# Git 的17条基本用法

原创 2017-11-13 谢瑛俊 博文视点Broadview

**小编说：**在开发过程中，经常会遇到一个项目由多人合力完成这种情况，每个人负责其中一个模块。项目开发过程中为了确保代码的可追溯，我们引入了版本控制概念，每个人修改了什么代码或提交了什么代码都能够跟踪记录。现在流行的版本控制主要有：集中式版本控制（SVN）和分布式版本控制（GIT）。本文将介绍Git的17条基本用法。

本文选自**《Python全栈开发实践入门》**了解更多本书信息请点击**阅读原文**。

## 1．初始化Git仓库

Git仓库分为两种类型：一种是存放在服务器上面的裸仓库，里面没有保存文件，只是存放.git的内容；一种是标准仓库，会在项目根目录下创建一个.git目录。

```
$ git init <project_name>  # 创建标准仓库，在项目根目录下创建一个隐藏的.git
# 文件夹
$ git init --bare <project_name>  # 创建一个裸仓库，裸仓库只有.git目录内容，
# 而没有工作区域，一般用于在共享服务器上面创建。
```

## 2．查看当前Git配置

Git配置信息分成三个级别，分别存放在三个不同的地方。

- 一个是系统级别的配置文件，系统基本配置文件存放在Git的安装目录中。
- 一个是用户级别配置文件，用户级别配置文件存放在当前用户目录下的.gitconfig文件内。
- 一个是项目级别配置文件，项目级别的配置文件会存放在.git目录的config文件中。使用git config --list显示的Git配置信息，是从系统级配置•用户级配置•项目级配置一层层叠加显示出来的，当遇到同项不同内容时以低级的配置为准，如图1至图3所示。

```
$ git config --list  # 显示当前Git配置信息
$ git config --system --list  # 显示系统级别Git配置信息
$ cat .git/config  # 显示项目配置文件
$ cat ~/.gitconfig  # 显示用户级别配置信息
```

```
xyj@xyj-home:~/python_fullstack/.git$ cat config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
xyj@xyj-home:~/python_fullstack/.git$
```

项目级别配置，配置文件存放在.git/config中

图1

```
xyj@xyj-home:~/python_fullstack$ cat ~/.gitconfig
[user]
        name = xyj
        email = xieyingjun@vip.qq.com
xyj@xyj-home:~/python_fullstack$
```

用户级别配置文件存放在当前home对应用户目录下的.gitconfig文件中

图2

```
xyj@xyj-home:~$ git config --system --list
fatal: unable to read config file '/etc/gitconfig': No such file or directory
```

因为我们没有配置系统级配置，所以这个文件没生成

图3

## 3．配置当前用户名和邮箱

前面我们说过，用Git进行版本控制与集中式版本控制不同，集中版本控制需要验证用户信息后才能提交代码，这样可以识别出谁提交了代码；而分布式版本控制的所有文件都保存在本地磁盘中，当提交代码的时候，需要配置一个用户信息才能被Git执行，在团体合作开发的时候用于识别文件是谁提交的，但这个识别并没有验证用户的真伪，如图4所示。

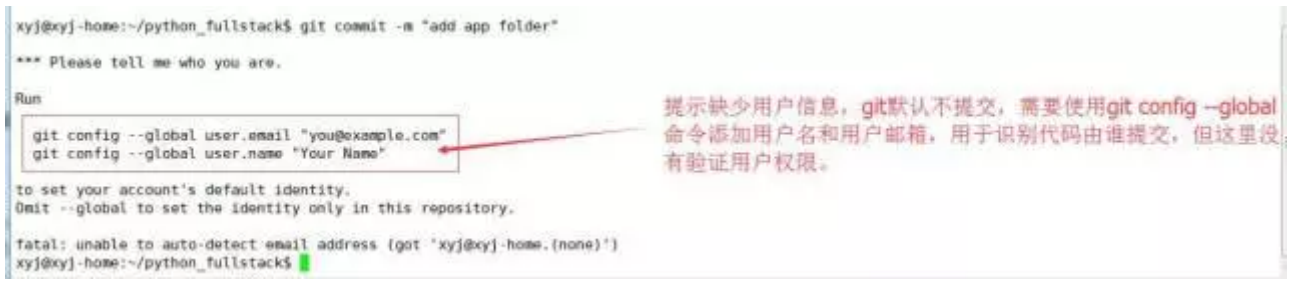


图4

$ git config --global user.name "用户名" # 在用户级配置上设置用户名

$ git config --global user.email "用户邮箱" # 在用户级配置上设置邮箱

如图5所示。

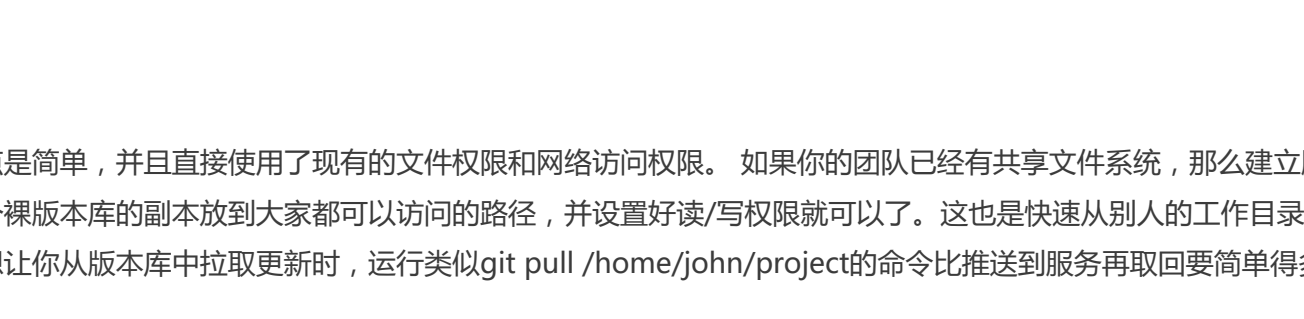


图5

注意：在用户级别配置上设置用户名和邮箱信息，应避免如下情形，假设开发用的电脑为多人使用，并且有一个用户忘记给项目设置用户信息，这时Git会把用户信息默认设置为系统级别的信息，而不给出任何提示。

## 4．克隆仓库

克隆仓库是从远程服务器上拉取一个完整的仓库到本地磁盘，这样做的好处在于每个人都有一个完整的代码库，避免把鸡蛋放在同一个篮子里。但相对的，每个人都有一个完整仓库，对代码的防泄露又是一个问题。

$ git clone <url> # 从一个远程Git仓库中克隆到本地磁盘

注意：Git支持URL传输协议：本地协议（Local）、HTTP 协议、SSH（Secure Shell）协议、FTP协议、Git 协议。

### （1）本地协议。

本地协议（Local protocol），使用的是File Protocol（本地文件传输协议），主要用于访问本地计算机中的文件，使用file://<文件路径>访问。所以远程版本库就是硬盘内的另一个目录。

**优点：**

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。 如果你的团队已经有共享文件系统，那么建立版本库会十分容易。只需像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写权限就可以了。这也是快速从别人的工作目录中拉取更新的方法。如果你和别人一起合作一个项目，他想让你从版本库中拉取更新时，运行类似git pull /home/john/project的命令比推送到服务再取回要简单得多。

**缺点：**

这种方法的缺点是，通常共享文件系统比较难配置，并且不方便从多个位置访问。如果你想从家里推送内容，则必须先挂载一个远程磁盘，与网络连接的访问方式相比，配置不方便，速度也慢。值得一提的是，如果你使用的是类似于共享挂载的文件系统，那么这个方法也不一定是最快的。访问本地版本库的速度与访问数据的速度是一样的。在同一个服务器上，如果允许Git访问本地硬盘，则一般来说，通过NFS访问版本库的速度要慢于通过SSH访问。

这个协议并不能使仓库避免意外的损坏。每一个用户都有“远程”目录的完整shell权限，我们无法阻止他们修改或删除Git内部文件或损坏仓库。

### （2）HTTP协议。

Git通过HTTP通信有两种模式。在Git 1.6.6版本之前只有一个方式可用，十分简单并且通常是只读模式的。Git 1.6.6版本引入了一种新的更智能的协议，让Git可以像通过SSH那样智能地协商和传输数据。之后几年，这个新的HTTP协议因为其简单、智能变得十分流行。新版本的HTTP协议一般被称为“智能”HTTP协议，旧版本的一般被称为“哑”HTTP协议。我们先了解一下“智能”HTTP协议。

① 智能（Smart）HTTP协议。

智能HTTP协议的运行方式和SSH协议及Git协议类似，只是运行在标准的HTTP/S端口上，并且可以使用各种HTTP验证机制，这意味着使用起来要比SSH协议简单得多。比如可以使用HTTP协议的用户名 / 密码的基础授权，免去设置SSH公钥。

智能HTTP协议或许已经是最流行的使用Git的方式了，它既支持像git://协议一样设置匿名服务，也可以像SSH协议一样提供传输时的授权和加密。而且只用一个URL就可以做到，不必为不同的需求设置不同的URL。如果你要推送到一个需要授权的服务器上（一般来讲都需要），那么服务器会提示你输入用户名和密码。从服务器获取数据时也是如此。

② 哑（Dumb）HTTP协议。

如果服务器没有提供智能HTTP协议的服务，则Git客户端会尝试使用更简单的哑HTTP协议。在哑HTTP协议里，Web服务器仅把裸版本库当作普通文件来对待，提供文件服务。哑HTTP协议的优美之处在于设置起来简单。基本只需把一个裸版本库放在HTTP根目录上，设置一个叫作post-update的挂钩就可以了。此时，只要能访问Web服务器上你的版本库，就可以克隆你的版本库。下面是设置从HTTP访问版本库的方法。

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update  # 将示例脚本重命名，需要去
#掉.sample脚本才能被识别运行
$ chmod a+x hooks/post-update
```

这样就可以了。Git自带的post-update挂钩会默认执行合适的命令（ git update-server-info ），来确保通过HTTP的获取和克隆操作正常工作。这条命令会在你通过SSH向版本库推送之后被执行，然后别人就可以通过类似下面的命令来克隆了：

```
$ git clone https://example.com/gitproject.git
```

这里我们使用了Apache里设置时常用的路径 /var/www/htdocs，不过你可以使用任何静态Web服务器——只需把裸版本库放到正确的目录下即可。Git的数据是以基本的静态文件形式提供的。通常会在可以提供读 / 写的智能HTTP服务和简单的只读的哑HTTP服务之间选一个。极少会将二者混合起来提供服务。

**优点：**

我们将只关注智能HTTP协议的优点。

不同的访问方式只需一个URL，且服务器只需在授权时提示输入授权信息，这两个简便性让终端用户使用Git变得非常简单。相比SSH协议，可以使用用户名 / 密码授权是一个很大的优势，这样用户就不必在使用Git之前先在本地生成SSH密钥对再把公钥上传到服务器。对非资深的使用者，或者系统上缺少SSH相关程序的使用者而言，HTTP协议的可用性是主要的优势。与SSH协议类似，HTTP协议也非常快速和高效。

你也可以在HTTPS协议上提供只读版本库的服务，这样你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的SSL证书。

另一个好处是HTTPS协议被广泛使用，一般的企业防火墙都允许这些端口的数据通过。

**缺点：**

在一些服务器上，架设HTTPS协议的服务端会比架设SSH协议的服务端棘手一些。除了这一点，用其他协议提供Git服务与智能HTTP协议相比就几乎没有优势了。

如果你在HTTP上使用需授权的推送，那么管理凭证会比使用SSH密钥认证麻烦一些。然而，你可以使用凭证存储工具，比如OSX的Keychain或者Windows的凭证管理器。

**（3）SSH协议。**

架设Git服务器时常用SSH协议作为传输协议。因为大多数环境下已经支持通过SSH访问——即使没有也比较容易架设。SSH协议是一个验证授权的网络协议，并且，因为其普遍性，架设和使用都很容易。

通过SSH协议克隆版本库，你可以指定一个ssh://的URL：

```
$ git clone ssh://user@server/project.git
```

或者使用一个简短的scp式的写法：

```
$ git clone user@server:project.git
```

也可以不指定用户，Git会使用当前登录的用户名。

**优点：**

用SSH协议的优势很多。首先，SSH架设相对简单——SSH守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。其次，通过SSH访问是安全的——所有传输数据都要经过授权和加密。最后，与HTTP/S协议、Git协议及本地协议一样，SSH协议很高效，在传输前也会尽量压缩数据。

**缺点：**

SSH协议的缺点在于你不能通过它实现匿名访问。即便只是读取数据，使用者也要有通过SSH访问你的主机的权限，这使得SSH协议不利于开源的项目。如果只在公司网络使用，那么SSH协议可能是你唯一要用到的协议。如果要同时提供匿名只读访问和SSH协议，那么除了为自己推送架设SSH服务外，还要架设一个可以让其他人访问的服务。

**（4）Git协议。**

接下来是Git协议。这是包含在Git里的一个特殊的守护进程；它监听在一个特定的端口（ 9418 ），类似于SSH服务，但是访问无须任何授权。要想让版本库支持Git协议，则需要先创建一个git-daemon-export-ok文件——它是Git协议守护进程为这个版本库提供服务的必要条件——但是除此之外，没有任何安全措施。要么谁都可

以克隆这个版本库，要么谁都不能。这意味着，通常不能通过Git协议推送。由于没有授权机制，一旦你开放推送操作，就意味着网络上知道这个项目URL的人都可以向项目推送数据。不用说，极少会有人这么做。

**优点：**

目前，Git协议是Git使用的网络传输协议里速度最快的。如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，那么架设Git守护进程来提供服务是不错的选择。它使用与SSH相同的数据传输机制，但是省去了加密和授权的开销。

**缺点：**

Git协议的缺点是缺乏授权机制。把Git协议作为访问项目版本库的唯一手段是不可取的。一般的做法是，同时提供SSH或者HTTPS协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过git://访问只有读权限。Git协议许也是最难架设的。它要求有自己的守护进程，这就要配置xinetd或者其他程序，这些工作并不简单。它还要求防火墙开放9418端口，但是企业防火墙一般不会开放这个非标准端口。而大型的企业防火墙通常会封锁这个端口。

说明：clone和checkout的区别如下。

git clone命令是将版本库完整克隆到本地新目录中，在创建好本地库后会自动检出当前活动分支或初始化分支。

git checkout命令是创建分支或切换分支，使用该命令后会自动更新HEAD文件，将其改写成当前分支。

## 5．查看文件状态

使用git status可以查看当前工作区域内文件的状态，没被跟踪内容会在Untracked files中显示，可以通过git add <file_name>添加被跟踪，如图6所示。

$ git status

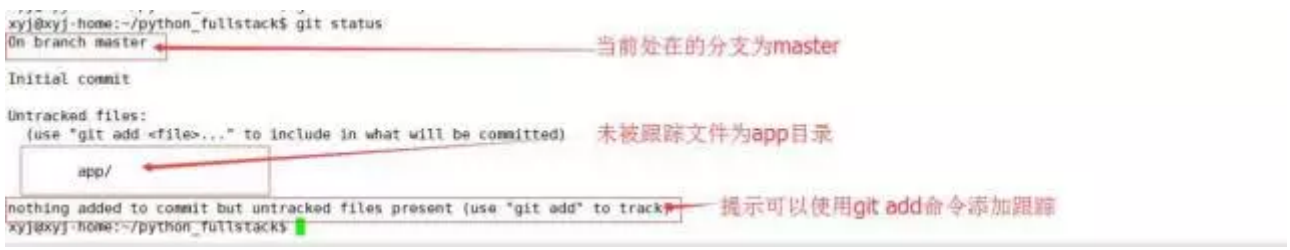


图6

## 6．添加文件追踪

使用git add <file_name>命令将文件添加到index（索引）文件中，这些文件列表将在下一次提交时记录到仓库，如图7所示。

$ git add app/ # 将app目录添加到index文件中


图7

## 7．提交代码

使用git commit命令将index文件中的更改记录提交到本地版本库。使用参数-m可以直接将要添加的备注信息写入，如图8所示。

$ git commit -m "add app folder" # 提交到版本库，并写入提交信息


图8

## 8．查看远程仓库

使用git remote命令可以显示当前版本库的远程仓库服务器信息。参数-v显示远程仓库简写名称和URL地址，如图9所示。

$ git remote -v # 显示版本库连接的远程仓库和URL

图9

## 9．添加远程仓库

使用git remote add <远程仓库别名> <URL>为本地版本库添加一个远程仓库，如图10所示。

$ git remote add origin https://github.com/lanmaokafei/python_fullstack.git # 添加一个远程仓库


图10

## 10．推送代码

使用git push <远程仓库别名> <本地分支>将本地版本库推送到远程仓库，如图11所示。

$ git push origin master # 将本地master分支提交到别名为origin的远程仓库

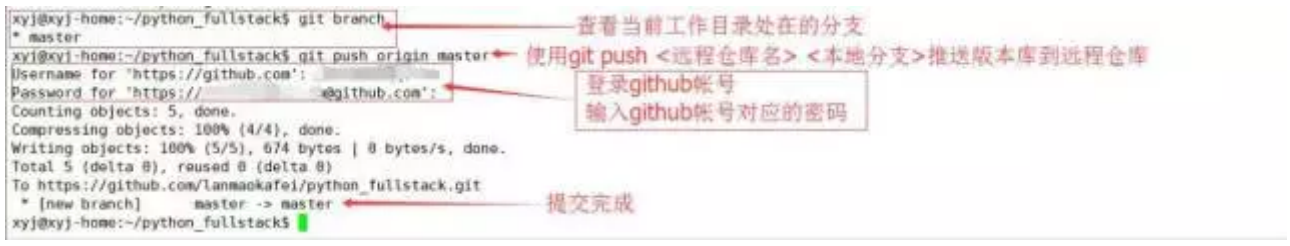


图11

## 11．从远程仓库更新代码到本地

将代码推送到远程仓库后，其他非最新版本的用户需要更新最新代码，可以使用git fetch或git pull命令来更新。区别在于fetch获取最新的代码到本地仓库，但不会自动合并分支，需要比较分支差异后合并，而pull则直接将远程仓库的版本合并到本地master上，所以git fetch相对安全些，如图12和图13所示。

$ git fetch origin master # 下载origin最新的代码


图12

$ git merge origin/master # 将origin/master分支合并到本地master中


图13

## 12．版本标记

很多书或网上会把版本标记翻译成里程碑，即给当前提交定义一个标签，标记出当前提交内容有别于其他提交。例如，在开发完一个新功能后，可以将其标记一个v1.1，代表这个功能开发完成，方便以后历史中定位这次提交。

Git有两种标签类型：一种是lightweight轻量级标签，只有版本号无其他信息；另一种是annotated附注标签，标签附带一条信息，可以让别人快速识别标签作用，建议使用这种标签。

使用git tag -n[数字] 显示当前分支下的标签信息，参数-n代表显示备注信息行数，如图14所示。

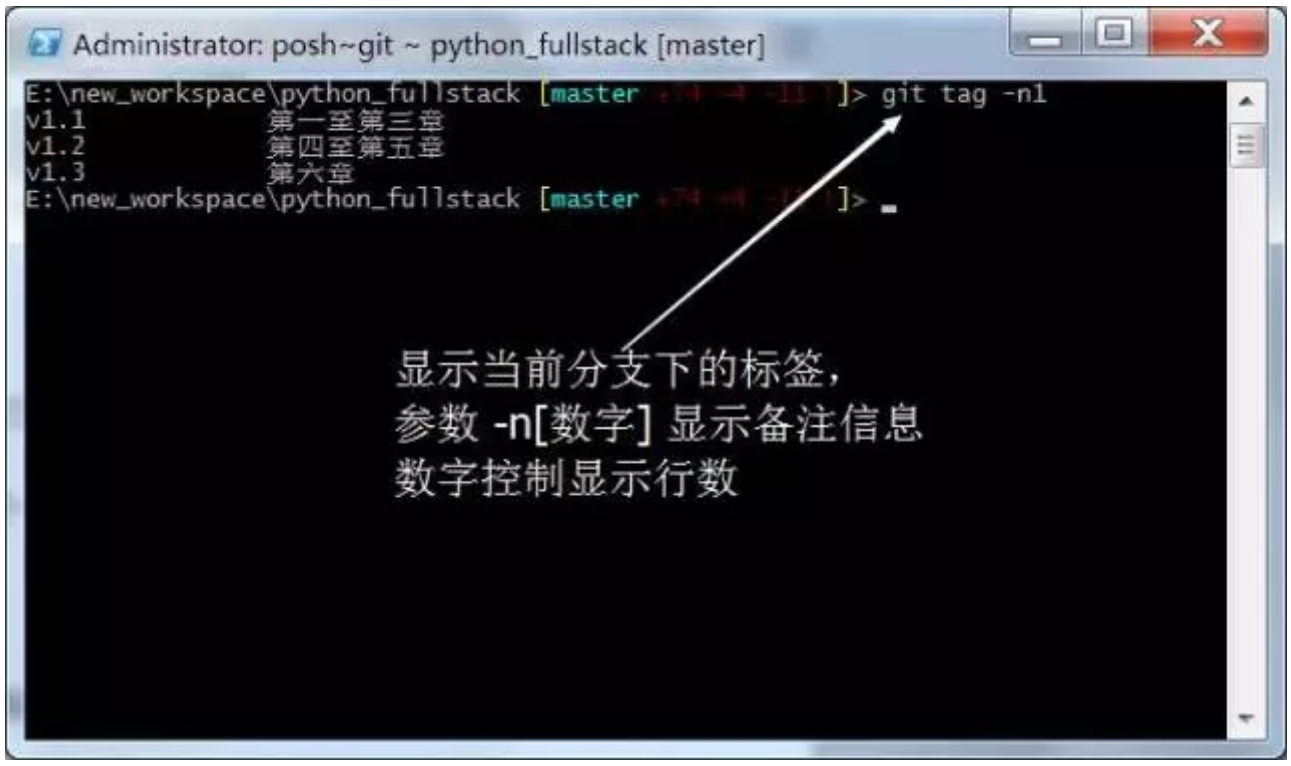


图14

使用git tag -a <版本号> -m “备注”为最新提交打上标签，如图15所示。

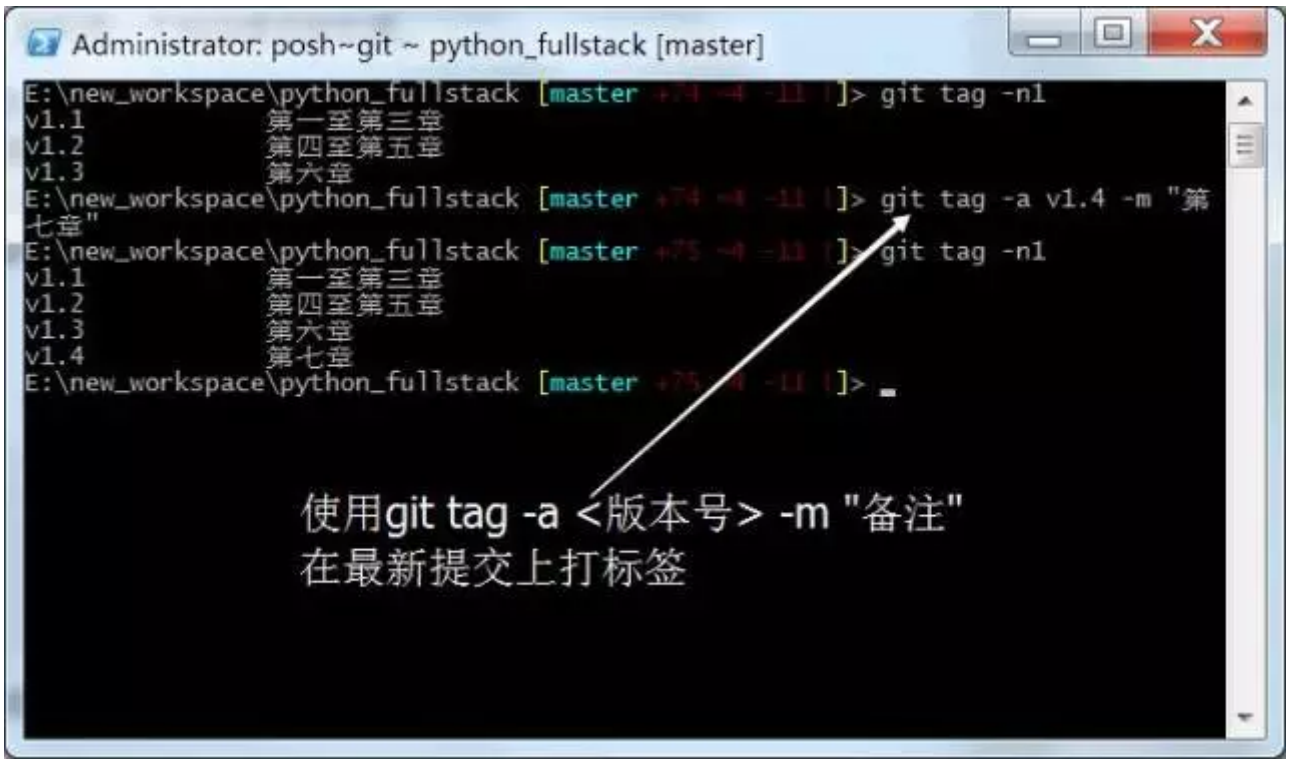


图15

使用git show <版本号>显示对应标签的版本信息和提交差异，如图16所示。

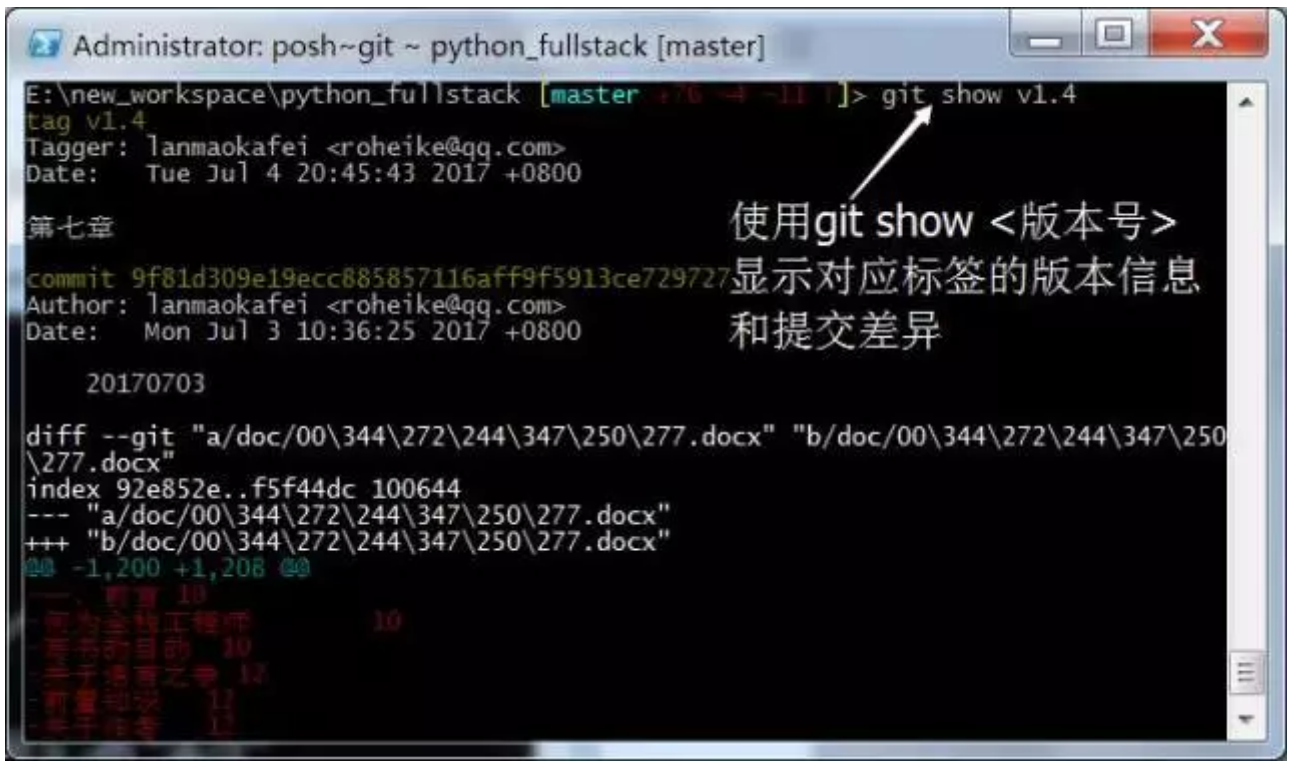


图16

## 13．添加忽略文件

在当前项目根目录下创建一个.gitignore文件，用于保存忽略列表，如图17所示。

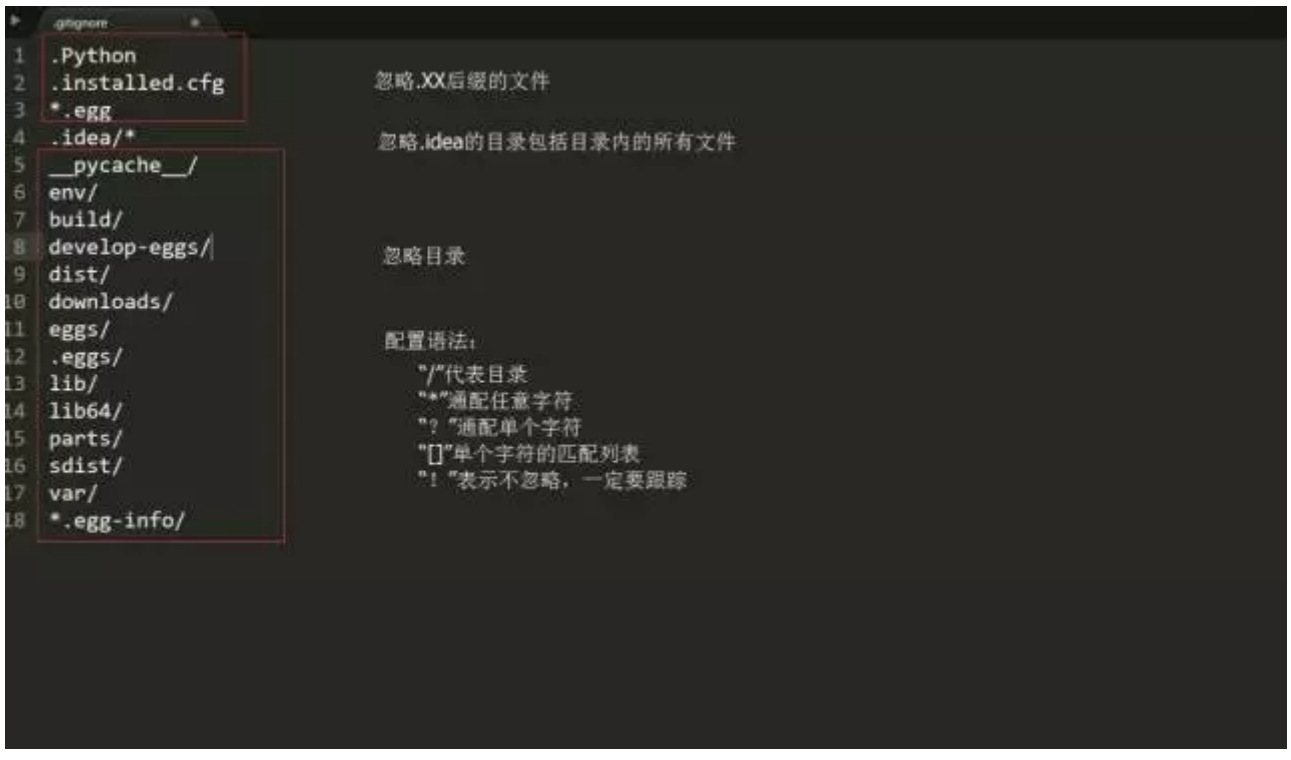


图17

配置语法如下：

> “/”代表目录；
>
> “*”代表通配任意字符；
>
> “？”代表通配单个字符；
>
> “[]”代表单个字符的匹配列表；
>
> “！”表示不忽略，一定要跟踪。

## 14．查看当前处在的分支

使用命令 git branch可以查看当前工作目录所在的分支，如图18所示。

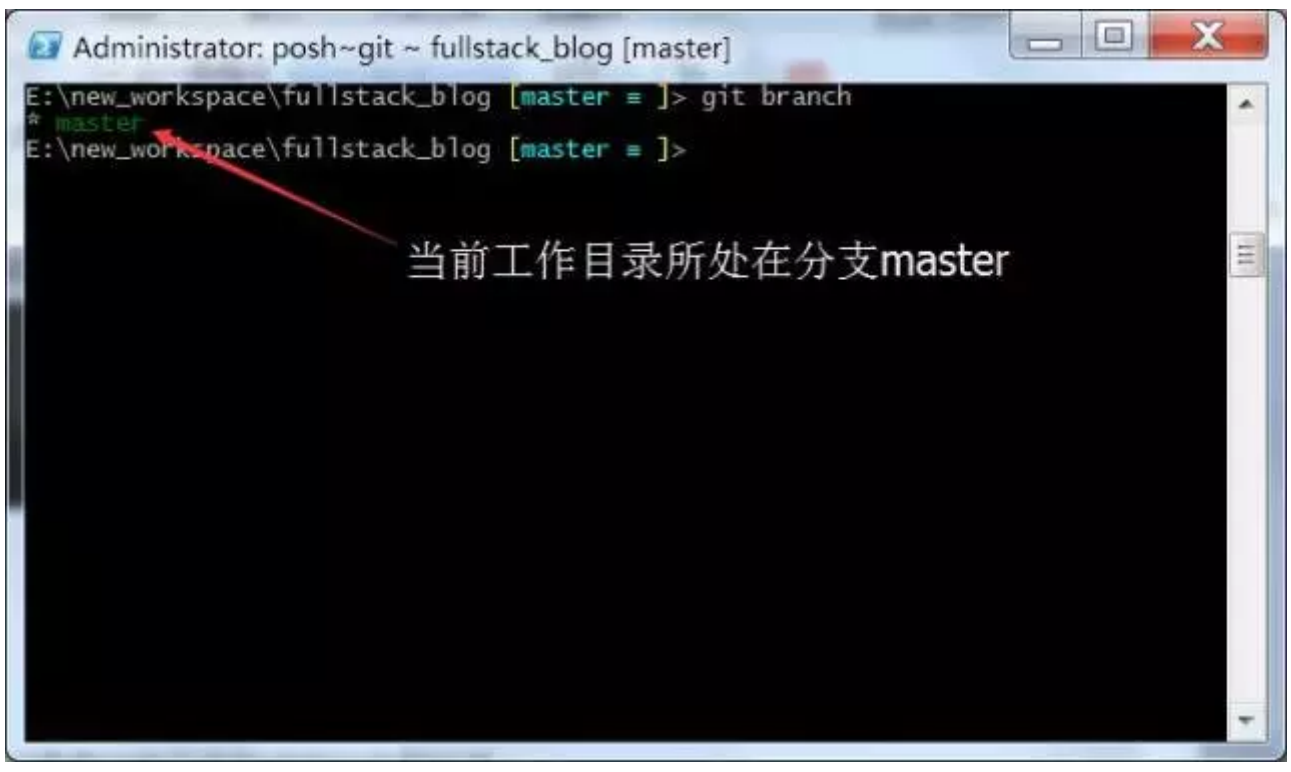


图18

## 15．创建分支

在家里开发使用的是PostgreSQL数据库，到公司后没有在线的数据就切换到SQLite，这样我们就创建一个新的分支以便在公司开发。使用命令git checkout -b jsb 创建并切换到jsb分支，命令git checkout -b等同于同时执行了命令git branch jsb 创建分支和命令git checkout jsb切换到jsb分支，如图19所示。

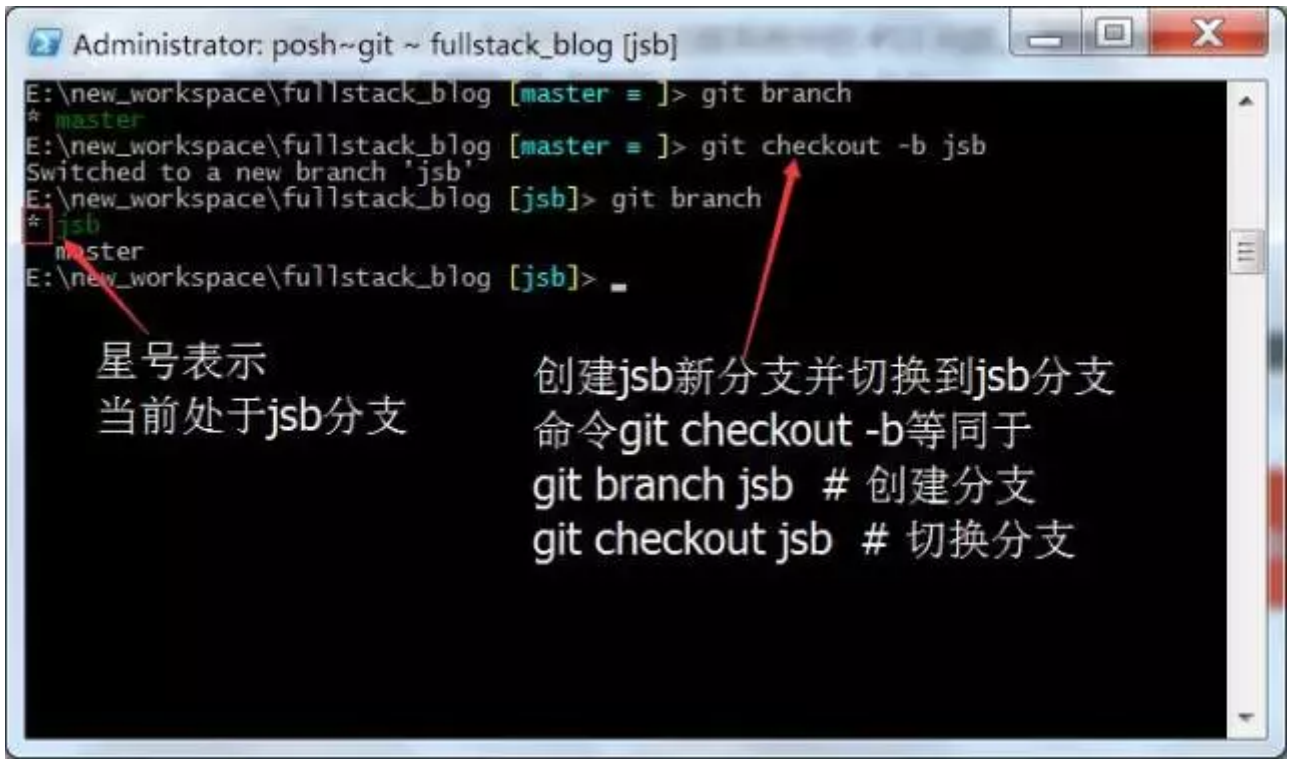


图19

## 16．合并分支

当回到家时再把在公司开发的代码和家里的版本库合并分支，切换回master分支，使用命令git merge<branch_name>合并两个分支，如图20所示。


图20

## 17．解决冲突

之前使用了不同的忽略语句，两个分支间没有冲突，但是如果两个分支同时修改了同一个文件相同位置的不同参数时，在合并的时候就会产生冲突，如图21所示。

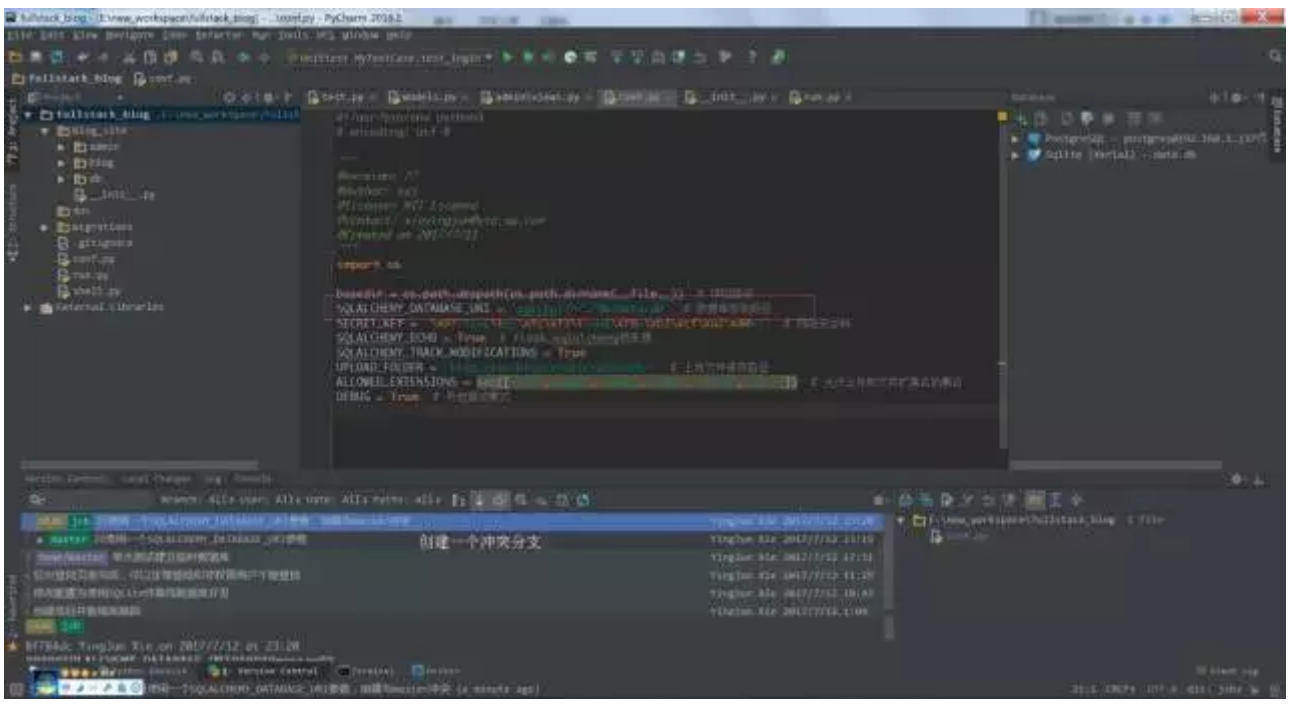


图21

在master版本中，SQLALCHEMY_DATABASE_URI 是连接PostgreSQL的数据连接'postgresql://xyj:lanmaokafei@192.168.1.137/postgres'。

在jsb版本中，SQLALCHEMY_DATABASE_URI是指向SQLite的数据文件 'sqlite:///./db/ data.db' 。

当这两个分支合并的时候就会产生冲突，需要人工修改才能合并，如图22和图23所示。

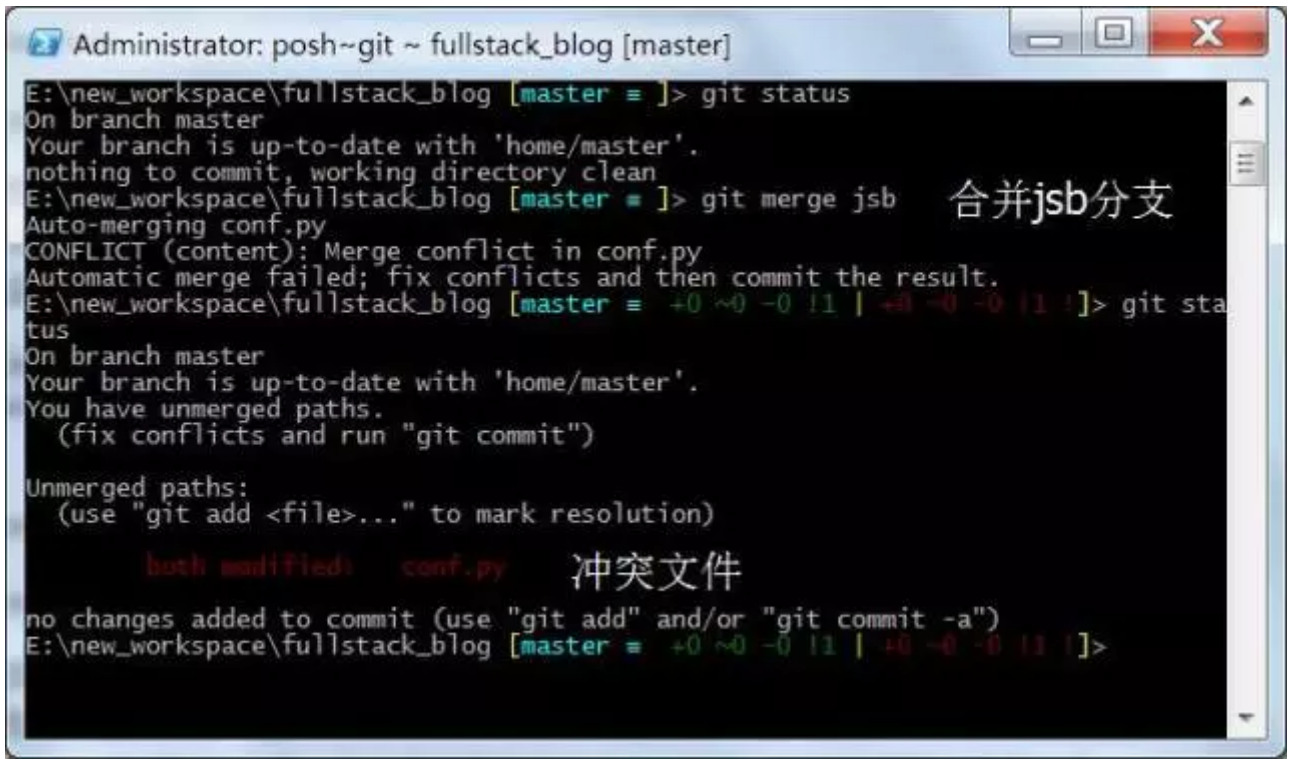


图22

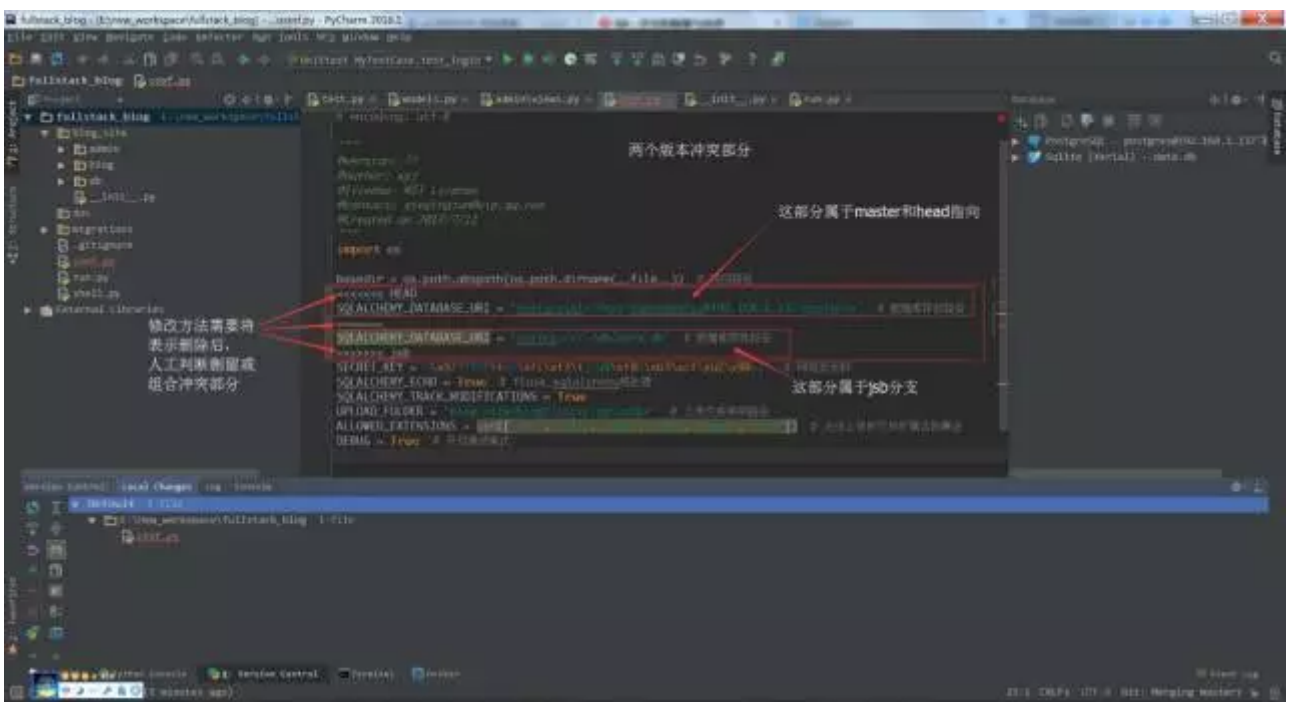


图23

说明：人工修改冲突部分，需要将自动生成的<<<<<<< HEAD、=======、>>>>>>> jsb全部删除，自行判断冲突部分需要保留什么内容或者将两者融合，修改完成后保存文件，重新使用命令git add添加文件，后再提交一次，即可解决冲突问题。

阅读原文