# Git学习记录

日期：2019年9.27

目录

**[Git学习记录](#_Toc27581_WPSOffice_Level1)** **[4](#_Toc27581_WPSOffice_Level1)**

**[1.](#_Toc10297_WPSOffice_Level1)** **[4](#_Toc10297_WPSOffice_Level1)**

**[2.Git 和代码托管中心](#_Toc6800_WPSOffice_Level1)** **[4](#_Toc6800_WPSOffice_Level1)**

[2.1代码托管中心任务：](#_Toc10297_WPSOffice_Level2) [4](#_Toc10297_WPSOffice_Level2)

[2.2局域网环境下：](#_Toc6800_WPSOffice_Level2) [4](#_Toc6800_WPSOffice_Level2)

[2.3外网环境下：](#_Toc30112_WPSOffice_Level2) [4](#_Toc30112_WPSOffice_Level2)

**[3.本地库和远程库](#_Toc30112_WPSOffice_Level1)** **[4](#_Toc30112_WPSOffice_Level1)**

[3.1团队内协作](#_Toc14435_WPSOffice_Level2) [4](#_Toc14435_WPSOffice_Level2)

[3.2团队外协作](#_Toc30134_WPSOffice_Level2) [5](#_Toc30134_WPSOffice_Level2)

**[4.Git 命令行操作](#_Toc14435_WPSOffice_Level1)** **[5](#_Toc14435_WPSOffice_Level1)**

[4.1本地库初始化](#_Toc26777_WPSOffice_Level2) [5](#_Toc26777_WPSOffice_Level2)

[4.2设置签名](#_Toc7237_WPSOffice_Level2) [6](#_Toc7237_WPSOffice_Level2)

[4.3基本操作](#_Toc30069_WPSOffice_Level2) [7](#_Toc30069_WPSOffice_Level2)

[4.4分支管理](#_Toc445_WPSOffice_Level2) [13](#_Toc445_WPSOffice_Level2)

**[5基本原理](#_Toc30134_WPSOffice_Level1)** **[16](#_Toc30134_WPSOffice_Level1)**

**[5.1git保存版本的机制](#_Toc26777_WPSOffice_Level1)** **[16](#_Toc26777_WPSOffice_Level1)**

[5.1.1 集中式版本控制工具的文件管理机制（如svn）](#_Toc30731_WPSOffice_Level2) [16](#_Toc30731_WPSOffice_Level2)

[5.1.2 git的文件管理机制](#_Toc23457_WPSOffice_Level2) [16](#_Toc23457_WPSOffice_Level2)

**[5.2分支管理机制](#_Toc7237_WPSOffice_Level1)** **[16](#_Toc7237_WPSOffice_Level1)**

**[6Github](#_Toc30069_WPSOffice_Level1)** **[16](#_Toc30069_WPSOffice_Level1)**

**[6.1注册github账号](#_Toc445_WPSOffice_Level1)** **[16](#_Toc445_WPSOffice_Level1)**

**[6.2修改头像](#_Toc30731_WPSOffice_Level1)** **[16](#_Toc30731_WPSOffice_Level1)**

**[6.3创建本地库和远程库](#_Toc23457_WPSOffice_Level1)** **[17](#_Toc23457_WPSOffice_Level1)**

**[6.4创建远程库别名](#_Toc21903_WPSOffice_Level1)** **[18](#_Toc21903_WPSOffice_Level1)**

**[6.5从本地库推送到远程库](#_Toc20776_WPSOffice_Level1)** **[19](#_Toc20776_WPSOffice_Level1)**

[命令： git push -u giedemo1 [分支名]](#_Toc21903_WPSOffice_Level2) [19](#_Toc21903_WPSOffice_Level2)

**[6.6克隆操作](#_Toc23524_WPSOffice_Level1)** **[20](#_Toc23524_WPSOffice_Level1)**

**[6.6团队协作开发](#_Toc22025_WPSOffice_Level1)** **[21](#_Toc22025_WPSOffice_Level1)**

[6.6.1加入团队成员](#_Toc20776_WPSOffice_Level2) [21](#_Toc20776_WPSOffice_Level2)

[6.6.2远程库修改的拉取](#_Toc23524_WPSOffice_Level2) [22](#_Toc23524_WPSOffice_Level2)

**[6.7跨团队协作](#_Toc8130_WPSOffice_Level1)** **[22](#_Toc8130_WPSOffice_Level1)**

**[6.7ssh免密登录](#_Toc17550_WPSOffice_Level1)** **[23](#_Toc17550_WPSOffice_Level1)**

[第一步：在bash中操作进入本地库看看有没有.ssh文件，如果没有就执行下面命令](#_Toc22025_WPSOffice_Level2) [24](#_Toc22025_WPSOffice_Level2)

[第二步：进入github，点击右上角菜单进入settings然后进入](#_Toc8130_WPSOffice_Level2) [24](#_Toc8130_WPSOffice_Level2)

[第三步：点击new SSH key](#_Toc17550_WPSOffice_Level2) [25](#_Toc17550_WPSOffice_Level2)

[第四步：把复制文件的内容粘贴到key中，tittle可以自己命名](#_Toc4336_WPSOffice_Level2) [25](#_Toc4336_WPSOffice_Level2)

[第五步：点击Add SSH key完成创建](#_Toc31077_WPSOffice_Level2) [25](#_Toc31077_WPSOffice_Level2)

[第六步：本地库测试ssh的使用](#_Toc7523_WPSOffice_Level2) [26](#_Toc7523_WPSOffice_Level2)

[1. 创建于ssh地址相对应的别名git1-ssh](#_Toc1733_WPSOffice_Level2) [26](#_Toc1733_WPSOffice_Level2)

**[6.8 commit message 规范](#_Toc4336_WPSOffice_Level1)** **[28](#_Toc4336_WPSOffice_Level1)**

[6.8.1格式](#_Toc16960_WPSOffice_Level2) [28](#_Toc16960_WPSOffice_Level2)

[6.8.2 type](#_Toc474_WPSOffice_Level2) [28](#_Toc474_WPSOffice_Level2)

[6.8.3 scope](#_Toc8729_WPSOffice_Level2) [28](#_Toc8729_WPSOffice_Level2)

# Git学习记录

### 1.

工作区 ：写代码

Git add

暂存区：临时存储

Git commit

本地库’：历史版本

### 2.Git 和代码托管中心

**2.1代码托管中心任务**：

维护远程库

**2.2局域网环境下**：

Gitlab服务器（自己可以搭建）

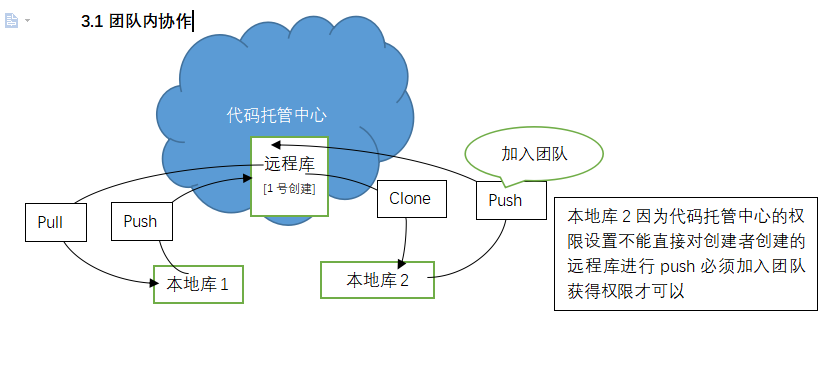
**2.3外网环境下**：

Github

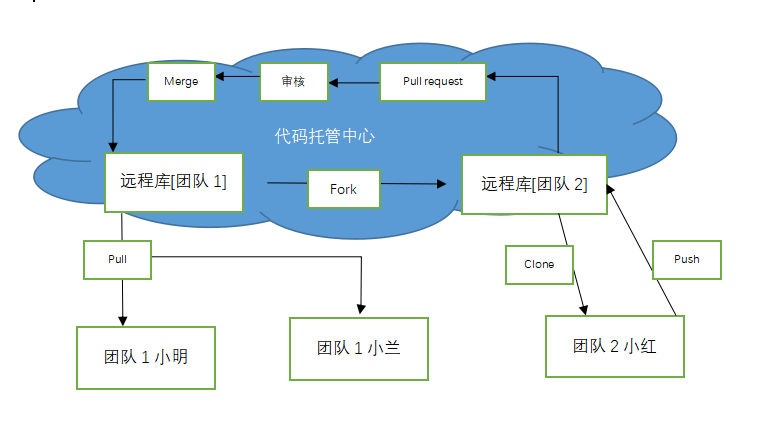
码云

### 3.本地库和远程库

**3.1团队内协作**



**3.2团队外协作**



### 4.Git 命令行操作

**4.1本地库初始化**

首先创建一个目录作为本地库mkdir

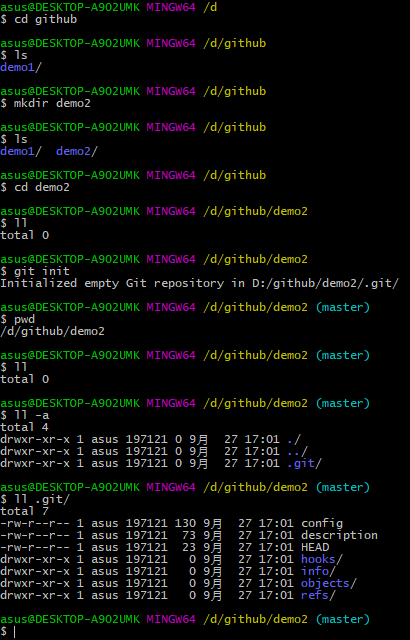
然后进入目录执行git init

用ls -a查看此目录[-a可以查看隐藏目录]发现初始化后自动创建了git目录

[注：在查看目录的时候如果我们看到有’.’加目录名时表明此目录是隐藏目录，可以用ls –a命令查看]

命令：git init

效果：



注意：.git目录存放的是本地库相关的子目录文件，不能删除或胡乱修改

**4.2设置签名**

**4.2.1形式**

用户名：username

Email地址：[XX.@qq.com](mailto:XX.@qq.com)

**4.2.2作用**

区分不同开发人员身份

注意：这里设置的签名和登录远程库(代码托管中心)的账号密码没有关系

命令

**4.2.3项目级别[仓库级别]：仅在当前本地库范围有效**

Git config user.name username

Git config user.email [xx.@qq.com](mailto:xx.@qq.com)

信息保存在.git里的config里



**4.2.4系统用户级别：登录当前操作系统的用户范围（范围大）**

Git config --global user.name username

Git config --global user.email [xx.@qq.com](mailto:xx.@qq.com)

信息保存位置为 ～/.gitconfig中

优先级：就近原则，项目优先级优于系统用户级别，二者都有时采用项目级别的签名，如果都没有是不允许的。

**4.3基本操作**

**4.3.1状态查看**

git status

查看工作区，暂存区状态

**4.3.2添加操作**

git add 文件名

将工作区新建或修改添加到暂存区

git rm –cached 文件名

将暂存区的文件拉出来

**4.3.3提交操作**

git commit 文件名

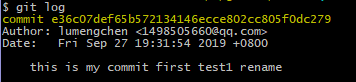
将暂存区提交到本地库

git commit –a 文件名（针对的是已经提交后修改的文件的再次提交，可以直接用commit但是不经过暂存区而直接提交是无法撤回的）

git –m ‘‘my first commit’’ 文件名（用 –m 的作用是提交的时候加上信息，不用再vim里编辑了）

**4.3.4查看历史记录**

git log



多屏显示控制方式

空格向下翻页

B向上翻页

Q退出

git log –pretty=oneline [一行显示]



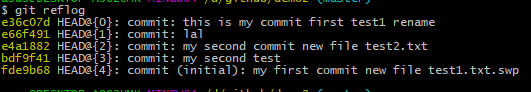
git log –oneline



git reflog(显示了head)



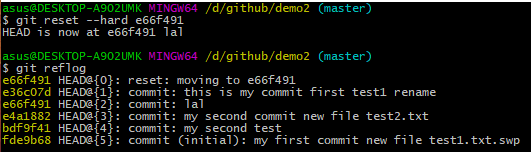
**4.3.5前进后退**



基于索引值操作（可以前进后者后退）

git reset –hard 索引值（eg:上图e66f491这个值是每次提交都会生成的一个唯一哈希值相当于操作的主键）

操作后变为



回到了e66f491这个版本，可以让HEAD回到任意一个版本

使用^符号(只能往后退)

git reset --hard HEAD^ 退一步

git reset --hard HEAD^^ 退两步

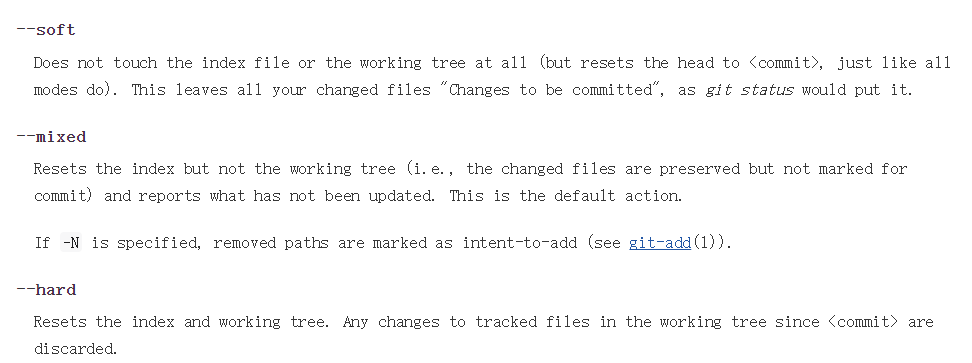
git reset --hard HEAD^^^ 退三步

使用～符号(^的代替运用)

git reset --hard HEAD ～n 后退n步

**4.3.6 reset 命令 的三个参数 soft hard mixed**

用命令 git help + reset 可以查看reset的具体功能其中三个参数解释如下（index file 暂存区 working tree 工作区）



--soft：仅仅在本地库移动HEAD指针

工作区

暂存区

本地库

上图的关系就是如果使用soft后退了一步仅仅是本地库往后移动了一步，而其他两项没有移动，当使用cat查看文件的时候文件并没有改变，所以以本地库为基准好似暂存区和工作区向前移了一步所以当用git status 的时候这个文件时绿色警告(相当于进行了git add 而没有进行commit 如果将此文件进行commit操作就会恢复，让本地库上移)

--mixed：

在本地库移动HEAD指针

重置暂存区

工作区

本地库

暂存区

上图的关系就是如果使用mixed后退了一步本地库和暂存区往后移动了一步，而工作区没有移动，用cat查看时文件依然没有改变，所以以本地库为基准好似工作区向前移了一步所以当用git status 的时候这个文件是红色警告（相当于没有进行git add的状态）

--hard：

1.在本地库移动HEAD指针

2.重置暂存区

3.重置工作区

工作区

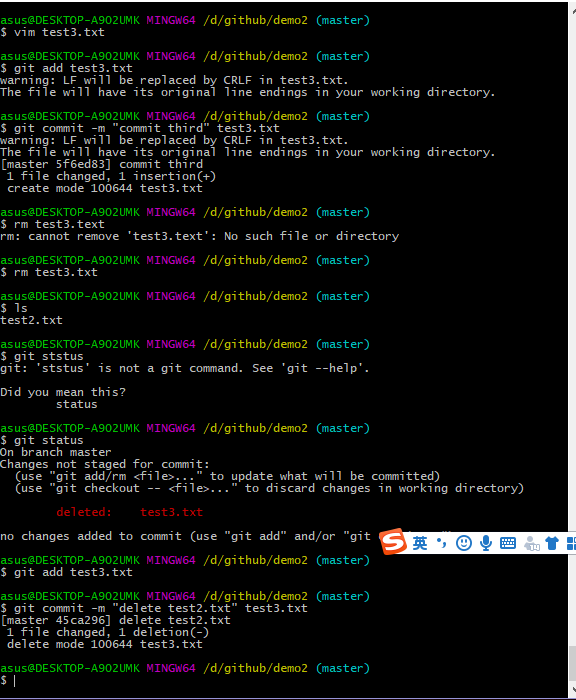
本地库

暂存区

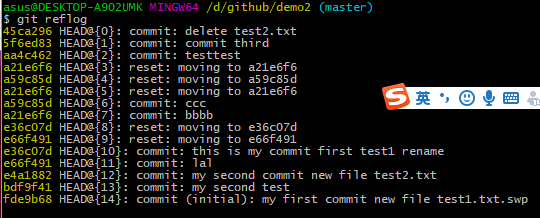
使用hard时三个区域同时更新，相当于是同步的，所以在git status 的信息里没有什么变化

**4.3.7删除文件恢复**

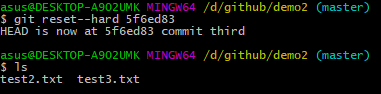
创建test3.txt并用rm命令删除并add以及commit如下



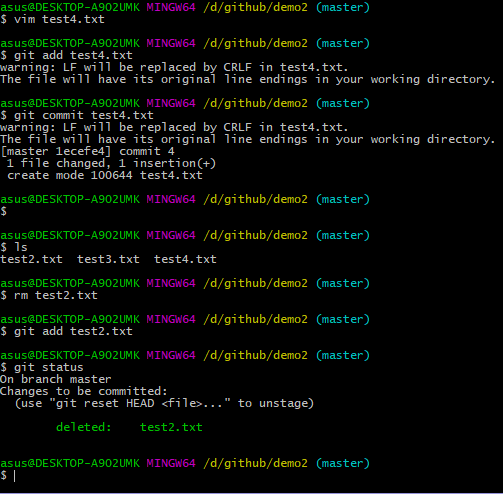
此时查看log操作日志查看版本，可以看到我们刚才删除test3.txt的操作记录，git记录的版本是所有的即使是回退到上一个版本也是新增了一个操作，而当我们想要恢复误删的文件目录时只需要回到它还存在的版本就行了，比如回到删除操作前的一个版本5f6ed83



恢复命令git reset –hard 5f6ed83，操作后如下图



创建test4.txt 执行rm命令并执行add但是并没有执行commit操作



与上面相同

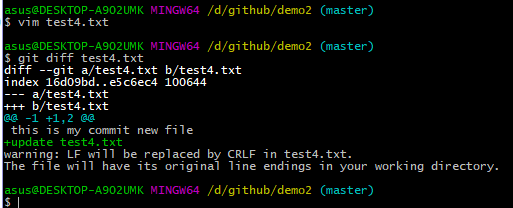
前提：删除前，文件存在时的状态提交到了本地库

操作：git reset --hard（当前本地库版本）

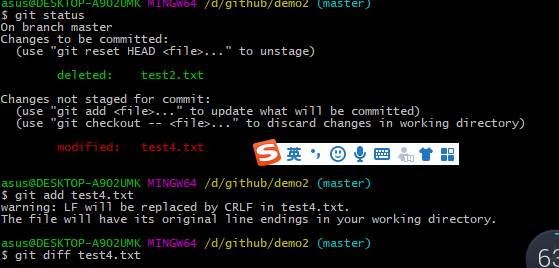
**4.3.8比较文件差异**

git diff 文件名（和暂存区比较）

在test4.txt文件中用vim命令添加了一行’update test4.txt’字样，然后用git diff test4.txt进行文件比较（vim编辑输入i进行编辑状态按esc退出编辑状态，然后按:wq进行保存）

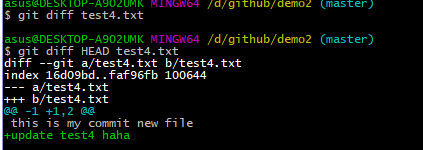


因为此命令是和暂存区的文件比较所以当用add命令把修改的文件添加到暂存区时再执行git diff 时就没有变化



git diff HEAD 文件名

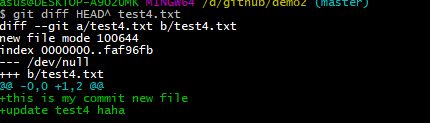
接着我们使用git diff HEAD test4.txt命令



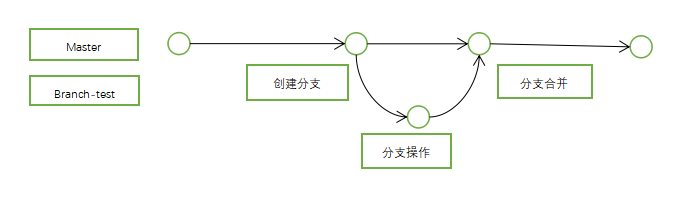
此时又出现了了差异说明此时是与本地库的test4.txt进行比较，因为还没有执行commit操作，本地库文件还没有同步

git diff HEAD^ test4.txt

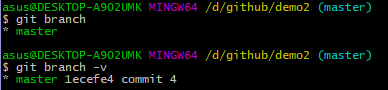
此命令是让此文件与上一个版本的test4.txt作比较



**4.4分支管理**

**4.4.1查看分支**

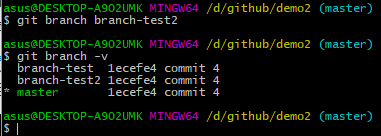
git branch [-v]



**4.4.2创建分支**

git branch 分支名

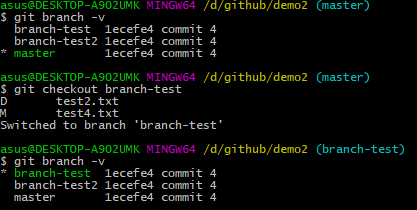
在master分支基础上创建两个分支 branch-test 和branch-test2



**4.4.3切换分支**

git checkout 分支名

由master分支切换到branch-test分支上，注意红色矩形框里的信息变化他表示处于的当前分支由原来的master换到了branch-test并且查看分支的时候当前分支是绿色的前面有\*



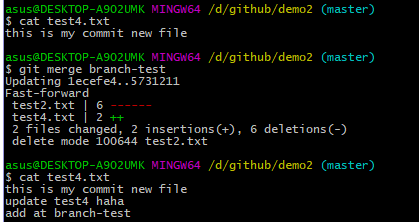
**4.4.4合并分支**

当在分支branch-test修复了一条bug后，branch-test的版本相对于master就前进了，如果想要在master也生效就要把branch-test合并到master上

要想把branch-test做的修改合并到master上

第一步：切换到master（git checkout 分支名:git checkout master）

第二步：执行合并( git merge 分支名：git merge branch-test)

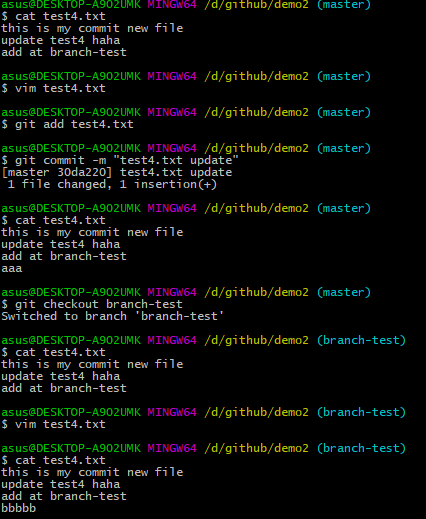


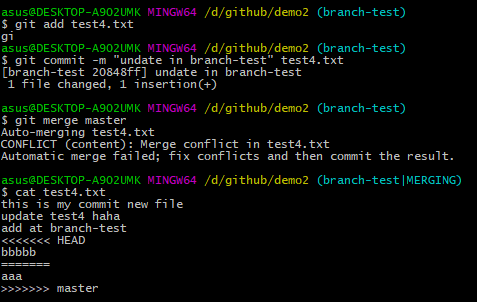
结果：合并前master分支的test4.txt的内容为this is my commit new file

合并后master中test4.txt的内容就发生了改变，就是在branch-test分支做的修改操作后的文件内容。

**4.4.5合并冲突**

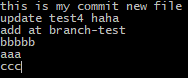
当一个分支进行了一个文件的修改，而另一个分支也进行了同一文件的修改，当我们合并这两条分支的时候，因为两条分支对同一文件进行了修改，所以合并的时候就会发生冲突



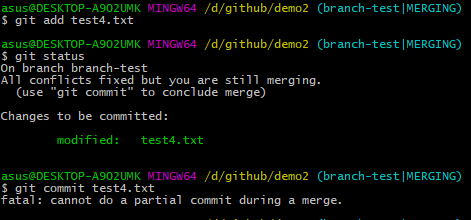


结果：当在进行合并的时候系统提示test4.txt有合并冲突，自动合并失败，自动调节冲突并commit结果，并且在查看test4.txt文件内容的时候，系统为我们自动添加了一些特殊符号，而这些特殊符号就是冲突的存在，此分支的bbbb和master分支里的添加aaaa操作，test4.txt的最终内容需要我们手动编辑最后提交。

第一步：删除特殊符号，并且编辑我们想要的的内容，如下图我们保存了两个分支的修改并且在此基础上有添加了cccc这一行



第二步，执行add和commit（直接用git commit 而不是git commit 文件名）



注：当用commit提交时不能加上平常的文件名而是直接使用git commit 如果加上文件名会报错

### 5基本原理

**5.1git保存版本的机制**

**5.1.1 集中式版本控制工具的文件管理机制（如svn）**

以文件变更的方式存储信息，这类系统将他们保存的信息看作是一组基本文件和每个文件随时间逐步积累的差异。（增量保存）

**5.1.2 git的文件管理机制**

Git相当于一个快照流，每次提交修改时git都会对当前的全部文件制作一个快照的索引为了高效，如果文件没有修改，git不在重新存储文件而是保留一个链接指向之前存储的文件

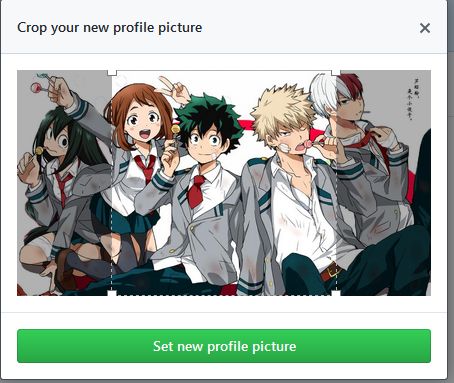
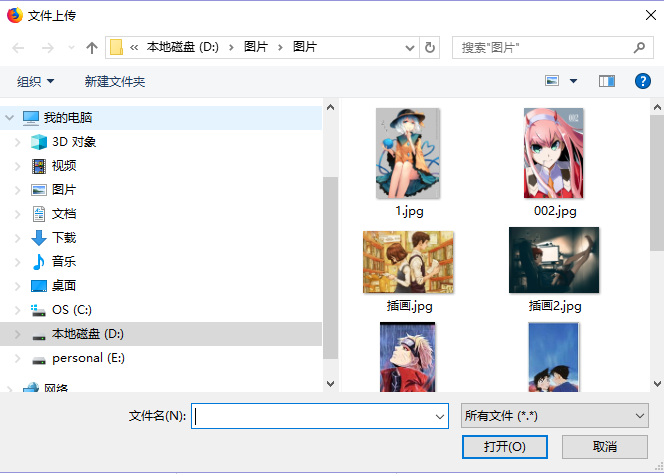
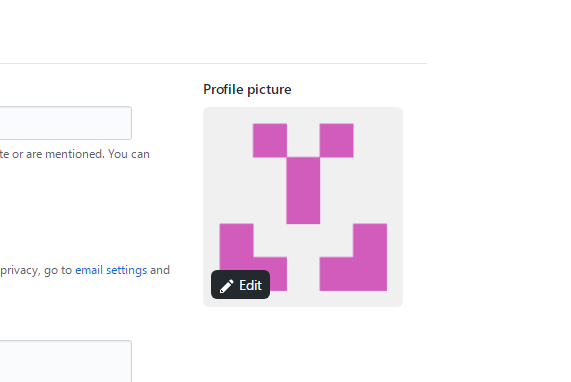
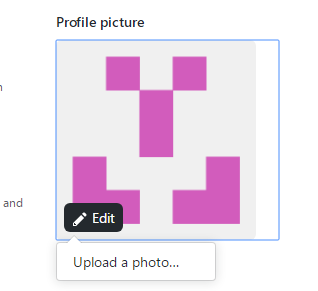
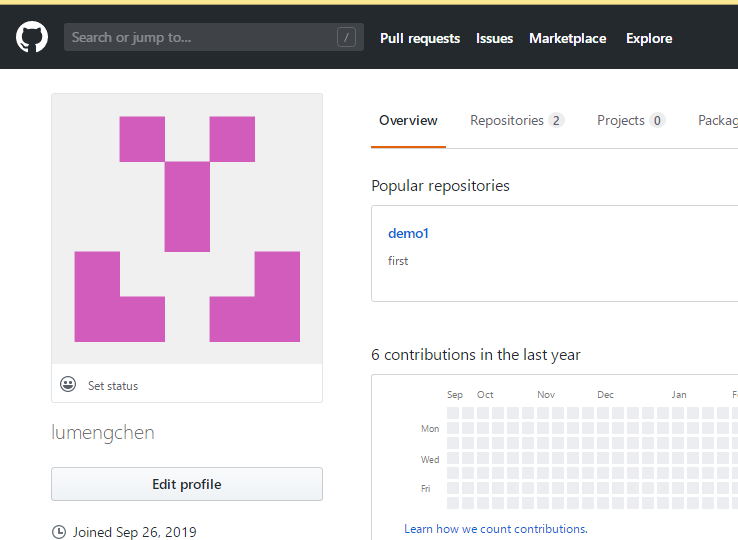
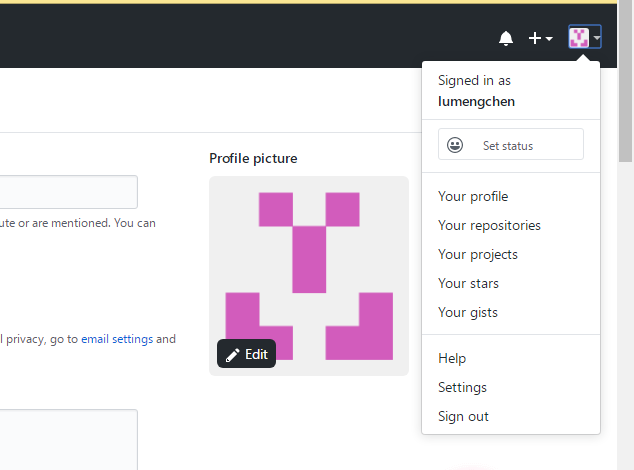
**5.2分支管理机制**

创建一个分支即复制了一份指针指向当前版本，而不是复制整个文件，当在创建的分支上做操作时，才数据才真正分叉存储，当切换分支的时候，只是让head指针指向其他分支，所以效率很高

### 6Github

**6.1注册github账号**

**6.2修改头像**

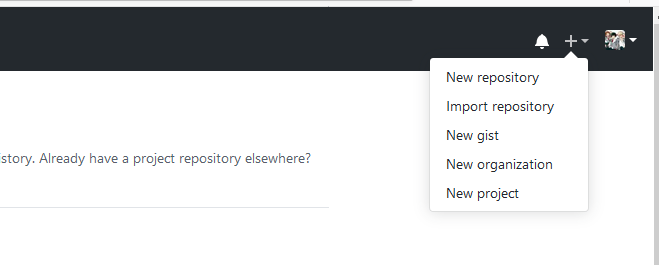


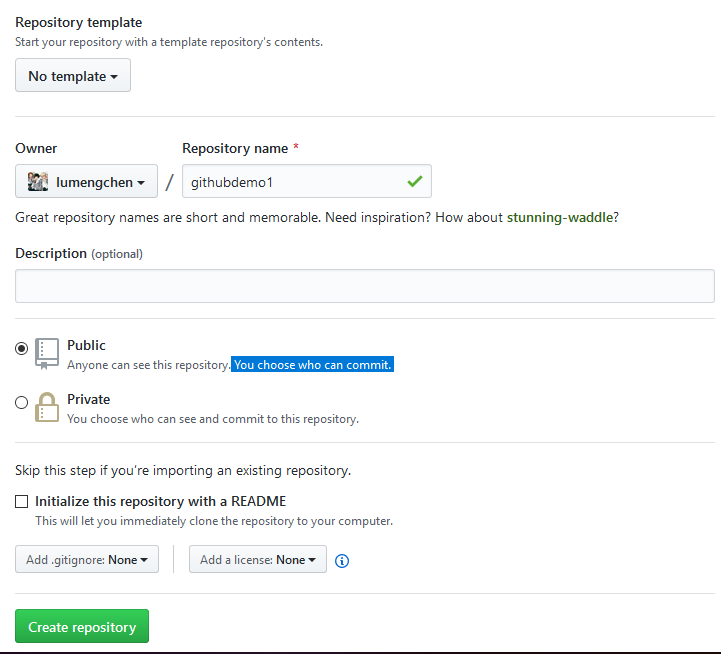
注：修改头像是可能和浏览器的兼容也有关系推荐使用谷歌和火狐，第一次用搜狗浏览器切换不了

**6.3创建本地库和远程库**

创建本地库：在bash里用命令mkdir创建一个本地库的文件夹，或者直接手动新建文件夹文件夹名为gitdemo1作为本地库

创建远程库：

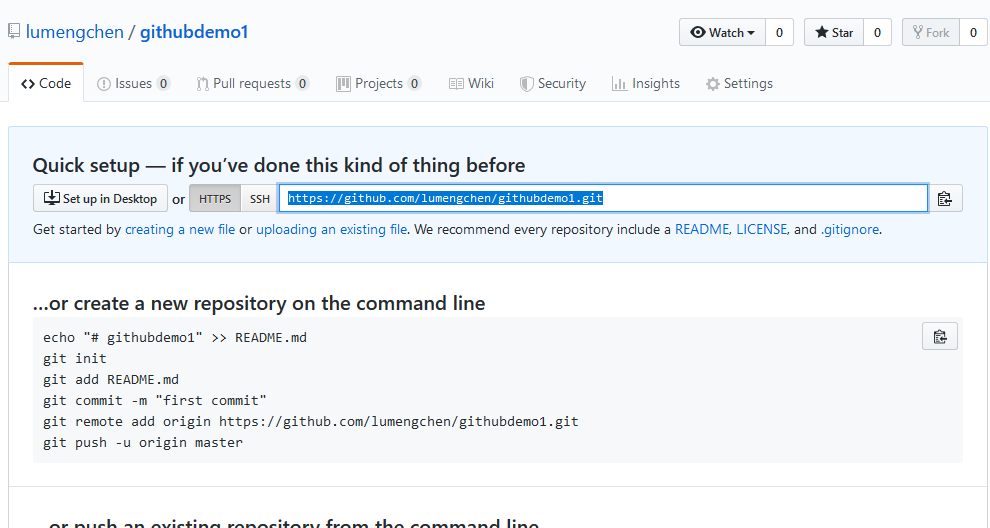




Private收费

远程库名字为了和本地库相对应，不容易混淆所以起了对应的名字，可以不一样

**6.4创建远程库别名**

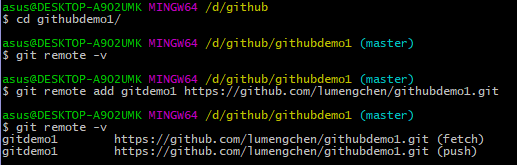


进入刚刚创建的远程库，可以查看远程库的地址

为了方便管理我们可以给远程库创建别名方便使用

查看别名命令：git remote -v

创建命令：git remote add 别名 地址

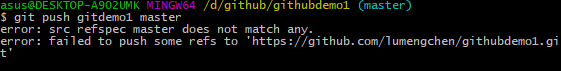


此次我为此地址设置的别名为gitdemo1

再次查看别名的时候我们可以看到fetch(用来取回)和push（用来推送）

**6.5从本地库推送到远程库**

命令： git push -u giedemo1 [分支名]



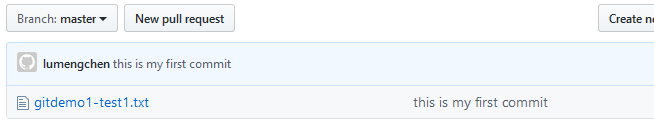
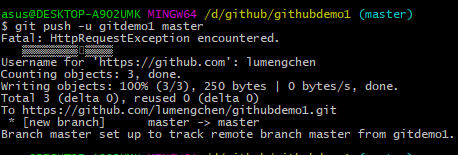
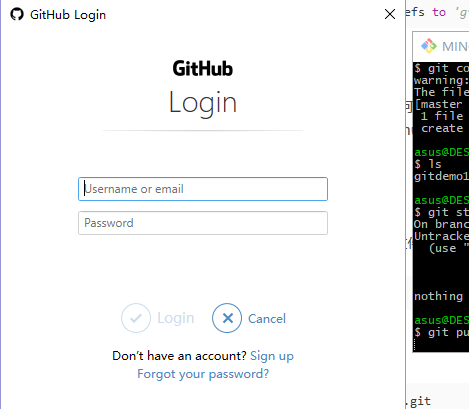
错误提示:上面error的信息为

Error：SRC ReFSPEC主控器不匹配任何

Error：未能将某些引用推送到'https://github.com/lumenghcen/githubdemo1.git’

说明仓库此时是空的没有什么要推送的所以就要查看是否创建了文件，或者文件进行了add却没有进行commit到本地库，我的原因是我创建的是新的库，库里还没有创建任何文件，所以在push的时候发生错误

当创建文件或者把文件直接拉进本地库中后进行add以及commit操作后再进行push会出现以下登录界面就是输入github账号

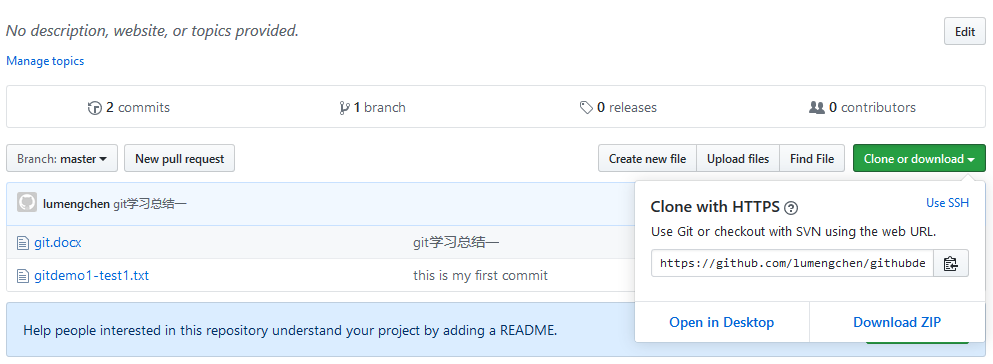


刷新网页可以看到gitdemo1-test1.txt已经提交到远程库了

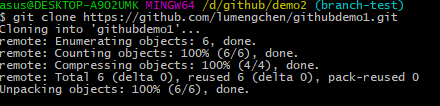
**6.6克隆操作**

克隆操作是指吧远程库的内容克隆到本地库(比如团队里的另一个人想要获取刚才创建的远程库里的内容，此时他就可以使用克隆操作)

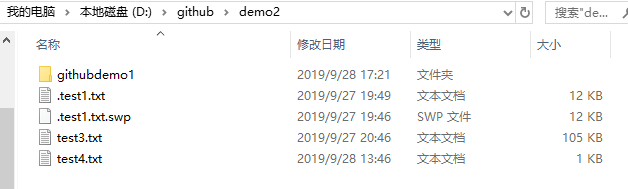
命令：git clone 远程库地址



点击clone or download 下面会出现此远程库的地址，复制地址，执行克隆命令



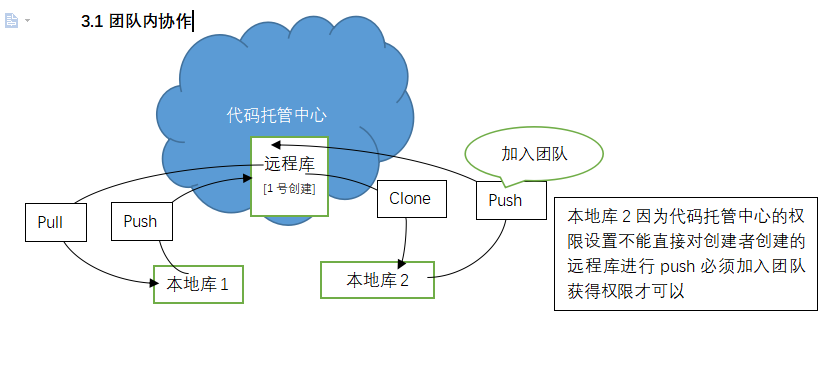
克隆成功后打开demo2这个本地库查看一下，发现除了之前自己添加的文件内容，新增了一个githubdemo1这个文件夹，这就是远程库githubdemo1的内容



克隆效果：完整的把远程库下载到本地变成本地库，并且自动进行了别名创建且自动初始化本地库

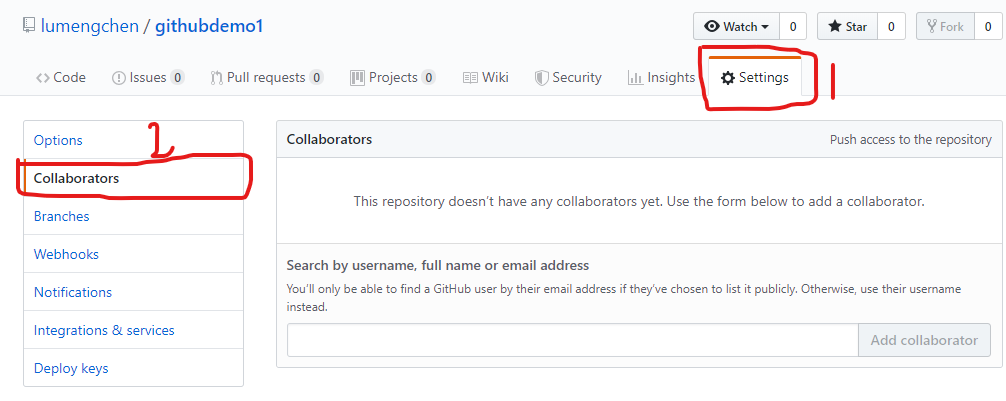
注意：如果此时demo2本地库的使用者想要对此远程库进行push是失败的，因为远程库还没有进行团队加入，还没有权限

**6.6团队协作开发**



**6.6.1加入团队成员**

第一步：在github中点击需要加入团队成员的仓库的settings->collaborators之后在选项框中填写你要邀请的人的github账号



第二步：添加完之后会弹出Copy invite link 发送给这个成员

第三步：等待队员访问我发送的连接点击accept invitation接受邀请即可

成员加入后他就可以对远程库有写的权限，就可以进行push操作了

**6.6.2远程库修改的拉取**

**Pull=fetch+merge(远程拉取操作是只读操作不需要用户名密码登录)**

分步拉取（fetch+merge）

git fetch 远程库别名 远程分支名

如：git fetch gitdemo1 master（此时下载了远程库但是还本地库还没有更新）

git checkout gitdemo1/master （切换到远程的master再查看文件时修改后的文件内容）

git checkout master（返回当前本地master）

git merge gitdemo1/master（把远程的master合并到本地master）

一步拉取(pull)

Git pull 远程库别名 远程分支名

当查内容的时候，此时已经是远程修改后的文件了

注意：当我们所做的操作并不是很复杂，内容相差不大可以直接用pull直接一步拉取，而当我们所拉取的修改后的内容与当前内容相差很大时我们可以先用fetch拉取过来，然后查看一下准确无误后我们就可以用merge合并，一面发生不必要的冲突

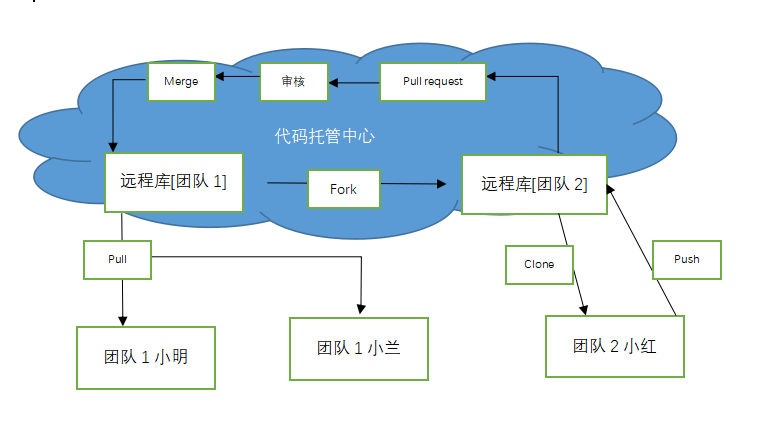
**6.6.2协同开发冲突**

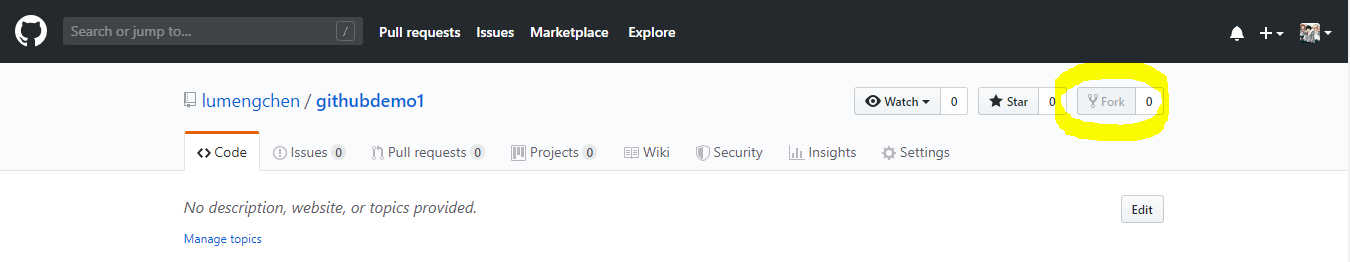
**如果不是基于github远程库的最新版本所做的修改，不能被推送（push）必须先拉取，用分支冲突解决操作即可**

比如对远程库进行修改一个人进行了push另一个人就不能进行push只能先进行pull拉取后才能进行pull，而当另一个人拉取内容后因为可能这两个人同时修改了同一个文件，所以会产生冲突，当用pull命令后会进入master|merging，此时就需要我们手动合并冲突文件（删除无关符号，修改好最后的内容）add以及commit（注 git commit 后面不加文件名）

完成后就可以进行push了

### 6.7跨团队协作





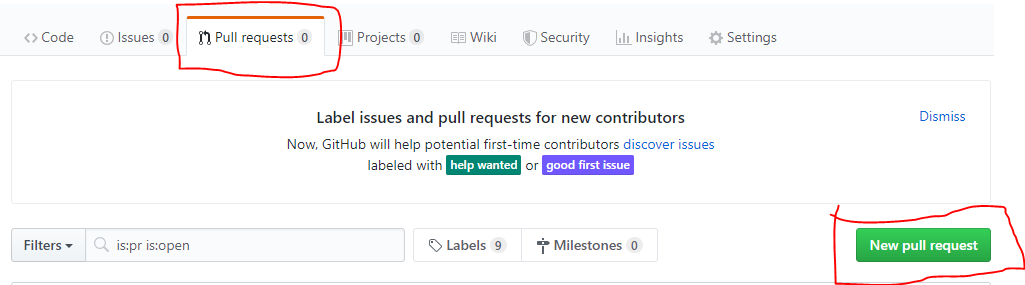
第一步：小红访问团队1的远程库，然后点击fork

第二步：克隆操作fork后的远程库 git clone 地址

第三步：进入克隆来的团队1的库里进行修改并进行commit提交

第四步：推送操作 git push 别名 分支名

第五步：发起pull request



第六步：creat pull request

第七步：团队一远程库的创建者小明的仓库的pull request 会看到小红pull request的请求

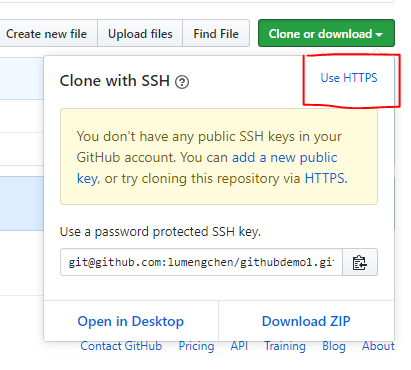
第八步：（审核操作）小明可以和小红在此窗口的conversation进行对话，可以在commits查看小红作了哪些提交，在files changed可以查看小红具体作了哪些修改。

第九步：审核完成后进行merge pull request合并代码-->附加操作备注-->confirm merge

第十步：把远程库拉取到本地就是小红跨团队帮助后的仓库

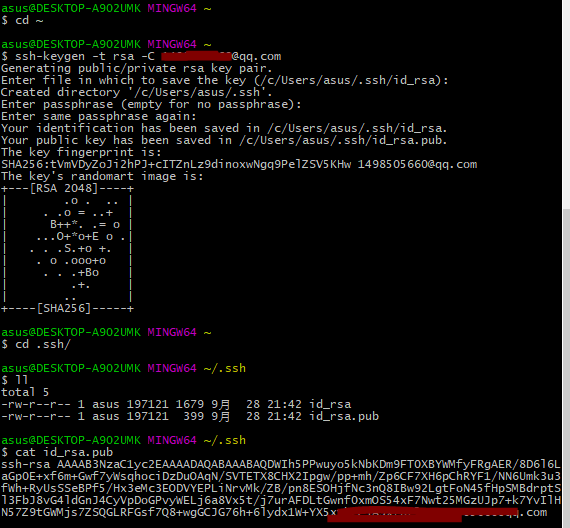
### 6.7ssh免密登录

进入github的仓库查看地址的时候有https和ssh两种地址，当我们用https地址方式时如果系统没有记录用户名密码，每次push的时候都要提供用户名密码才能提交，用ssh方式就可以避免每次都要提交用户名密码，但是ssh有局限，只能为一个账号设置



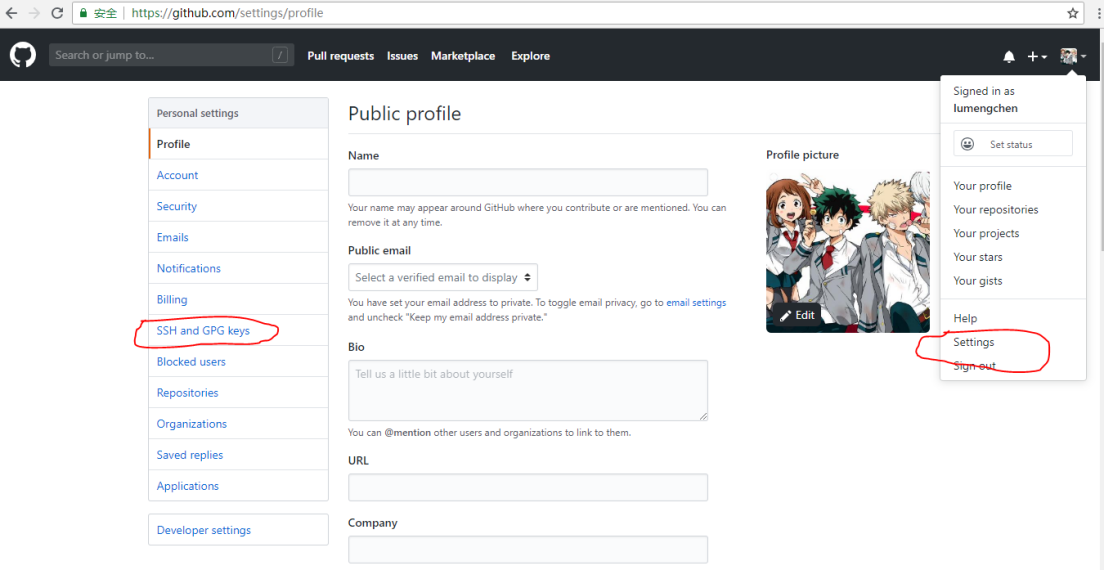
第一步：在bash中操作进入本地库看看有没有.ssh文件，如果没有就执行下面命令

ssh-keygen -t rsa -C github账号的邮箱，点击回车，进入.ssh文件目录，用cat查看.pub文件并复制文件内容

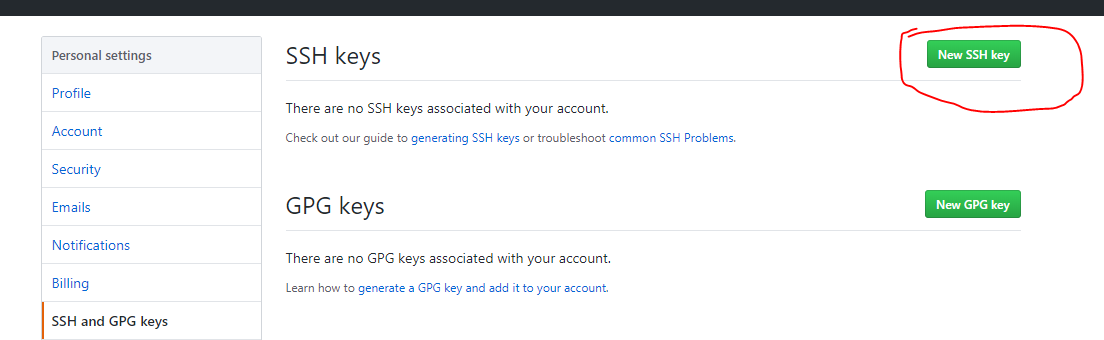


第二步：进入github，点击右上角菜单进入settings然后进入

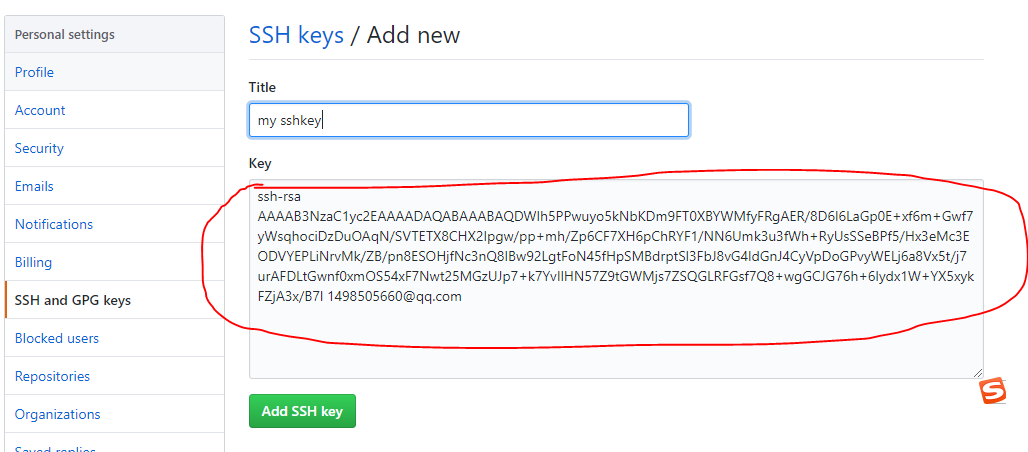
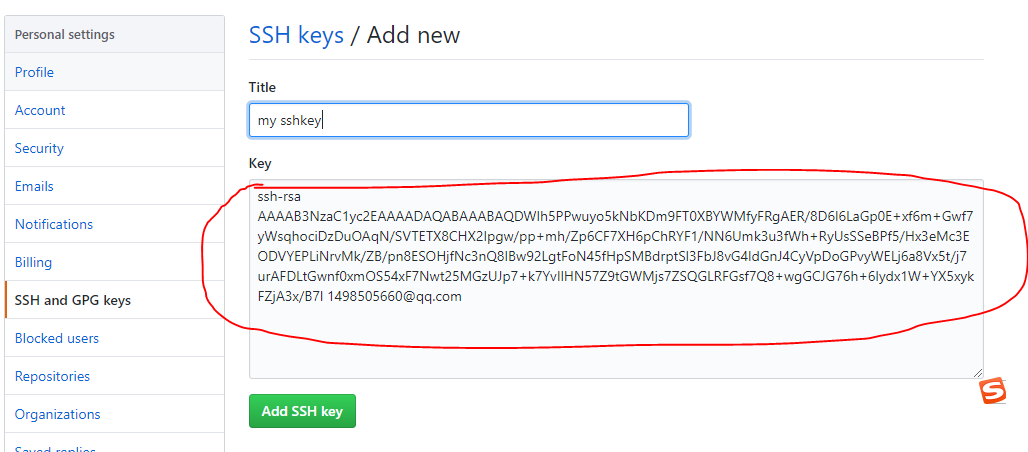
SSH and GPG keys 选项



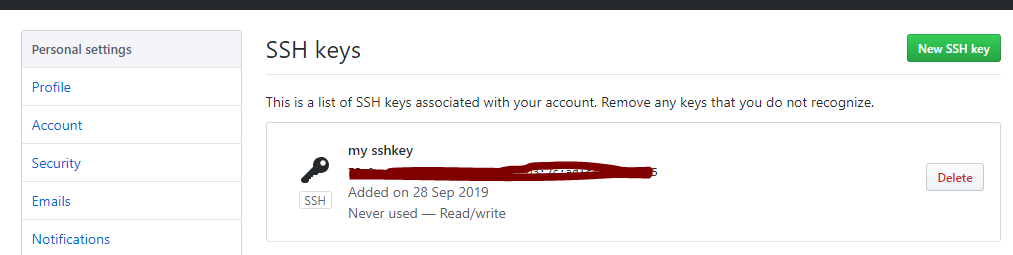
第三步：点击new SSH key



第四步：把复制文件的内容粘贴到key中，tittle可以自己命名

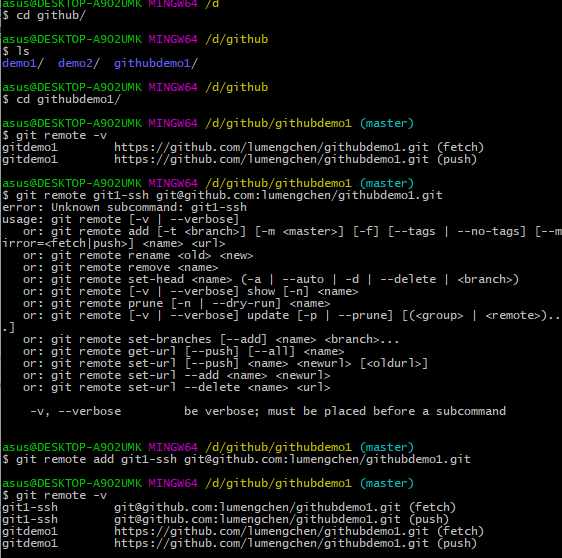


第五步：点击Add SSH key完成创建

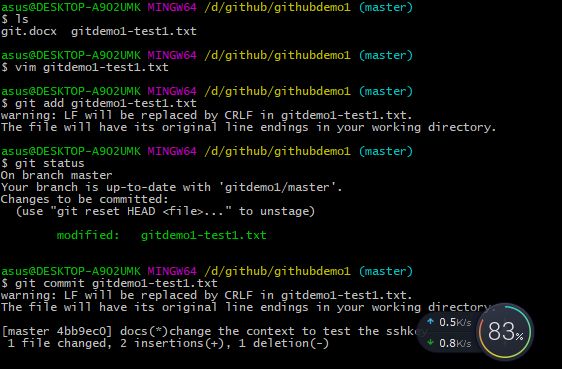


第六步：本地库测试ssh的使用

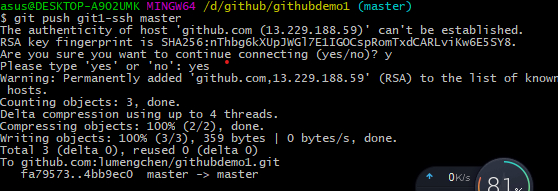
1. 创建于ssh地址相对应的别名git1-ssh



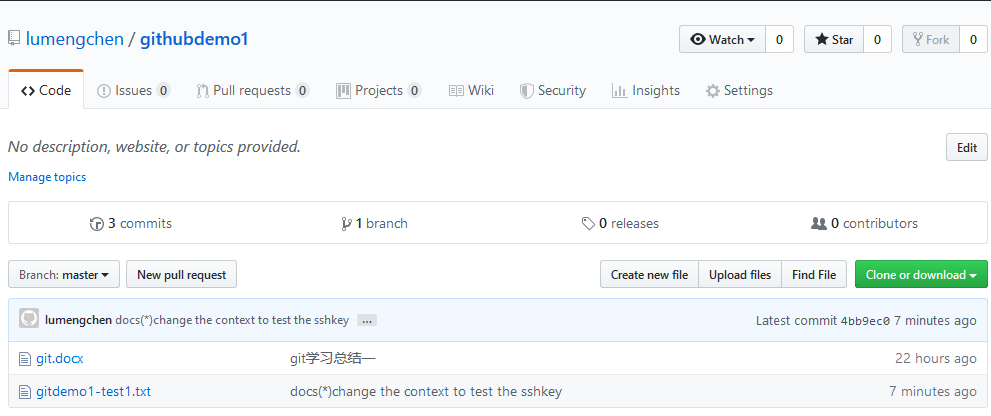
2编辑本地库内容并commit



3push 操作用ssh推送到远程库，提交的时候并没有要求输入密码



4.查看远程库的更新可以看到gitdemo1-test1.txt的改动提交到了远程库



### 6.8 commit message 规范

**6.8.1格式**

<type>(<scope>): <subject> #id（冒号和下一个单词之间有空格，#之前有一个空格）

Eg: docs():完成学习文档的编写 #1

**6.8.2 type**

type：用于说明commit的类别，必填并且只允许有以下7个标识

feat：新功能（feature）

fix:修补bug

docx：文档（documentation）

style：样式（不影响代码运行的变动）

refactor：重构（既不是新增功能，也不是修改bug的代码变动）

test：构建过程或辅助工具的变动

**6.8.3 scope**

Scope 用于说明commit影响范围，比如数据层，控制层，视图层，视项目不同而不同（取值只能从该项目的label中取功能模块的值）

**6.8.3 subject**

**是commit的简单描述，建议使用精简语言描述，最好不超过50个字符，另外如果是内部项目可以使用中文描述，方便交流**

**6.8.4 id**

**是任务的编号，因此每个commit的提交都必须有对应的编号 id+#是能够让github识别并绑定到现有的任务中**