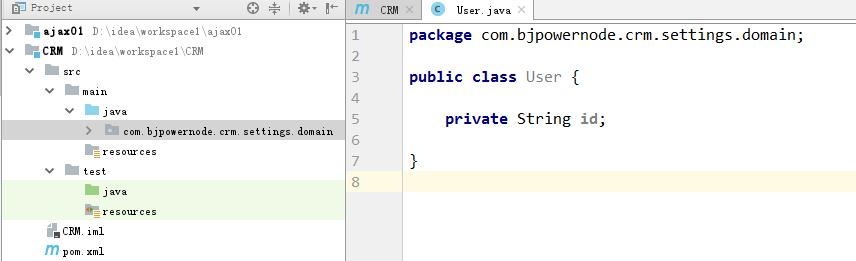


# 第四章 IDEA对于Git&GitHub的支持

## 4-1 IDEA对GitHub和Git的基本配置

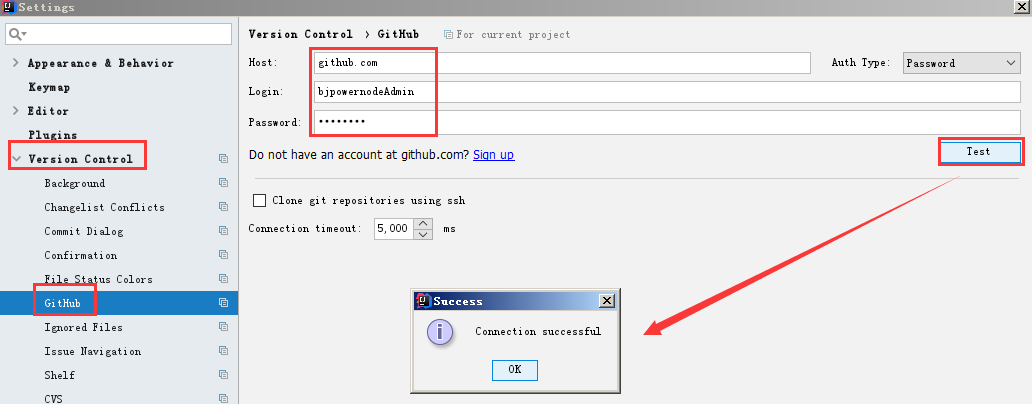
案例演示

首先建立一个演示项目（web项目即可），然后建立一个User类，里面写上初始测试内容



打开settings --> Version Control --> GitHub

填写GitHub网址，账号，密码，然后点击Test测试



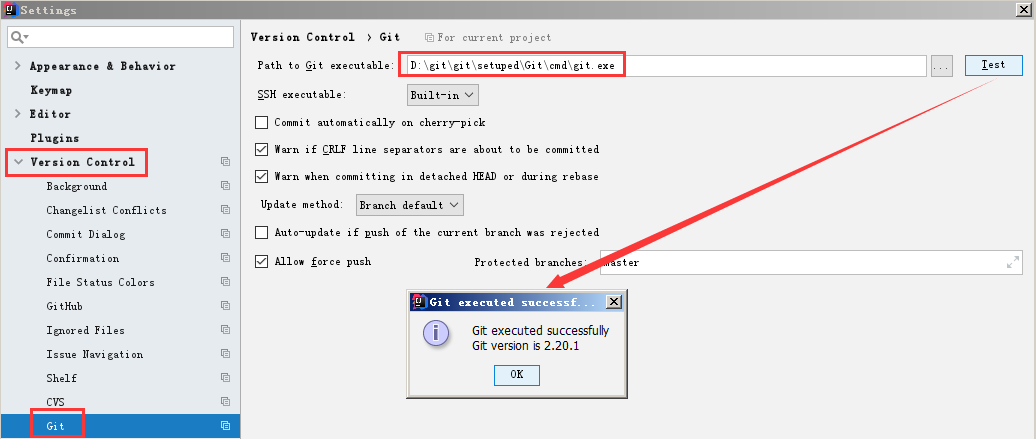
上述测试成功后，配置Git

打开settings --> Version Control --> Git

第一栏文本框会自动识别到本地Git安装路径下的git.exe

点击测试

其他项维持默认即可

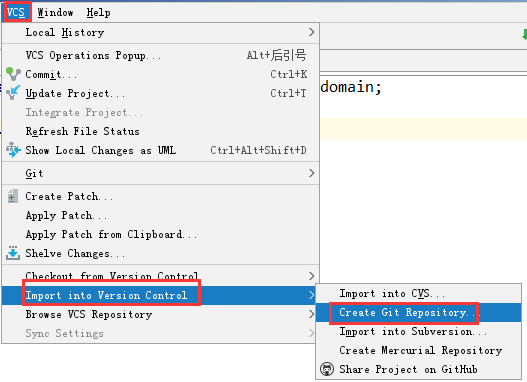


以上就是IDEA对于GitHub和Git的基本配置

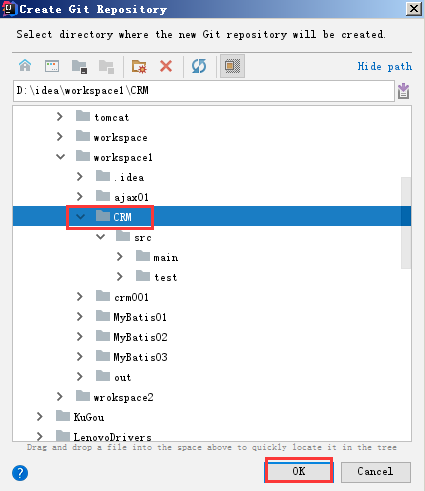
以上配置完毕后，我们来操作项目

## 4-2将IDEA项目push（推送）到GitHub

创建本地库

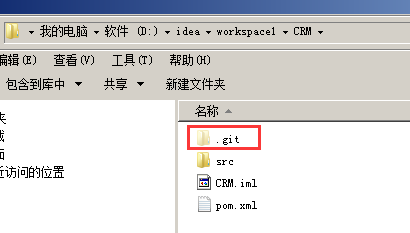


选中当前项目作为本地库



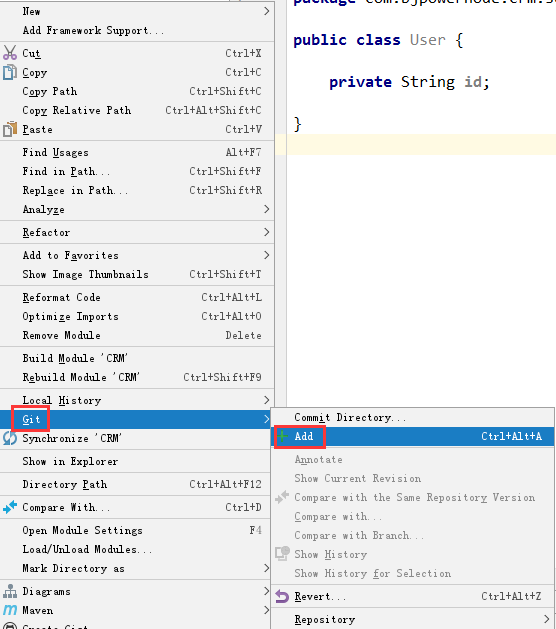
以上操作之后，会在当前项目的本地看到.git文件夹

说明当前项目文件夹作为本地库存在



将项目提交到暂存区

项目右键 --> Git --> Add

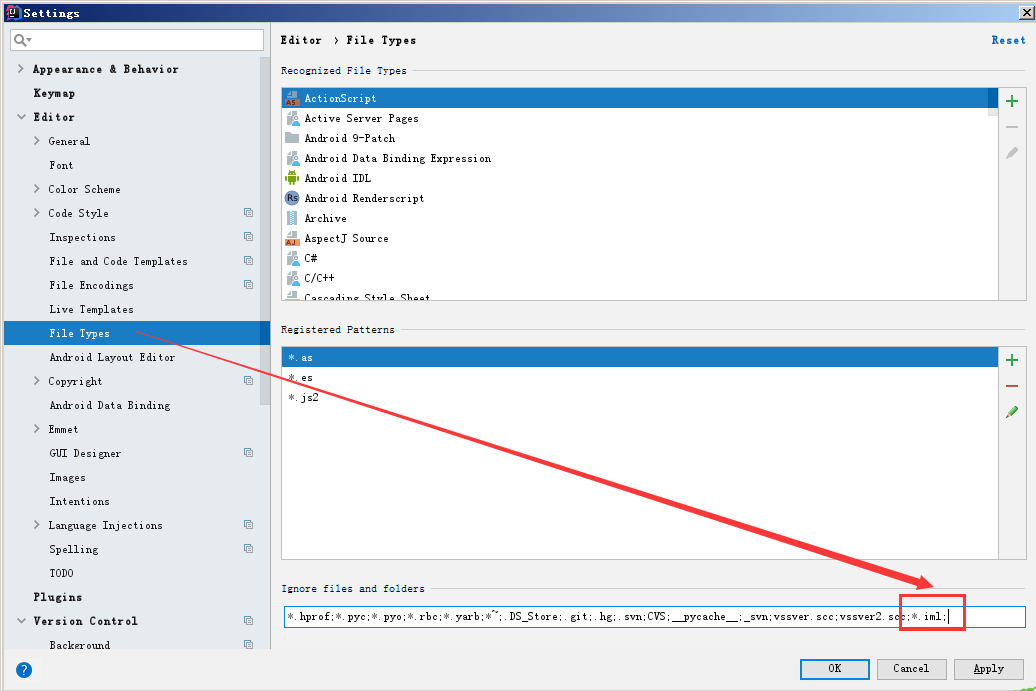


将项目提交到本地库

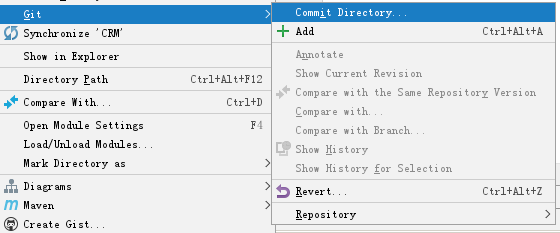
首先我们在提交前，需要忽略掉.iml文件

settings --> Editor --> File Types

添加\*.iml

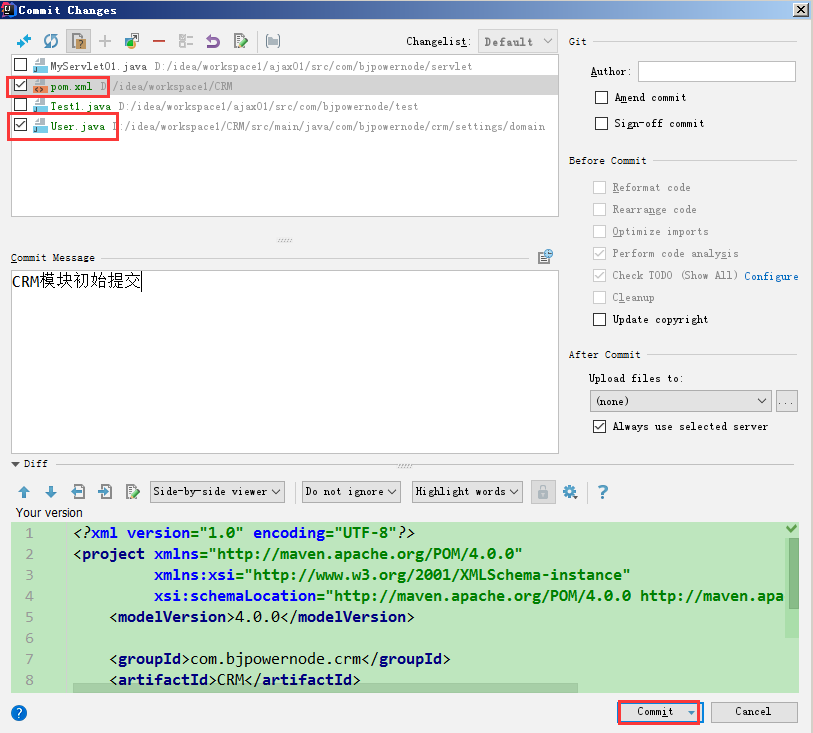


项目右键

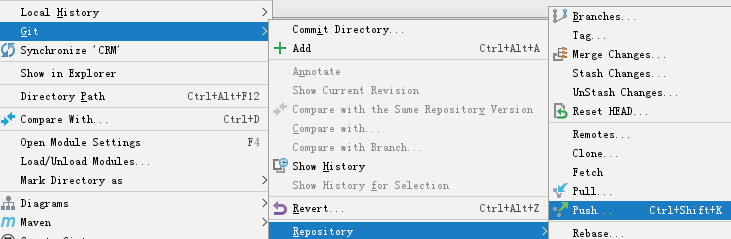


选中刚刚模块中写好的未提交过的文件

点击Commit提交



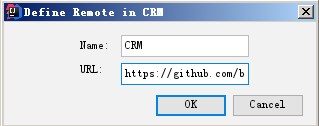
将本地库中的文件上传到远程库



第一次提交，需要先写入提交的GitHub库的名字，以及上传地址



在GitHub上创建一个新仓库CRM



填写完以上信息后，推送项目到GitHub远程库

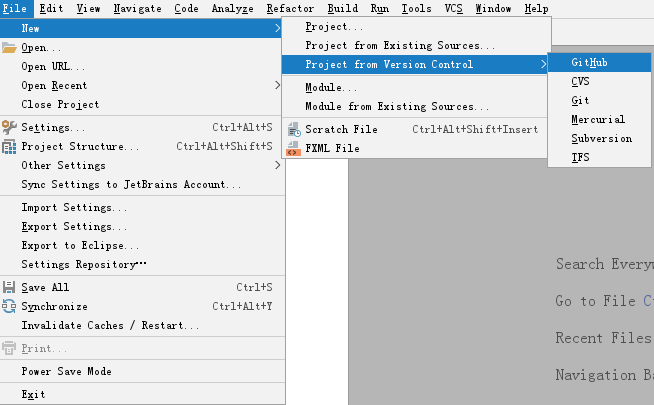


## 4-3 GitHub远程库clone项目到IDEA

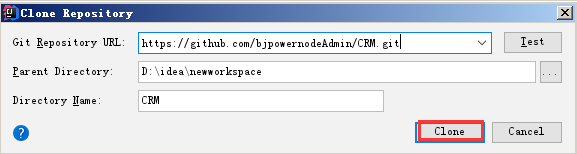
新建一个空的Project，在另一个窗口打开（同时保留原有的项目窗口）



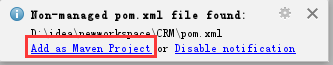
开始clone项目



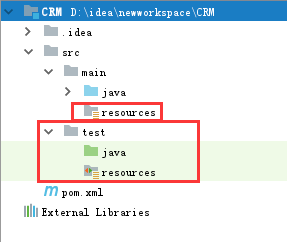
填写基本信息



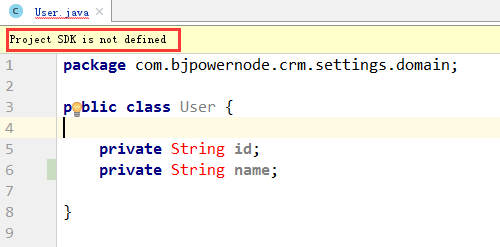
作为一个maven项目导入



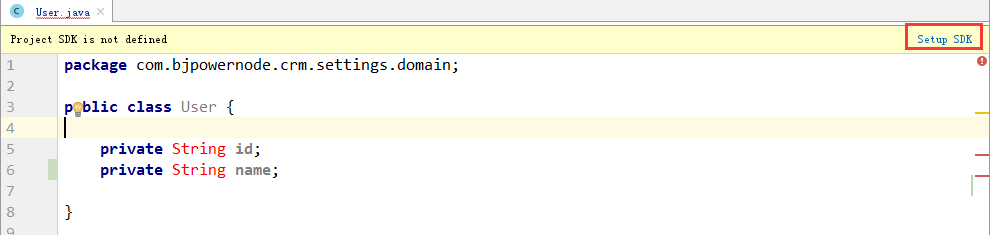
将Maven项目目录结构写完整（注意文件夹颜色要赋予正确的颜色）

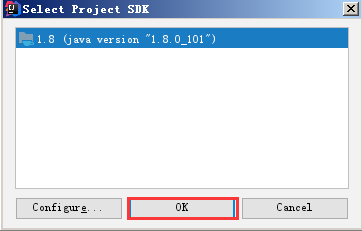


当编写java代码时，会提示没有编译环境



点击右侧的setup sdk





将加入了新属性的User类推送到远程库

注意：推送前，不要忘记先Add！！！

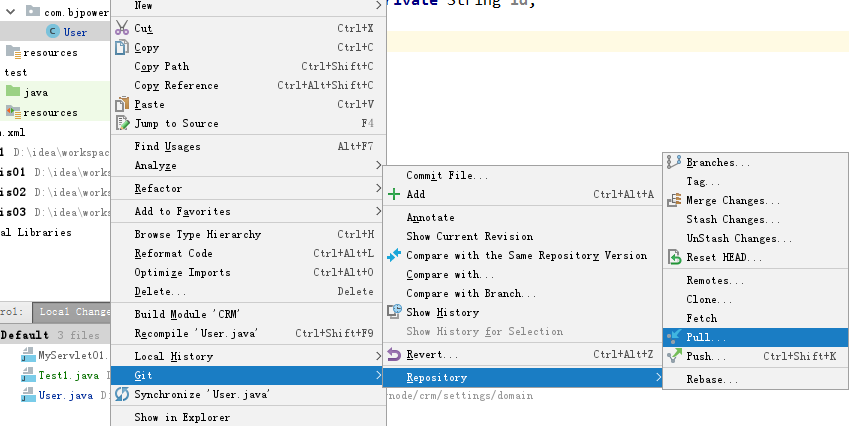
Add之后Commit and push

切换到第一个项目

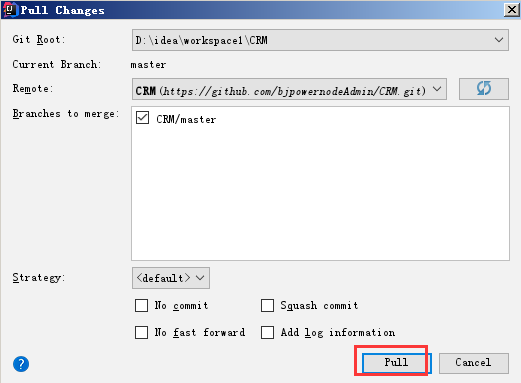
暂时只有id属性（第二个项目有id属性和name属性）

所以第一个项目需要将name属性拉取，将更新为最新版

右键User类

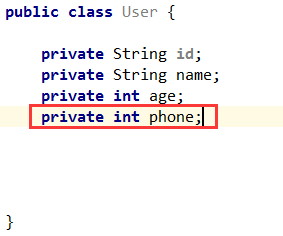


如下图所示，所有默认，点击Pull

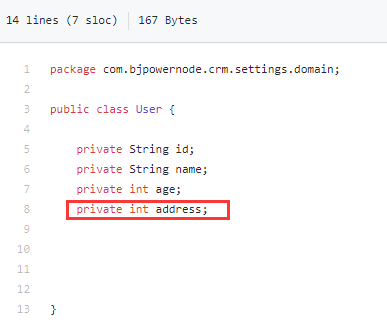


如果本地更改过文件（没有及时上传），拉取时会产生冲突

例如本地要上传的新属性是phone



但是远程库，最新的属性是address

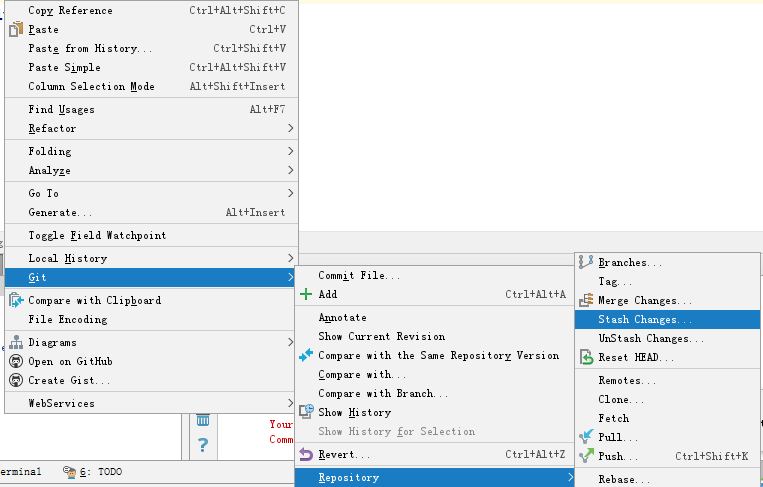


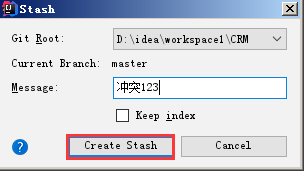
这样会产生冲突，pull会失败



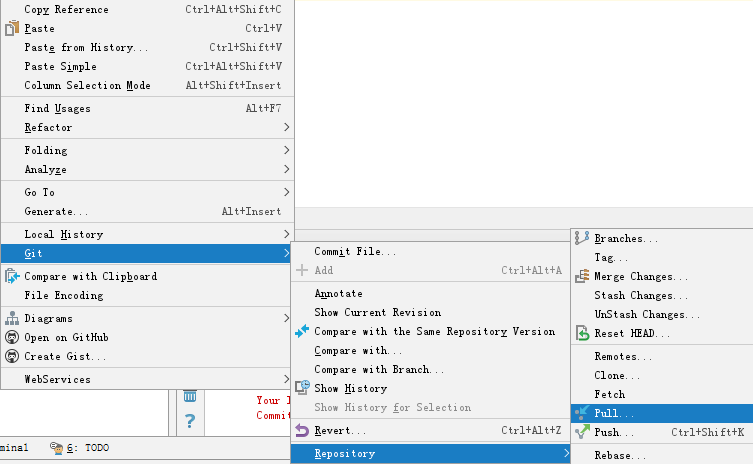
此时老版本idea，必须先要进行以下操作：

先将自己本地库最新的版本先以Stash的方式保存

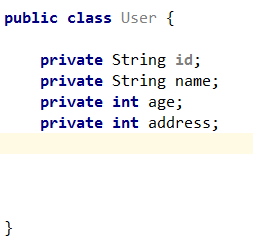




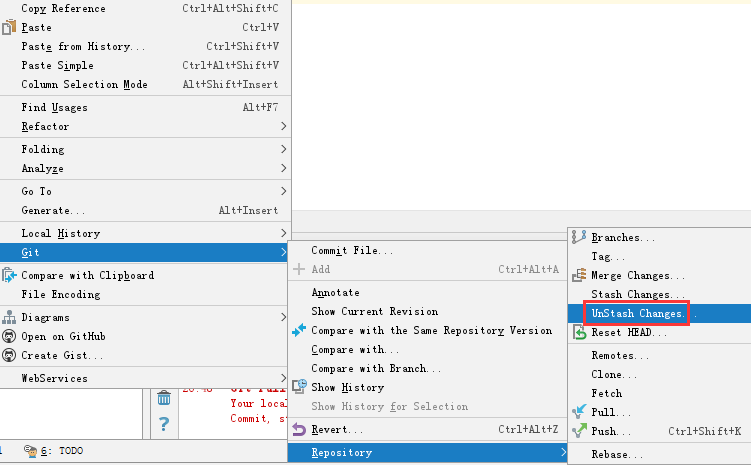
然后再拉取

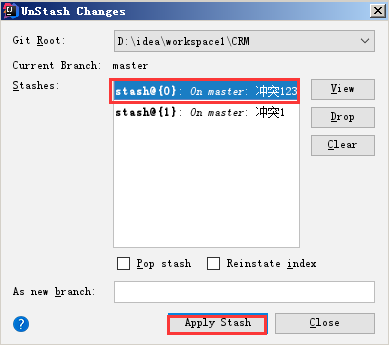


就会拉取成功了

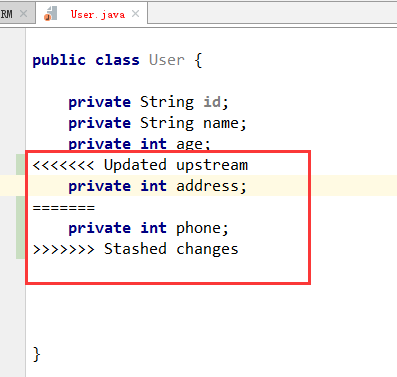


最后将本地最新版（属性phone）合并进去





我们会看到即保留了刚刚pull远程库的最新版本的代码（address），又保留了自己本地库（phone）的代码

（新版本idea会直接跳到这步）

调整冲突后（同时保留phone和address），执行add和commit操作，并push到远程库

**注意：解决冲突后的文件，必须先add，在commit**

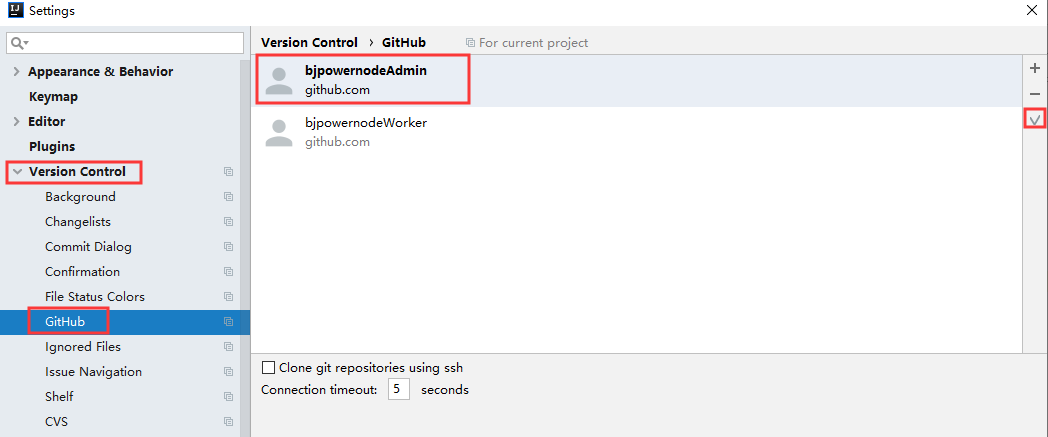
## 4-4 GitHub实战演练

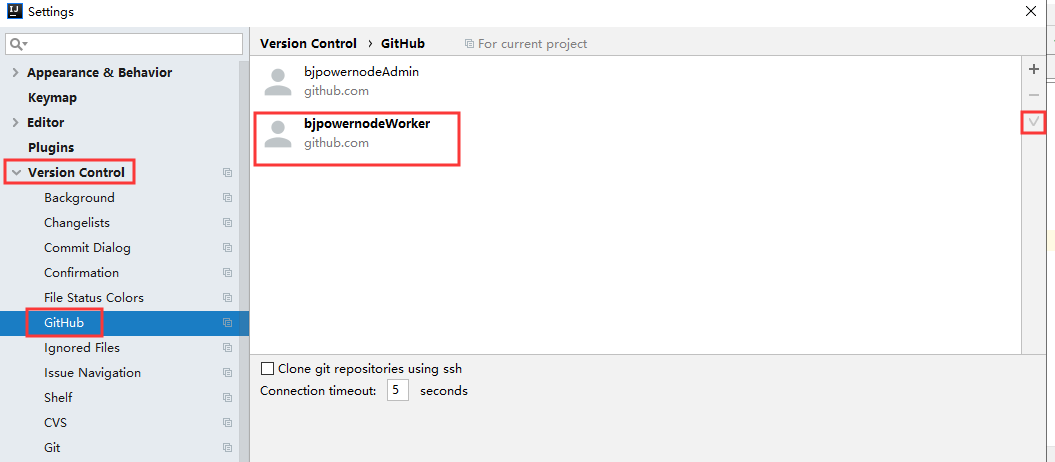
打开两个idea



设置GitHub远程库账号

一个登录bjpowernodeAdmin账号，一个登录bjpowernodeWorker账号





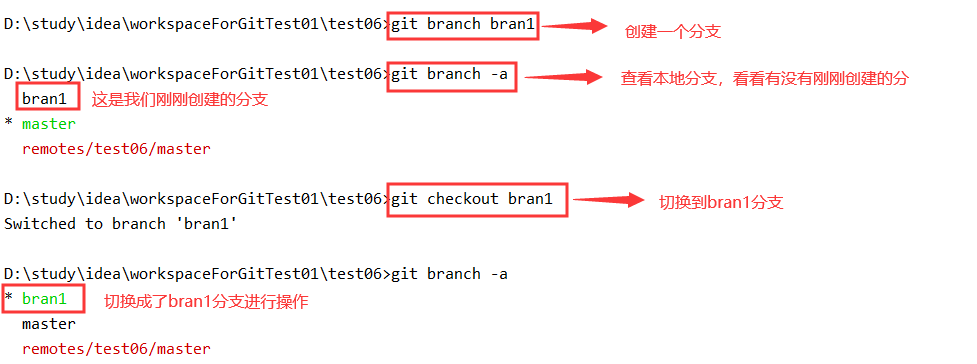
打开第一个窗口，然后打开idea中的控制台

执行命令：git branch -v 查看本地分支

执行命令：git branch -a 查看本地及远程分支

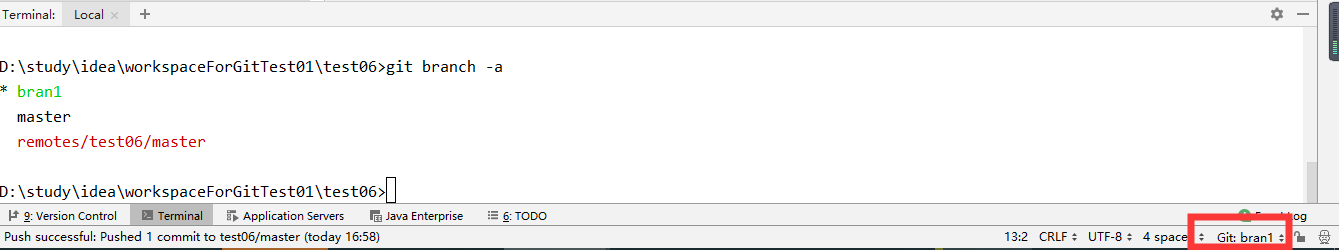


接下来按照之前的方式创建分支，并切换到该分支

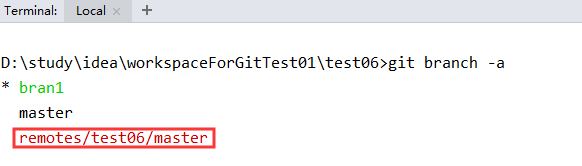


在当前项目的随便一个文件上，随便敲点东西

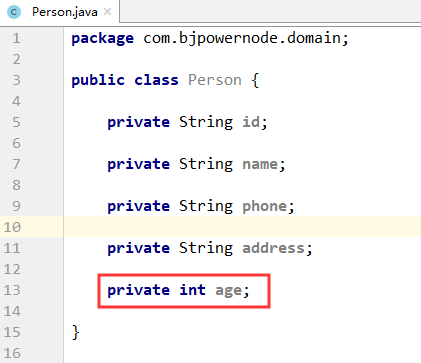
然后观察idea右下角，就会切换为刚刚切换的分支



同时从分支信息查看可以看到，我们现在虽然创建了本地的分支，但是远程分支仍然只有master

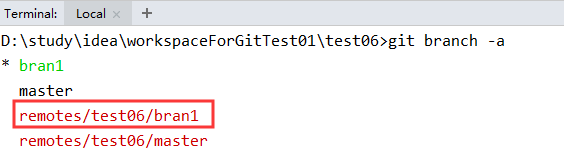


添加代码 private int age;



然后通过当前分支bran1（上一步切换的bran1），add，commit，push到远程库

查看远程分支，就多出了bran1了



刚才的步骤相当于将本地的bran1分支强推到了远程上，让远程也拥有了bran1分支，但是本地的bran1分支和远程的bran1分支还没有正式的关联上。

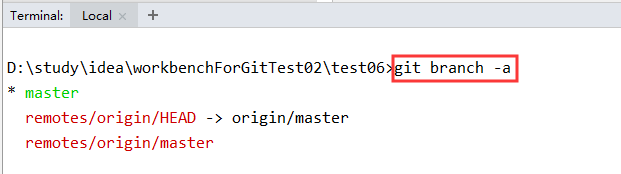
接下来，我们要让本地的bran1分支和远程的bran1分支做关联，将来本地的bran1分支做的push推送都会直接推送到远程的bran1分支上

执行命令：**git branch --set-upstream-to=test06/bran1**



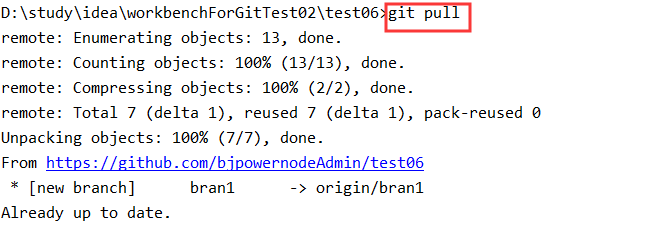
现在我们打开第二个idea，查看分支

我们会发现，第一个idea创建的bran1分支，在第二个idea中看不到

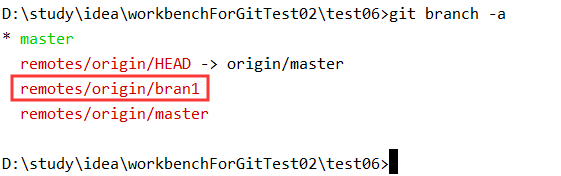


需要pull一下，这里的pull，即代表之前拉取的意义，也代表更新的意义

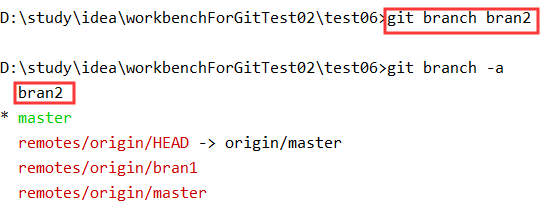
执行命令：get pull



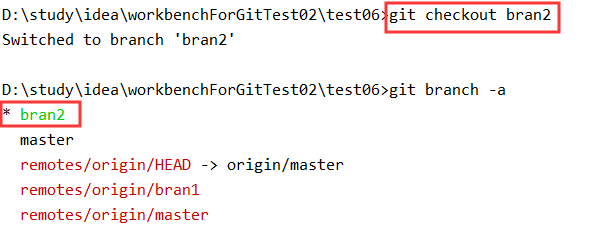
我们再查看一次分支，就可以看到刚刚第一个idea建立的远程的bran1分支了



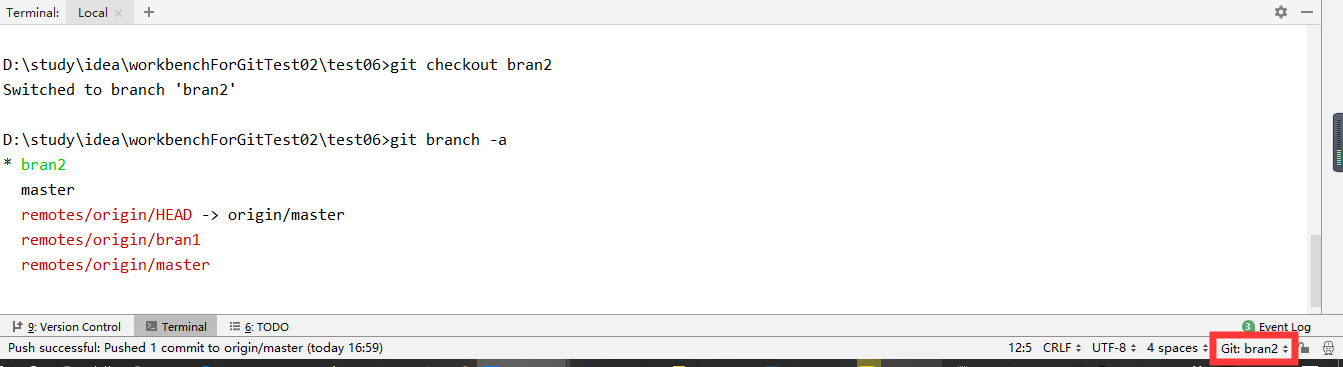
接下来在第二个idea中，新建本地分支bran2，建立完毕后查看本地分支



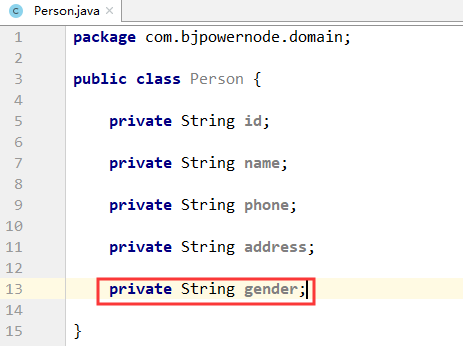
切换到bran2分支，并查看本地当前分支



与之前第一个idea的操作一样，在第二个idea的Person中随意敲点东西，查看idea右下角的当前分支



编辑Person，新增gender属性

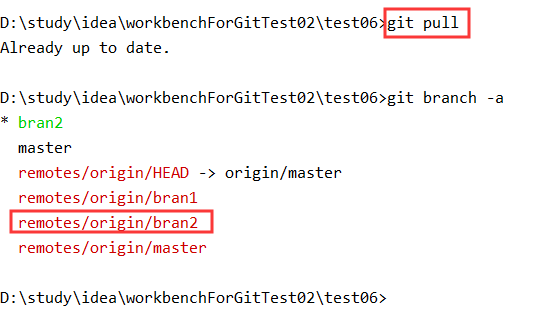


以bran2的本地分支，执行add，commit，push

如果是以前没有分支的操作，push肯定会产生冲突（因为之前bran1分支推送了age，我们并没有pull），但是这一次我们的push是成功的，因为我们使用的是不同分支的push

同时，也将本地的bran2分支强推到了GitHub上。

执行git pull，然后查看远程分支，出现了bran2

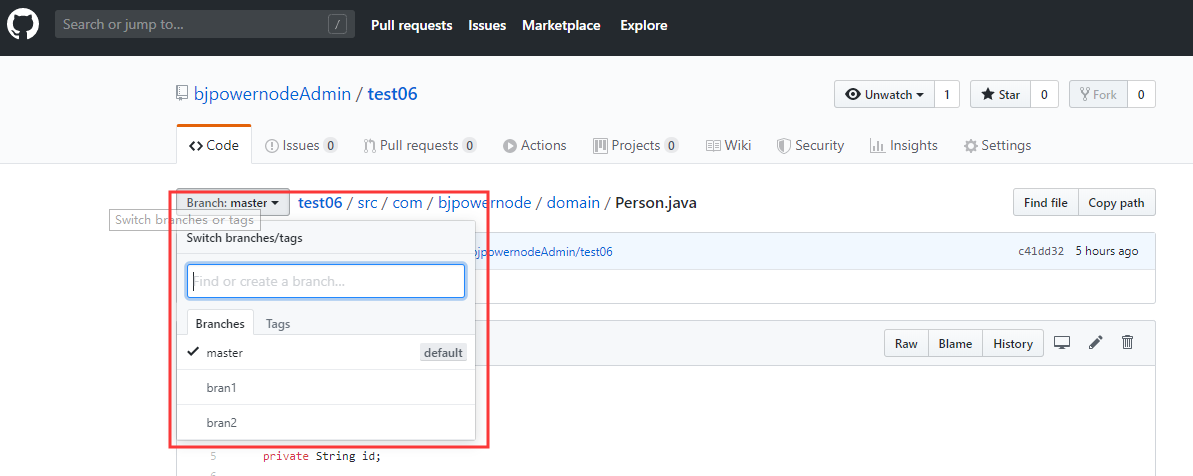


执行命令：**git branch --set-upstream-to=origin/bran2**

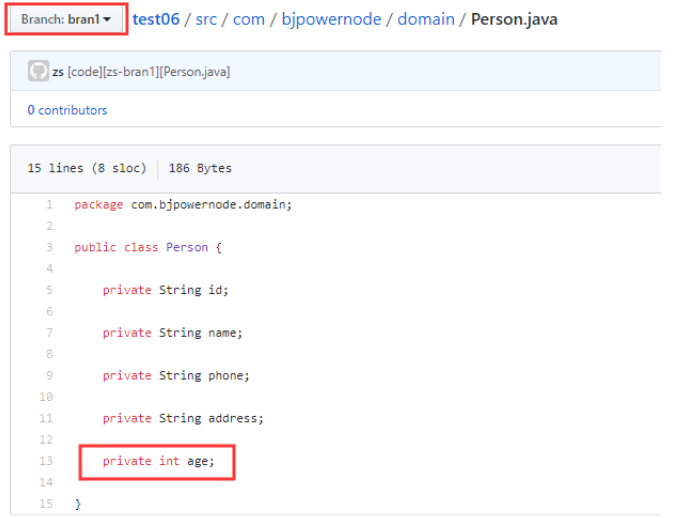
将本地bran2分支和远程bran2分支做关联



观察github远程库，除了早期的master，我们由看到了刚刚推送的bran1和bran2分支



切换到bran1，我们会看到age



切换到bran2，我们会看到gender



观察master，仍然只有之前的id，name，phone，address

因为我们还没有在远程合并分支



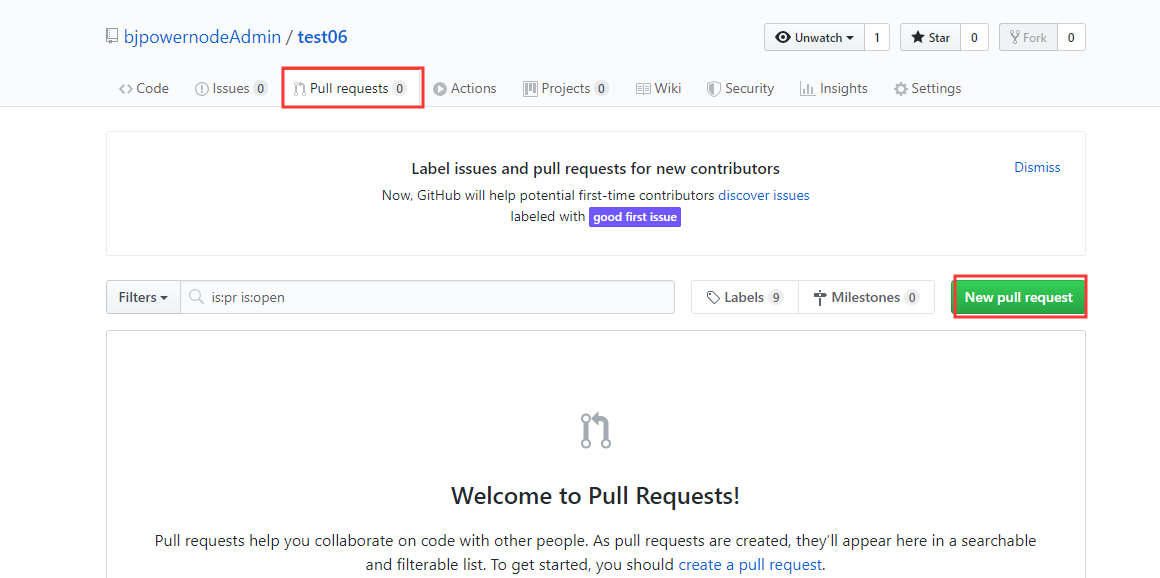
由于Git是在自己计算机中使用的版本控制工具，所以之前学习过的本地的分支冲突以及合并只适合在自己计算机中的开发。

但是现在，我们是多台计算机合作开发，所以不同的分支是需要在远程库中进行合并，我的操作方式仍然是将所有的分支合并到master上

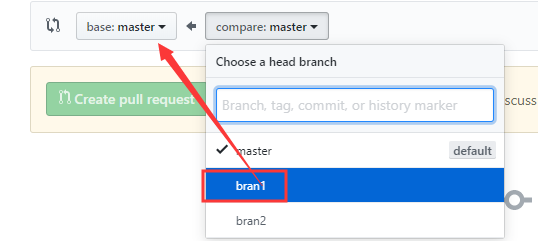
我们现在就要将bran1的age和bran2的gender合并到master上

点击菜单pull requests

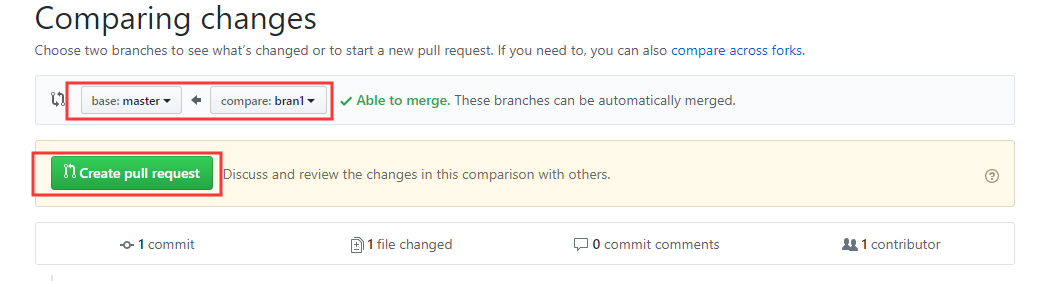
然后点击 New pull request



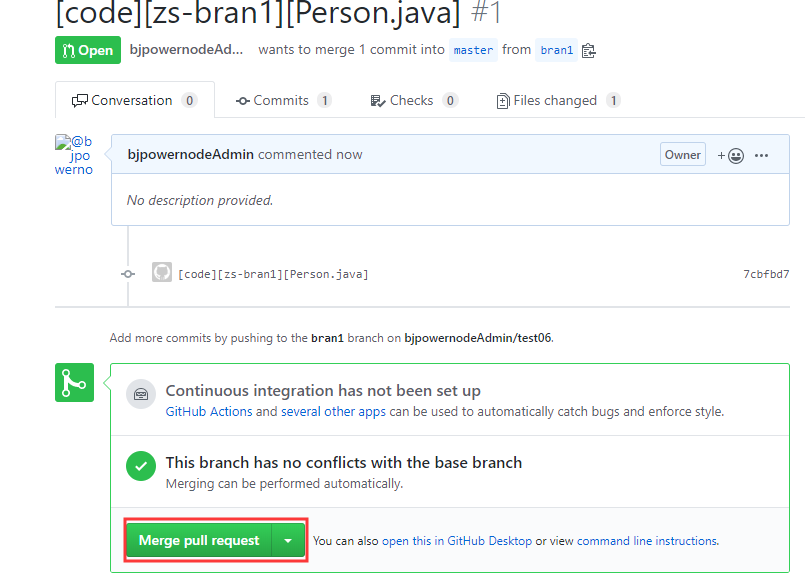
选中bran1，向master合并



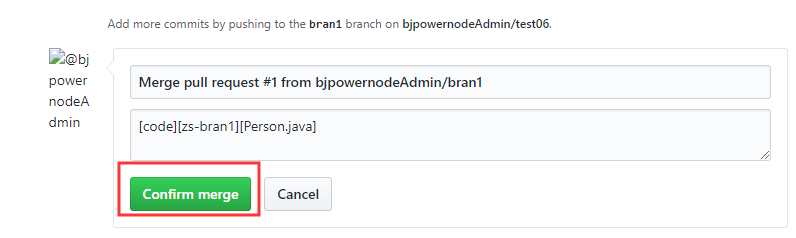
然后点击Create pull request



点击按钮，开始进行合并



继续点击按钮



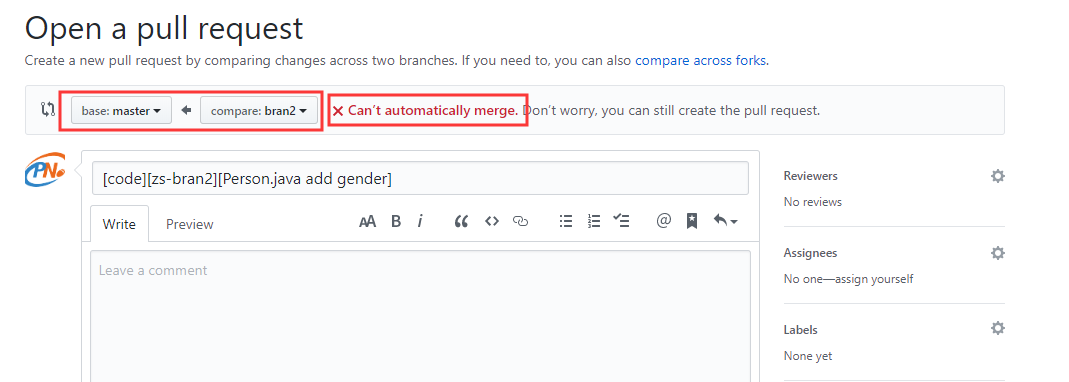
再次查看master

我们会看到bran1的age已经合并进去了



接下来我们可以通过相同的方式将bran2的gender合并进去

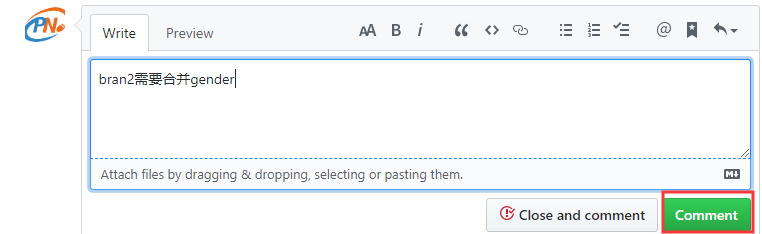
但是这时系统会告诉你不能进行合并



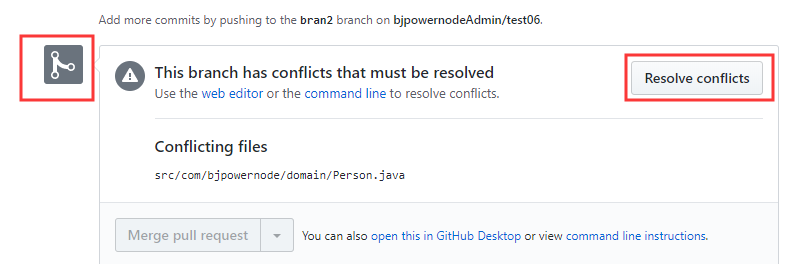
我们仍然点击下面的Create pull request按钮

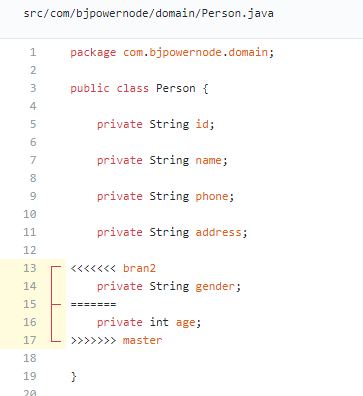


填写信息，然后点击按钮

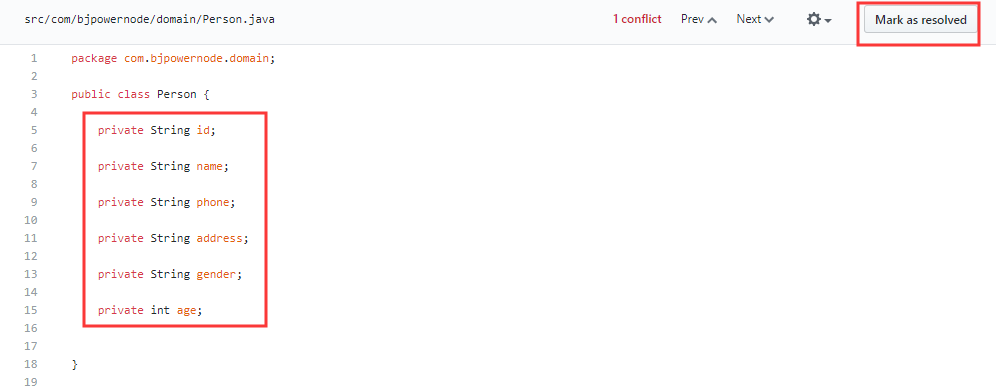


解决冲突





解决冲突



点击commit merge



这样就可以执行刚才未完成的合并master的操作了

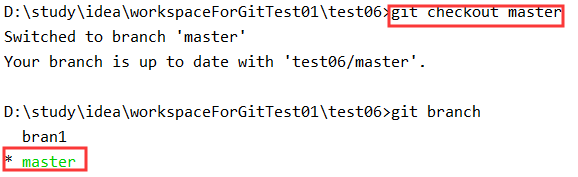


合并之后，查看master，bran1的age和bran2的gender就都有了

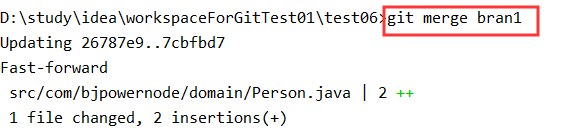


接下来我们将远程master的信息，拉取到本地master上

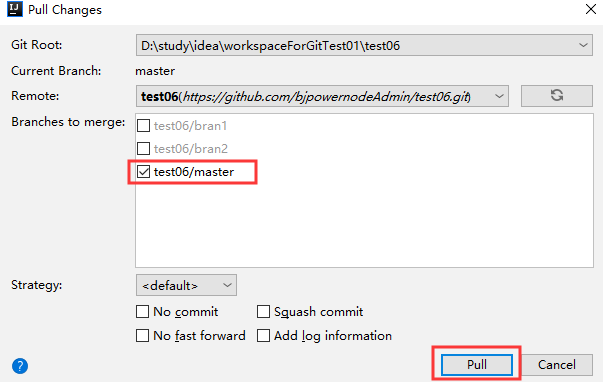
首先将本地分支先切换到master上



master先在本地与bran1合并

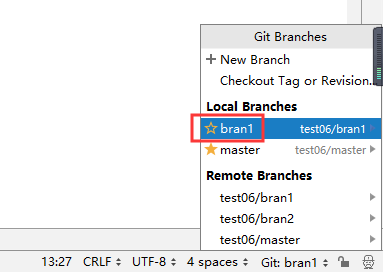


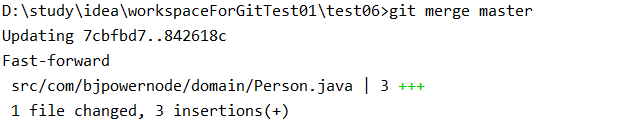
然后从远程master进行拉取操作



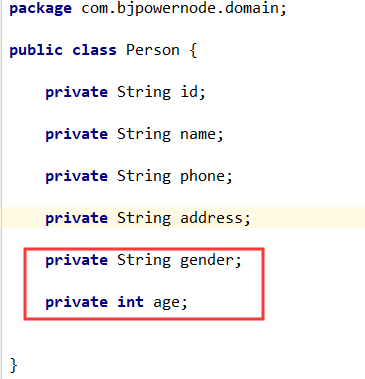
然后可以将master的信息合并到bran1分支上

如下我们也可以使用图形界面切换分支





然后bran1也将最信息的信息更新下来了



# 第五章 Git&GitHub使用注意事项

## 5-1 Git使用注意事项

### 5-1-1保持原子性的提交

每次提交的间歇尽可能地短，以几个小时的开发工作为宜。例如在更改 UI 界面的时候，可以每完成一个 UI 界面的修改或者设计，就提交一次。在开发功能模块的时候，可以每完成一个小细节功能的测试，就提交一次，在修改 bug 的时候，每修改掉一个 bug 并且确认修改了这个 bug，也就提交一次。我们提倡多提交，也就能多为代码添加上保险。

### 5-1-2对提交的信息采用明晰的标注

不论是在本机中使用本地库，还是未来推送到远程库，如果提交不明确的标注不仅仅会让自己怀疑当初提交的目的，也会让项目组中其他的成员感到很无奈，项目经理无法很清晰的掌握工作进度，无法清晰的把握此次提交的概要信息。在有必要的情况下也不能明确的的回到指定的历史记录。所以，在做提交工作时，要填写明晰的标注，能够概要的描述所提交文件的信息，让项目组其他成员在看到标注后不用详细看代码就能了解你所做的更新。

## 5-2 GitHub使用注意事项

### 5-2-1推送之前先拉取

 GitHub拉取的原则是要随时拉取，随时推送。当完成了一个小功能，能够通过编译并且自己测试之后，谨慎地推送。

 如果在修改的期间别人也更改了相同的文件，那么推送就可能会产生冲突，这种情况就需要同之前的开发人员联系，两个人一起协商解决冲突，解决冲突之后，需要两人一起测试保证解决冲突之后，程序不会影响其他功能。

### 5-2-2不要推送不能通过编译的代码

 代码在推送之前，首先要确认自己能够在本地编译。如果在代码中使用了第三方类库，要考虑到项目组成员中有些成员可能没有安装相应的第三方类库。项目经理在准备项目工作区域的时候，需要考虑到这样的情况，确保开发小组成员在签出代码之后能够在统一的环境中进行编译。

### 5-2-3不要推送自己不明白的代码

 代码在推送进入到GitHub之后，你的代码将被项目成员所分享。如果提交了你不明白的代码，你看不懂，别人也看不懂，如果在以后出现了问题将会成为项目质量的隐患。因此在引入任何第三方代码之前，确保你对这个代码有一个很清晰的了解。

### 5-2-4提前协调好项目组成员的工作计划

 项目经理应该合理分配工作计划。每个成员在准备开始进行某项功能的修改之前，如果有可能，先跟工作小组的成员谈谈自己的修改计划，让大家都能了解你的思想，了解你即将对软件作出的修改，这样能尽可能的减少在开发过程中可能出现的冲突，提高开发效率。同时你也能够在和成员的交流中发现自己之前设计的不足，完善你的设计。

# 第六章 Gitlab介绍

## 6-1 Gitlab概述

GitLab是一个远程的Git仓库，GitLab与GitHub及国内的码云类似，用户在GitLab上注册账号，上传项目到GitLab，通过GitLab对项目进行版本控制。通常企业使用GitLab在**局域网**搭建自己的Git代码管理仓库，这样做更加适合企业内部的开发管理，随着云技术的发展，越来越多的企业开始使用提供Git仓库管理**公共服务**的平台，比如：GitHub、码云等。

## 6-2 Gitlab官网

首页地址：https://about.gitlab.com/

安装地址：https://about.gitlab.com/installation/

## 6-3 Gitlab安装

1.首先需要将虚拟机内存升级为4个G

2.配置yum源

vim /etc/yum.repos.d/gitlab-ce.repo

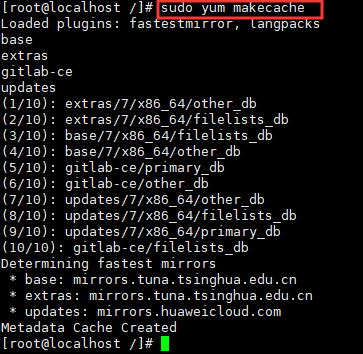


在文件中编辑以下信息，保存退出

|  |
| --- |
| [gitlab-ce]  name=gitlab-ce  baseurl=http://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el6  Repo\_gpgcheck=0  Enabled=1  Gpgkey=https://packages.gitlab.com/gpg.key |

3.更新本地yum缓存

执行命令sudo yum makecache



4.安装gitlab

自动安装最新版：sudo yum install gitlab-ce

安装所需指定版本：sudo yum install gitlab-ce-x.x.x

安装过程中如果出现如下选择，选择y表示安装继续



安装完毕后如果出现如下信息



将/etc/yum.conf中"gpgcheck=1"这一行改为"gpgcheck=0"



然后重新执行sudo yum install gitlab-ce进行安装

注意：第一次执行该命令的时候已经下载过了安装包

这一次就不用下载了，会自动进行安装

看到此画面表示安装完毕

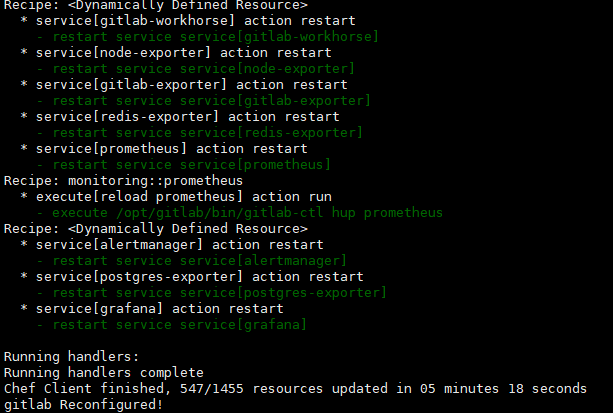


5.执行reboot命令重新启动服务器

6.接下来初始化Gitlab

执行命令gitlab-ctl reconfigure

注意：在执行此命令期间需要耐心等待



7.接下来启用gitlab组件

执行命令gitlab-ctl start



8.关闭Linux防火墙，否则不能通过外部访问服务



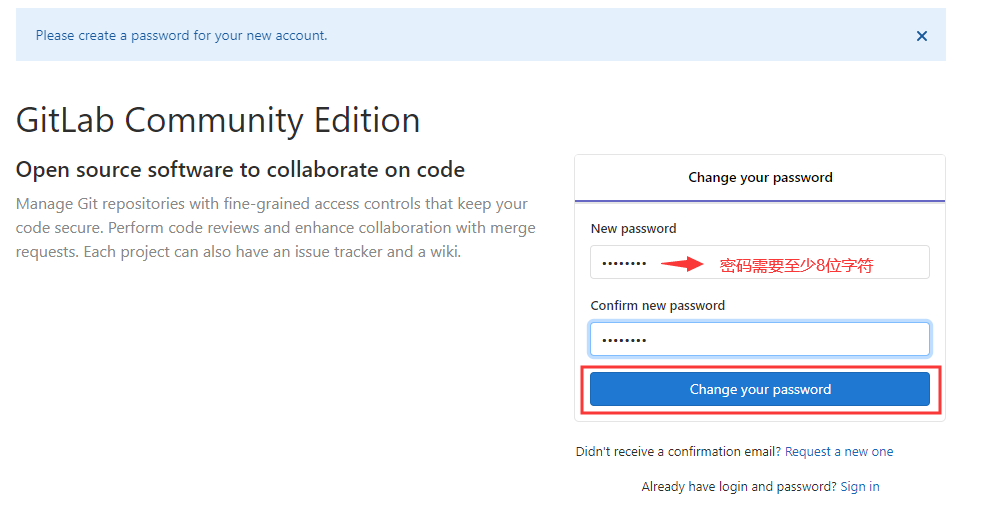
## 6-4 Gitlab访问

打开浏览器，在地址栏输入Linux的ip地址，敲回车键即可访问gitlab服务

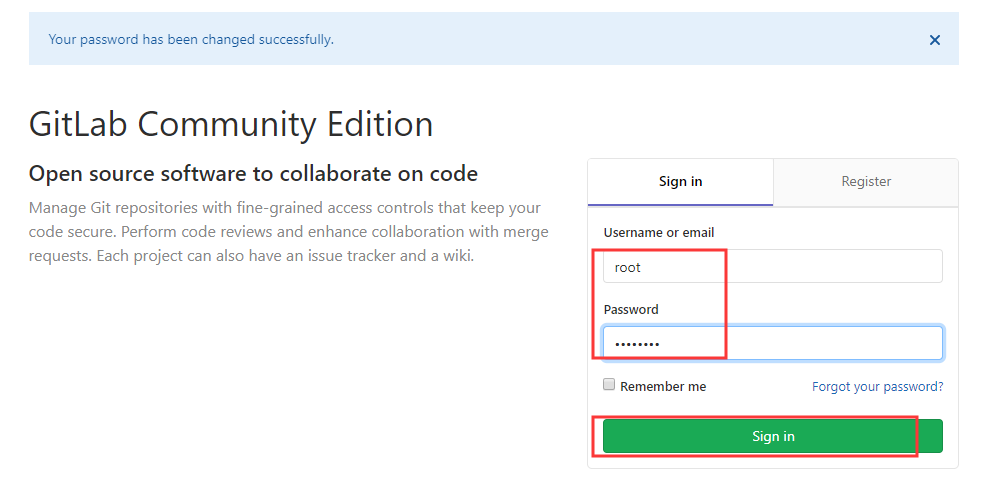
[http://192.xxx.xxx.x](http://192.168.40.128/)xx

进入到gitlab初始页

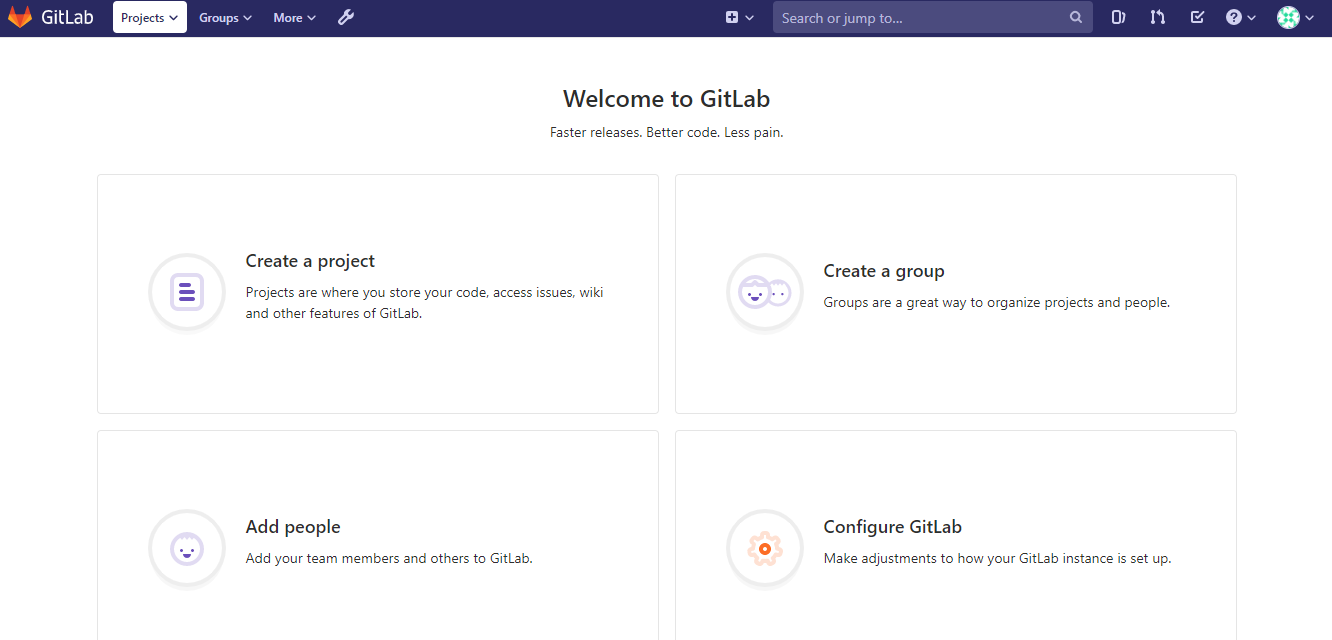
这个界面是要给gitlab的root账号设置一个密码



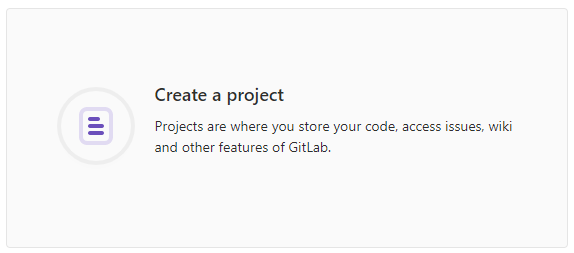
然后进入到登录页，账号为root，密码就是刚才我们设置的那个



进入到gitlab主页面

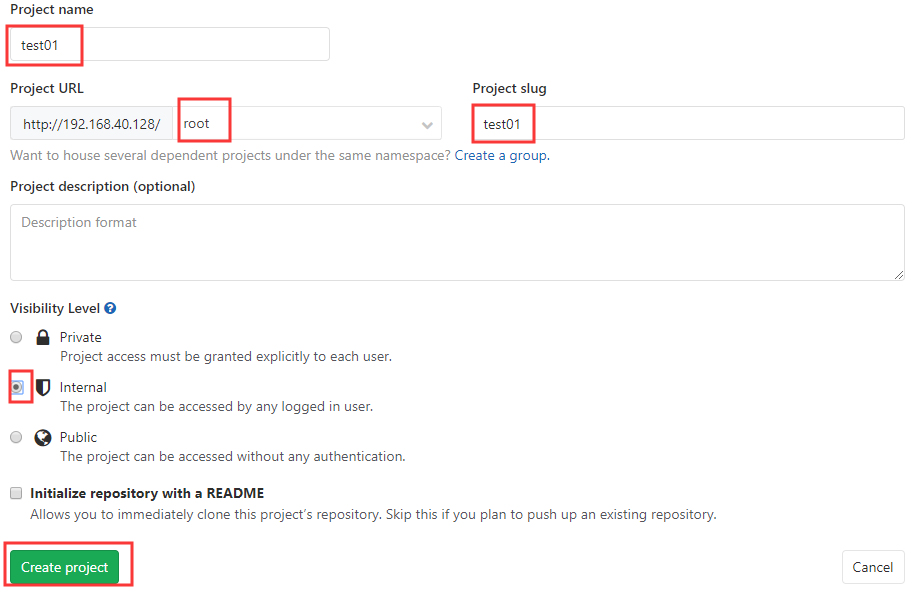


创建一个新工程（建立远程库）

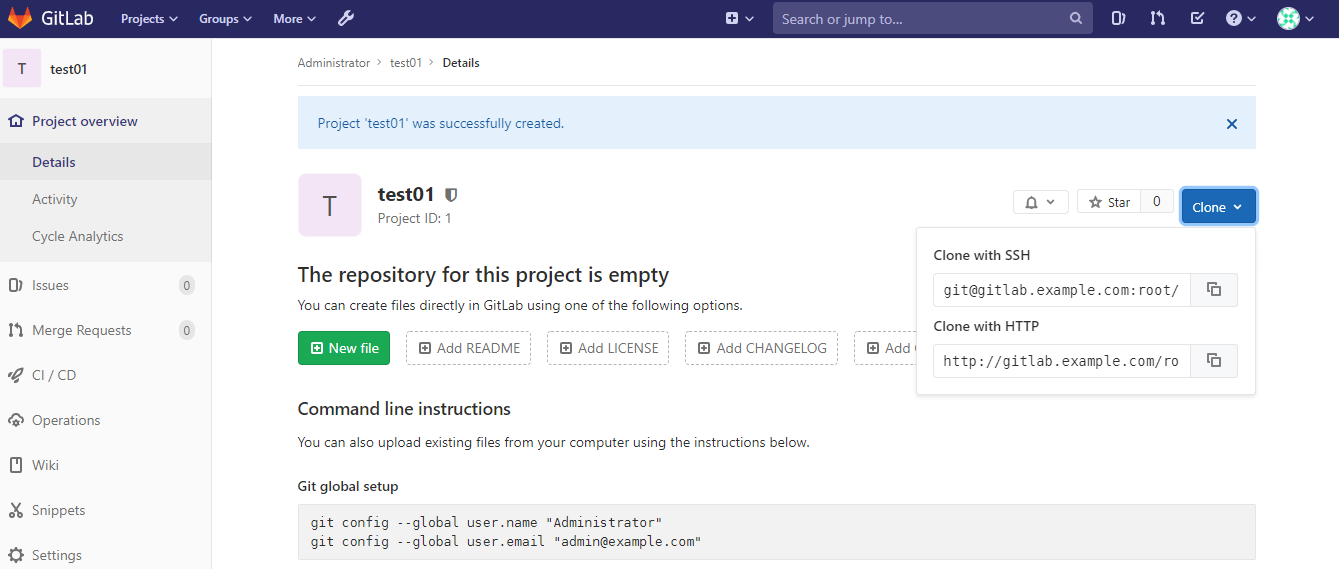


进入到如下页面

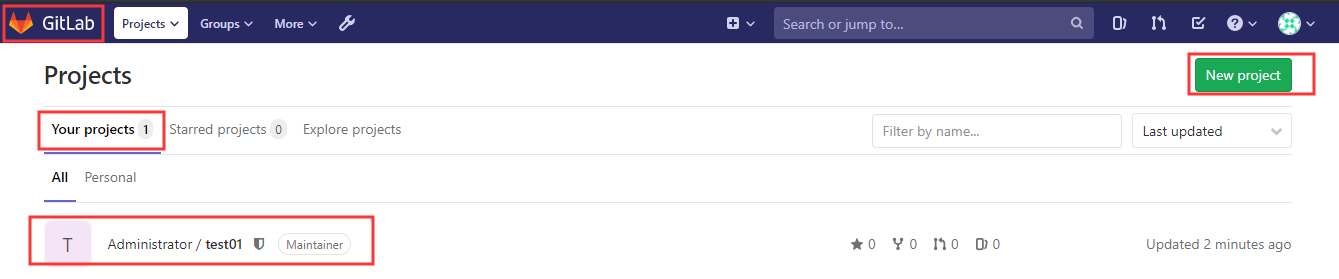
与之前的github用法一致，只不过界面样式不同



clone地址与github一模一样，都提供了SSH和HTTP两种方式



点击左上角LOGO进入到页面，通过该页面可以查看我们的远程库，同时也可以创建新的远程库



## 6-5 Gitlab补充

gitlab系统查看及设置等相关命令

|  |
| --- |
| sudo gitlab-ctl start # 启动所有 gitlab 组件；  sudo gitlab-ctl stop # 停止所有 gitlab 组件；  sudo gitlab-ctl restart # 重启所有 gitlab 组件；  sudo gitlab-ctl status # 查看服务状态；  sudo gitlab-ctl reconfigure # 启动服务；  sudo vim /etc/gitlab/gitlab.rb # 修改默认的配置文件；  sudo gitlab-ctl tail # 查看日志； |