**1.简介**

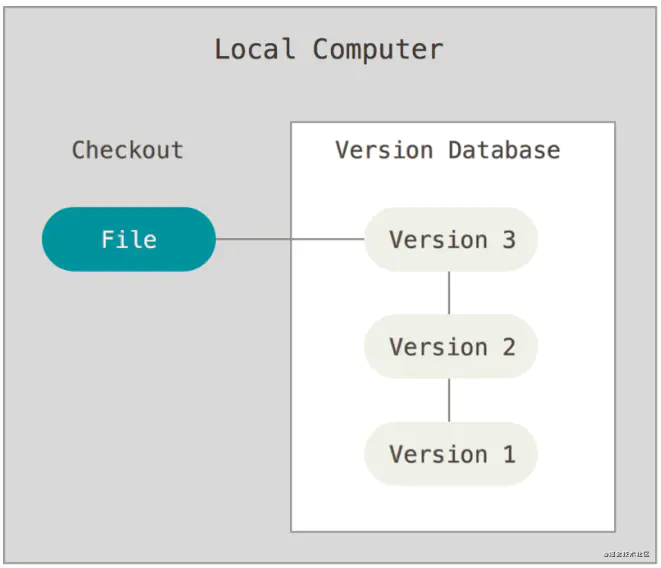
**1.1 关于版本控制**

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

**1.2 版本控制发展史**

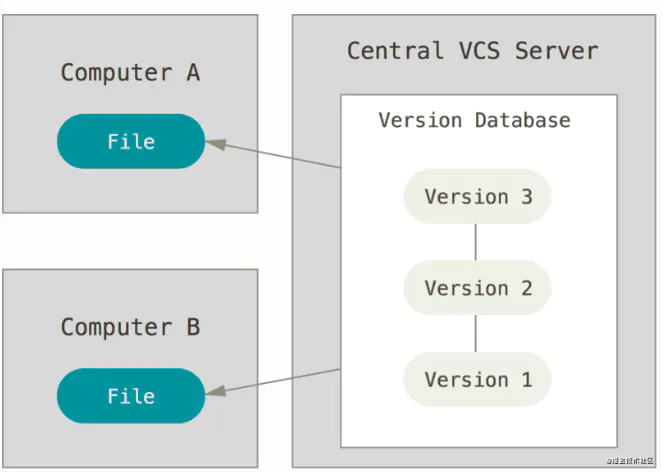
**1.2.1 本地版本控制**

记录文件每次的更新，可以对每个版本做一个快照，或是记录补丁文件，适合个人用，如RCS。



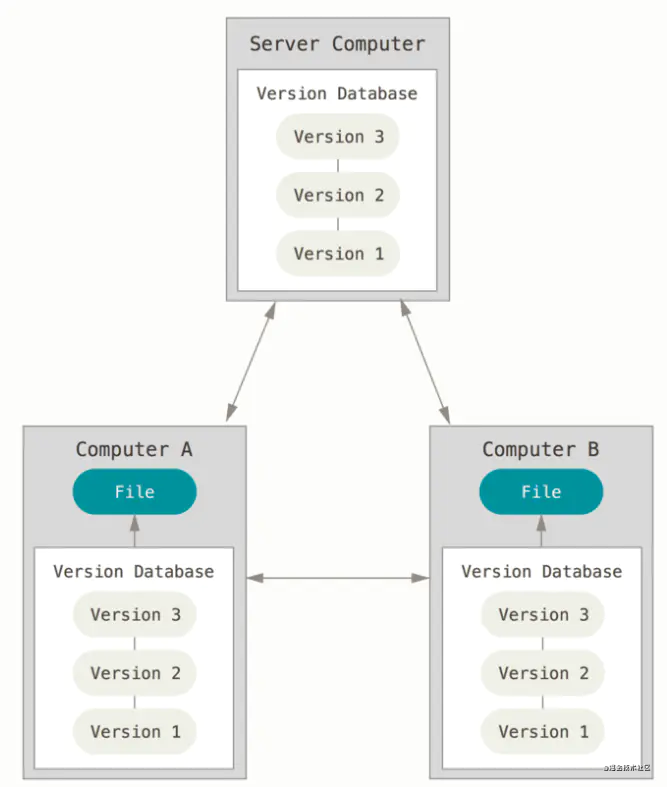
**1.2.2 集中式的版本控制**

所有的版本数据都保存在服务器上，协同开发者从服务器上同步更新或上传自己的修改

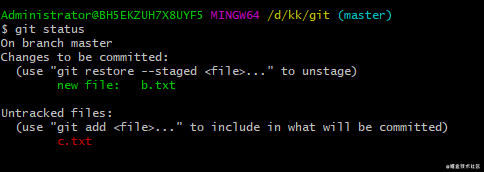
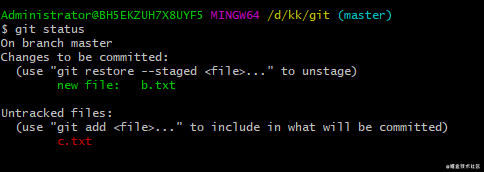
  
所有的版本数据都存在服务器上，用户的本地只有自己以前所同步的版本，如果不连网的话，用户就看不到历史版本，也无法切换版本验证问题，或在不同分支工作。而且，所有数据都保存在单一的服务器上，有很大的风险这个服务器会损坏，这样就会丢失所有的数据，当然可以定期备份。代表产品：SVN、CVS、VSS

**1.2.3 分布式版本控制系统**

所有版本信息仓库全部同步到本地的每个用户，这样就可以在本地查看所有版本历史，可以离线在本地提交，只需在连网时push到相应的服务器或其他用户那里。由于每个用户那里保存的都是所有的版本数据，只要有一个用户的设备没有问题就可以恢复所有的数据，但这增加了本地存储空间的占用。



**1.3 什么是git**

**Git是目前世界上最先进的分布式版本控制系统。**

优点：

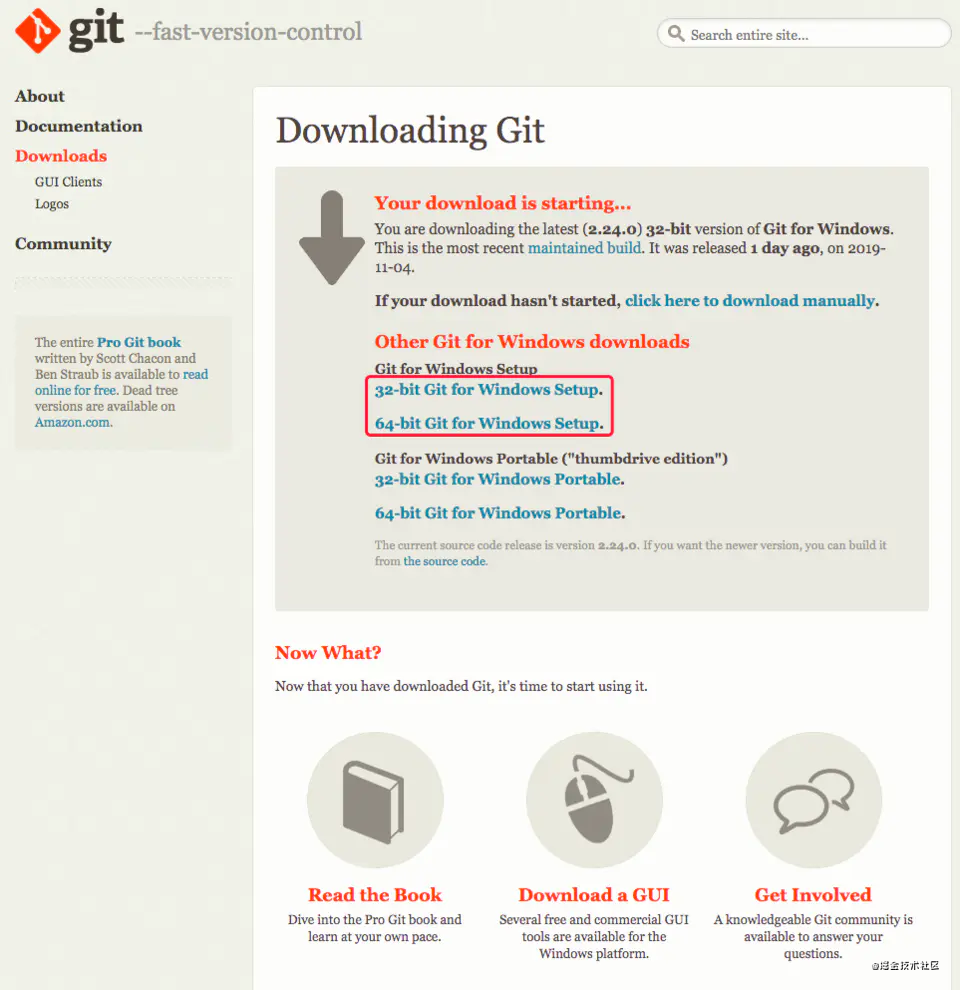
* 适合分布式开发，强调个体。
* 公共服务器压力和数据量都不会太大。
* 速度快、灵活。
* 任意两个开发者之间可以很容易的解决冲突。
* 离线工作  
  缺点：
* 模式上比SVN更加复杂。
* 不符合常规思维。
* 代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

**2.本地操作**

**2.1 安装git**

**2.1.1 window安装**

在官网下载安装文件,双击打开，然后一直“NEXT”就好。 下载地址如下：https://git-scm.com/download/win

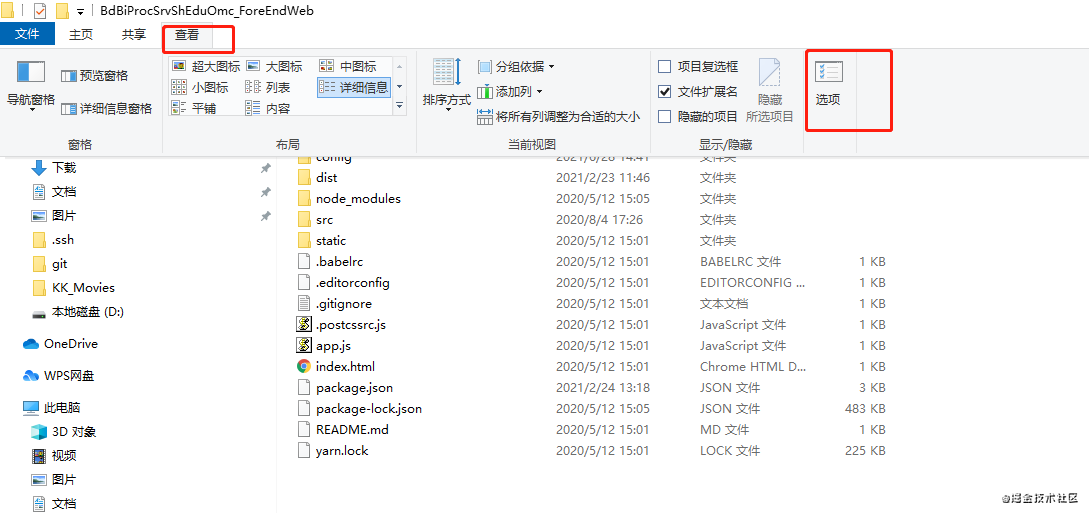


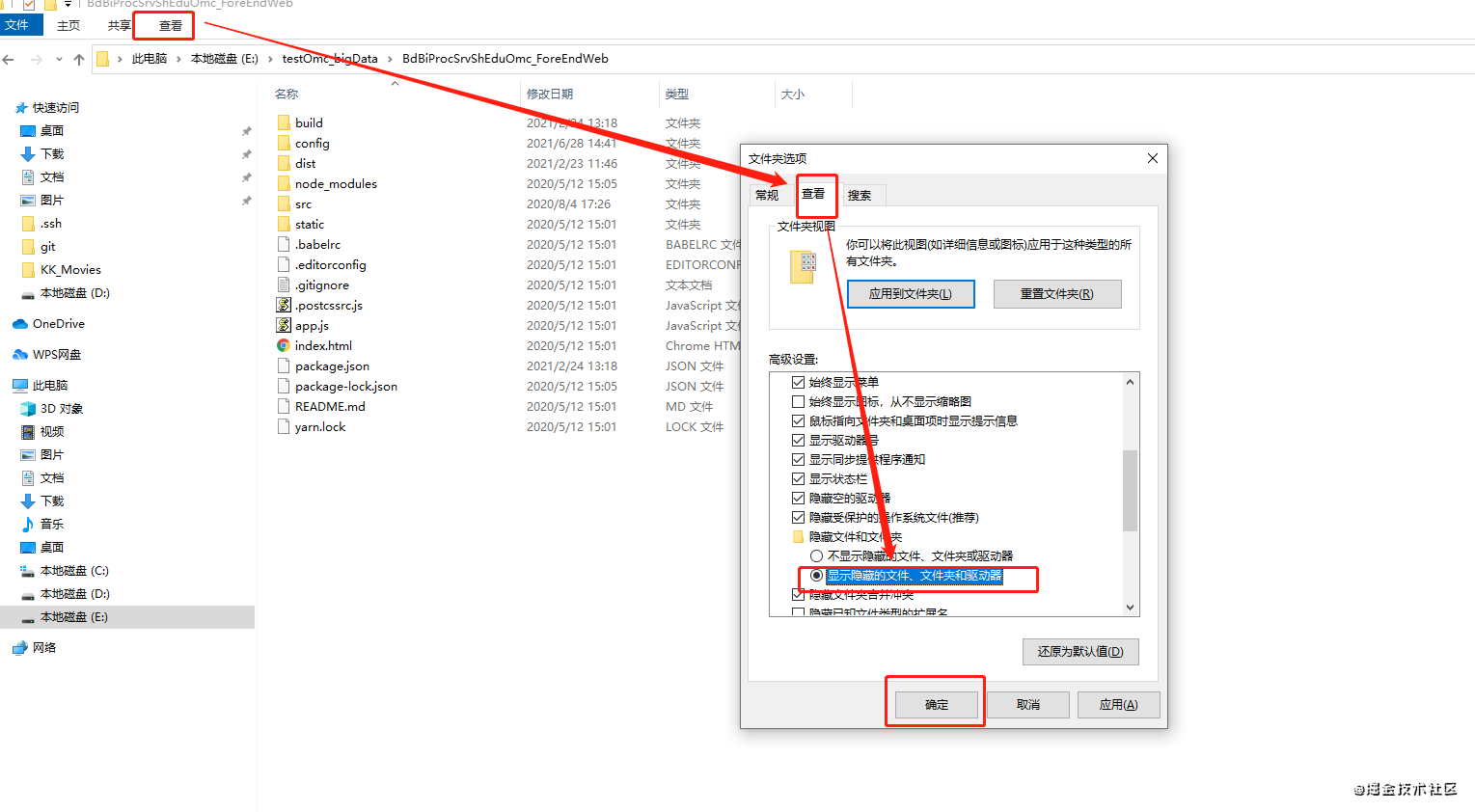
**2.2 初始化一个本地仓库**

我们现在要初始化一个本地仓库:

git init

创建之后，目录下会多一个".git"的文件夹,这个项目git的配置都在这个文件夹下面。***注意：如果这个.git文件没有显示出来，需要我们把隐藏文件设置显示出来即可***，一般我们不需要修改此文件夹下的内容。





**2.3 新增修改文件并且提交到git（以及git三个工作区）**

先新建一个文件，在进到改文件打开 git Bash Here 接着执行如下

**git add .** // 更新到暂存区，**.号代表添加当前路径下所有文件**

git commit -m "build: 初始化项目" // 更新到资源库, build\*\*为log信息

git commit <file name> -m "build: 初始化项目" // **提交指定文件更新到资源库**

执行后出现了个错误：



这里是说我们得告诉git我们是谁，所以执行一下下面的命令

git config --global user.email "xiaoyuzhou@example.com"

git config --global user.name "xiaoyuzhou"

再次执行 commit 命令

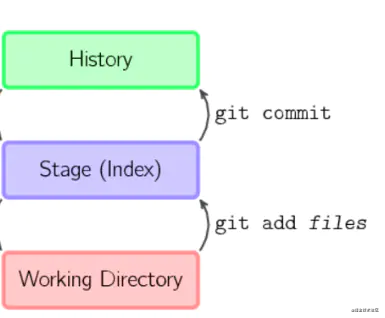
git commit -m "build: 初始化项目"

这样文件就提交上去了。

这里简单说明一下：

Git本地有三个工作区域：**工作目录（Working Directory）、暂存区(Stage/Index)、资源库(Repository或Git Directory)**。

* Workspace：工作区，就是你平时存放项目代码的地方
* Index / Stage：暂存区，用于临时存放你的改动，事实上它只是一个文件，保存即将提交到文件列表信息
* Repository：仓库区（或本地仓库），就是安全存放数据的位置，这里面有你提交到所有版本的数据。其中HEAD指向最新放入仓库的版本



总结：

* 我们使用 git add 工作区的文件添加到暂存区。
* 使用 git commit -m "xxxx" 将暂存取的内容推到仓库。里面的内容信息可以参照 “Conventional Commits” 书写规则。
* 书写良好的 commit message https://loveky.github.io/2018/06/04/write-good-commit-message/

**2.4 查看历史纪录和状态**

我们可以使用 git status 查看状态, 使用 git log 查看历史纪录。

我们来看下面的栗子

我们先建立三个文件

第一步

vi a.txt // 新增 a.txt 文件

git add .

git commit -m "feat: 新增a.txt文件"

第二步

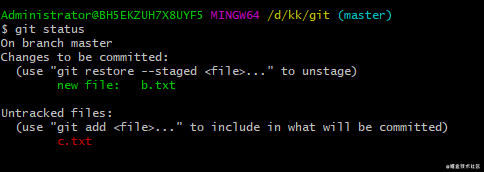
vi b.txt // 新增 b.txt 文件

git add b.txt

第三步

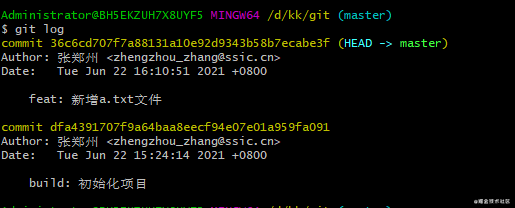
vi c.txt // 新增 c.txt 文件

我们使用 git status 查看状态



我们可以看到 a.txt 已提交到版本库里面，所以a.txt这里看不到；b.txt 在暂存区中， c.txt在工作区。

我们使用 git log 查看历史纪录：（**git log 查看到的必须添加是执行了git commit 到资源库(Repository）**）



**2.5 版本回退**

**2.5.1 工作区的回退**

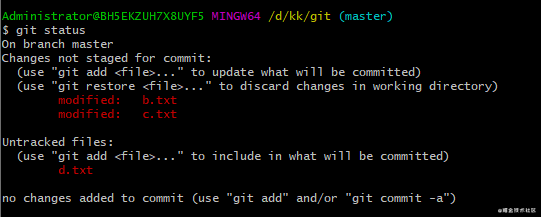
我们先把上面栗子中的文件提交到资源库：

git add .

git commit -m "feat: 新增b,c模块"

然后在重新分别对b、c文件进行修改，在再次执行git status

git status

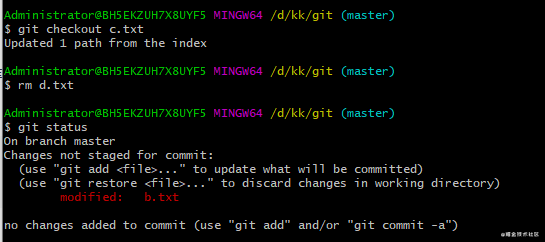


我们可以看到， c.txt之前存在版本库中，我们需要把修改丢弃。 d.txt在版本库中没有，所以我们需要删除文件。 使用如下命令：

git checkout c.txt（把c.txt修改的丢弃）

rm d.txt

rm -f 文件名 // 强制删进程



总结：

* 对于版本库中存在的文件使用 **git checkout <filename>** 命令放弃修改，全部文件放弃修改使用**git checkout .**
* 对于版本库中不存在的文件使用 **rm <filename>** 命令删除文件

**2.5.2 暂存区回退**

当文件已提交到暂存区，我们应该如何回退呢？我们先来修改一些文件，并且提交到暂存区

修改a.txt文件

新增b.txt文件

git add .

git status

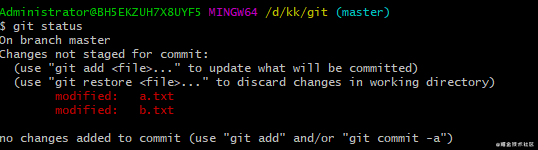


使用如下命令移除暂存区，让文件回到工作区，然后使用上一节的方法丢弃修改

git reset HEAD 文件名 //对当前文件进行revert

git reset HEAD . //对当前路径下所有文件全部revert.

git status //查询当前路径下文件的状态



总结：

* **使用 git reset HEAD <filename> 命令使文件移除暂存区，回到工作区**
* **使用git reset HEAD . //对当前路径下所有文件全部revert**

**2.5.2 资源库版本切换**

我们先查看一下历史：

**git log**

我们查到了版本对应的git的id,接下来使用下面命令回退到**之前提交的'项目初始化'这一版本**，执行完之后在git status 发现我们现在目录内容回到最初的样子了[*版本号可以只列出前10位*]。

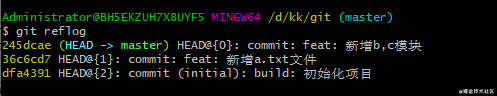
git reset --hard dfa4391707f9a64baa8eecf94e07e01a959fa091

git status



**注意**：如果突然我们发现自己回退错版本了，需要回退到 "新增a.txt文件模块"的那个版本，我们可以使用下面命令找到所以变更的版本历史：

**git reflog**



PS：我们经常看到 git reset --hard HEAD^ 这样的命令， 这句话表示放弃修改的内容，回退当前的版本，HEAD 表示当前版本，HEAD^ 表示上一个版本, HEAD~100表示上100个版本

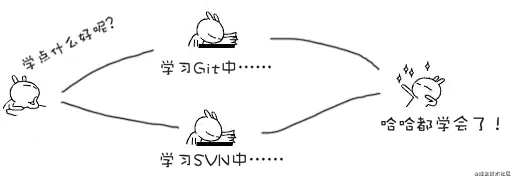
总结：

* **使用git reset --hard <版本号> 进行版本切换**
* **使用git reflog 查看所有的版本变更历史**
* **HEAD 表示当前版本**

**3 分支**

**3.1 分支介绍**

分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。 如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN！



分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

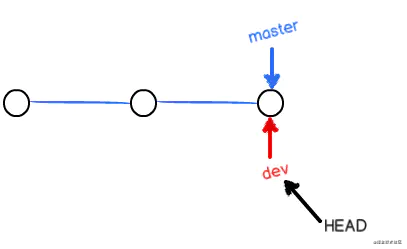
现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

**3.2 分支的创建、合并以及删除**

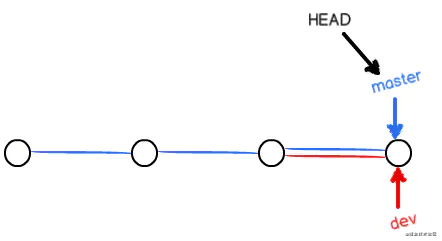
当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

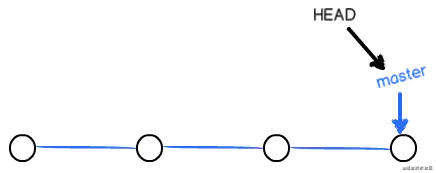
不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：

假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



下面开始实战：

1）创建分支

**git checkout -b dev**

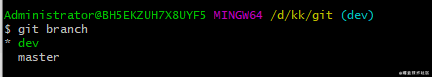
git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

**git branch dev （创建dev分支）**

**git checkout dev （切换到dev分支）**

2） 查看分支

git branch



3）修改文件并提交

修改 a.txt 文件里的内容

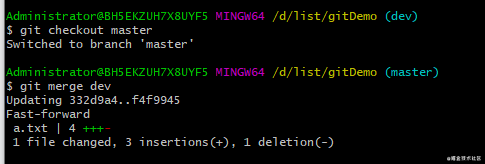
git add .

git commit -m "feat: a模块新增一行"

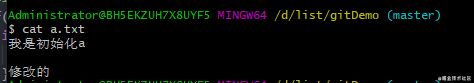
4）合并dev到master

git checkout master // 切回master分支

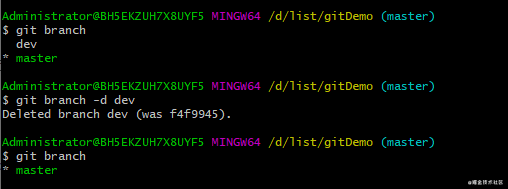
git merge dev // 合并dev分支到master



执行cat a.txt 就可以看到a.txt文件里面的内容了



4）删除dev分支，执行 **git branch -d dev**



总结

**1）Git鼓励大量使用分支**：

**查看分支**：**git branch**

*Note: 列出所有分支名，且当前分支以绿色字体显示*

**创建分支：git branch <name>**

**切换分支：git checkout <name>或者git switch <name>**

*Note: 建议使用switch，因为checkout可以撤回修改，容易混淆*

**创建+切换分支**：**git checkout -b <name>**或者**git switch -c <name>**

**合并某分支到当前分支：git merge <name>**

**删除分支：git branch -d <name>**

**// -d: 分支内容来merge到主干**

**// -D: 强制删除**

2）有文件存在**工作区或者暂存区**时，切换分支后，这些文件会原封不动的带到新的分支。

**3.3 解决冲突**

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

我们先修改一下master分支的a.txt文件

git add .

git commit -m "feat: 新增一行master"

我们再切到qa分支上，修改a.txt文件，注意：出现冲突的原因是要修改同一位置的内容

git checkout -b qa

git add .

git commit -m "feat: 新增一行qa"

然后我们再master分支合并一下

git checkout master

git merge qa

git status

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容。 ======= 上面的内容为当前分支的内容，下面的为合并过来分支的内容我们做相应的修改，然后提交.

git add .

git commit -m "chore: 解决qa合并到master的冲突"

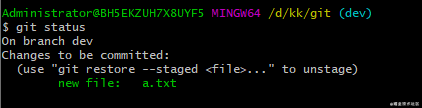
小结：

* 当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。
* 解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

**3.4 bug分支**

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

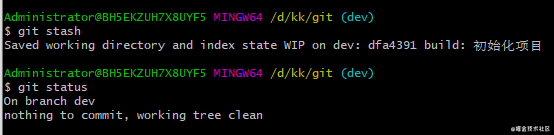
当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支bug-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：



我们可以使用 **stash**命令把修改储藏起来

**git stash**

**git status**



我们再创建bug-101分支进行修改

git checkout -b bug-101

新建b.tex,并进行修改一下

git add .

git commit -m "fix: 修正bug101"

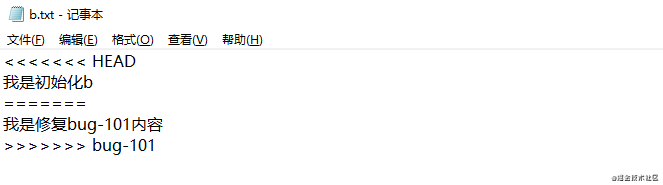
切回master分支,并且合并

git checkout master

git merge --no-ff -m "chore: bug101合并到master上" bug-101

取出储藏的修改

**git stash pop**



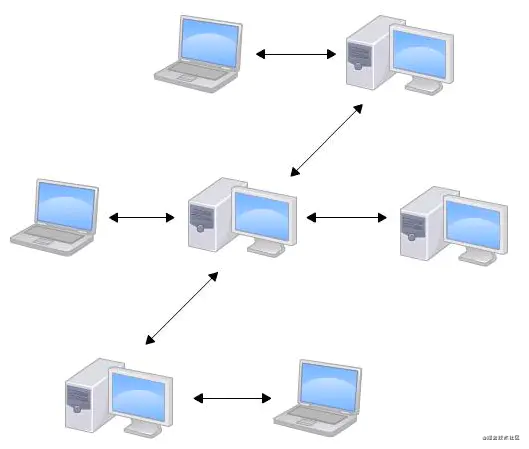
我们发现b.txt文件冲突了，根据上面的内容解决冲突就可以了。

总结：  
修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；  
当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。  
一般我们都是切独立分支修改的，所以这个方法我们用的不是特别多。

**4 远程仓库的使用**

上面的章节里面，我们介绍了git的本地使用，知道了在本地各版本之间进行穿梭。 但这仅限于我们自己玩，如果要大家一起工作完成一个项目，势必需要远程仓库。

虽然git是分布式的，大家可以相互传输文件，但其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。而为了方便管理，我们一般会把托管平台选为“中央服务器”。

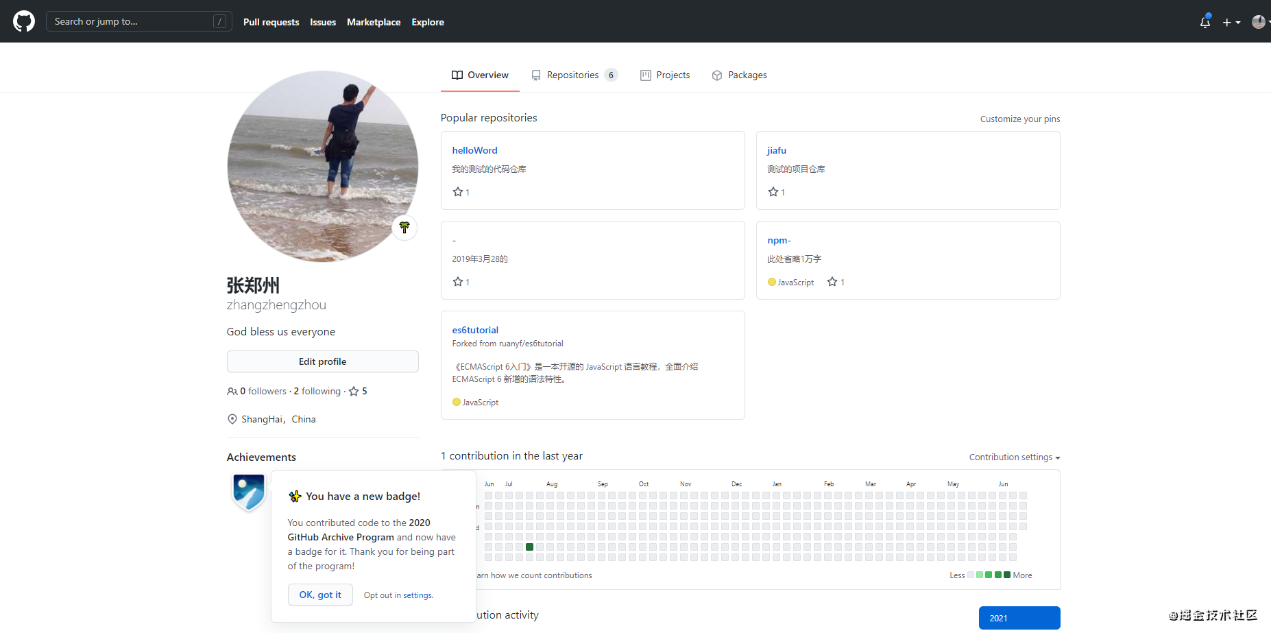


**4.1 托管平台（github拓展）**

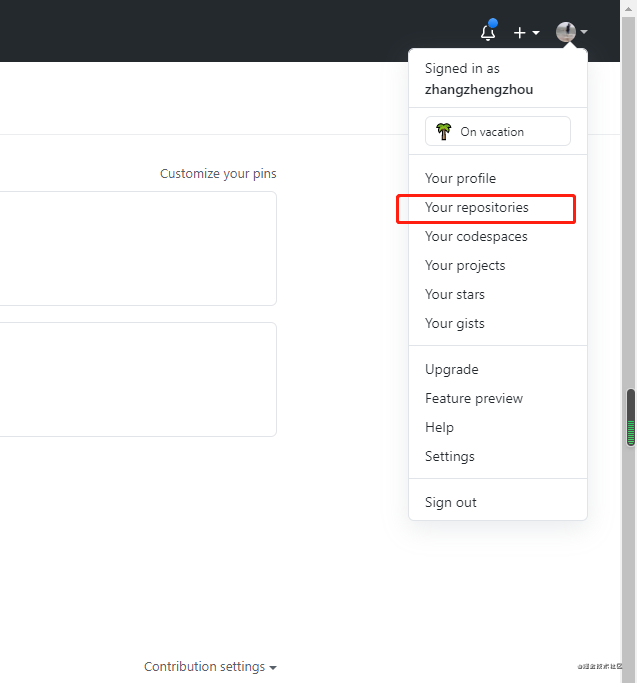
托管平台可以是第三方提供的，如github,码云等，也可以是我们自己搭建的（公司使用gogs自己搭建了一个）。各种托管平台操作大同小异，下面我就用第三方提供的github来当作我们的“中央处理器”。

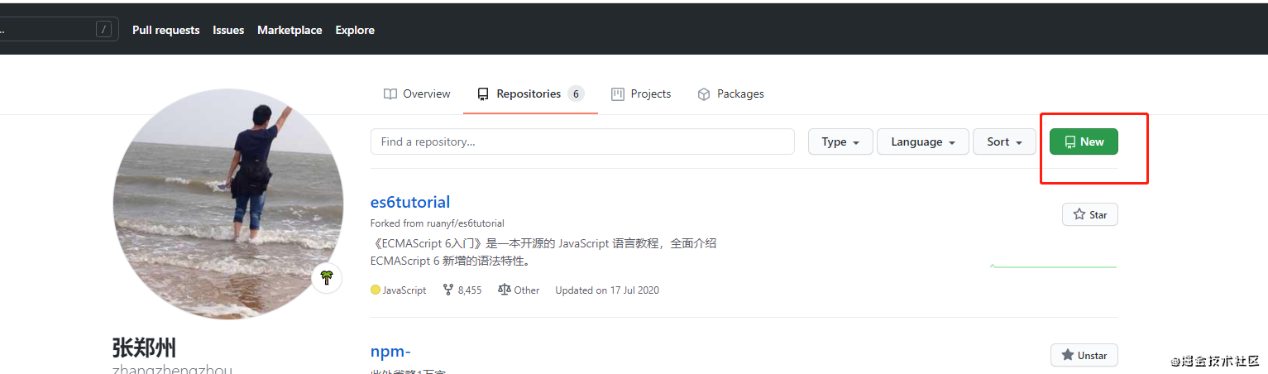
* github 网址：github.com/  
  接下来我们在github上新建一个项目：

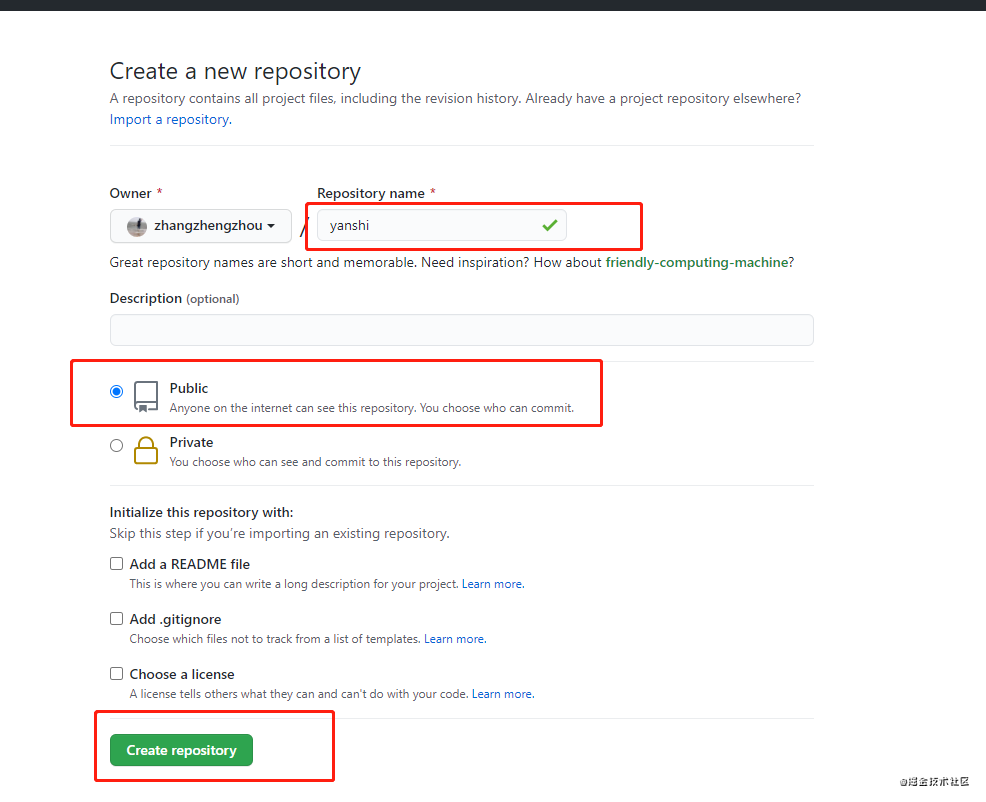
1） 注册登陆github



2） 新建仓库

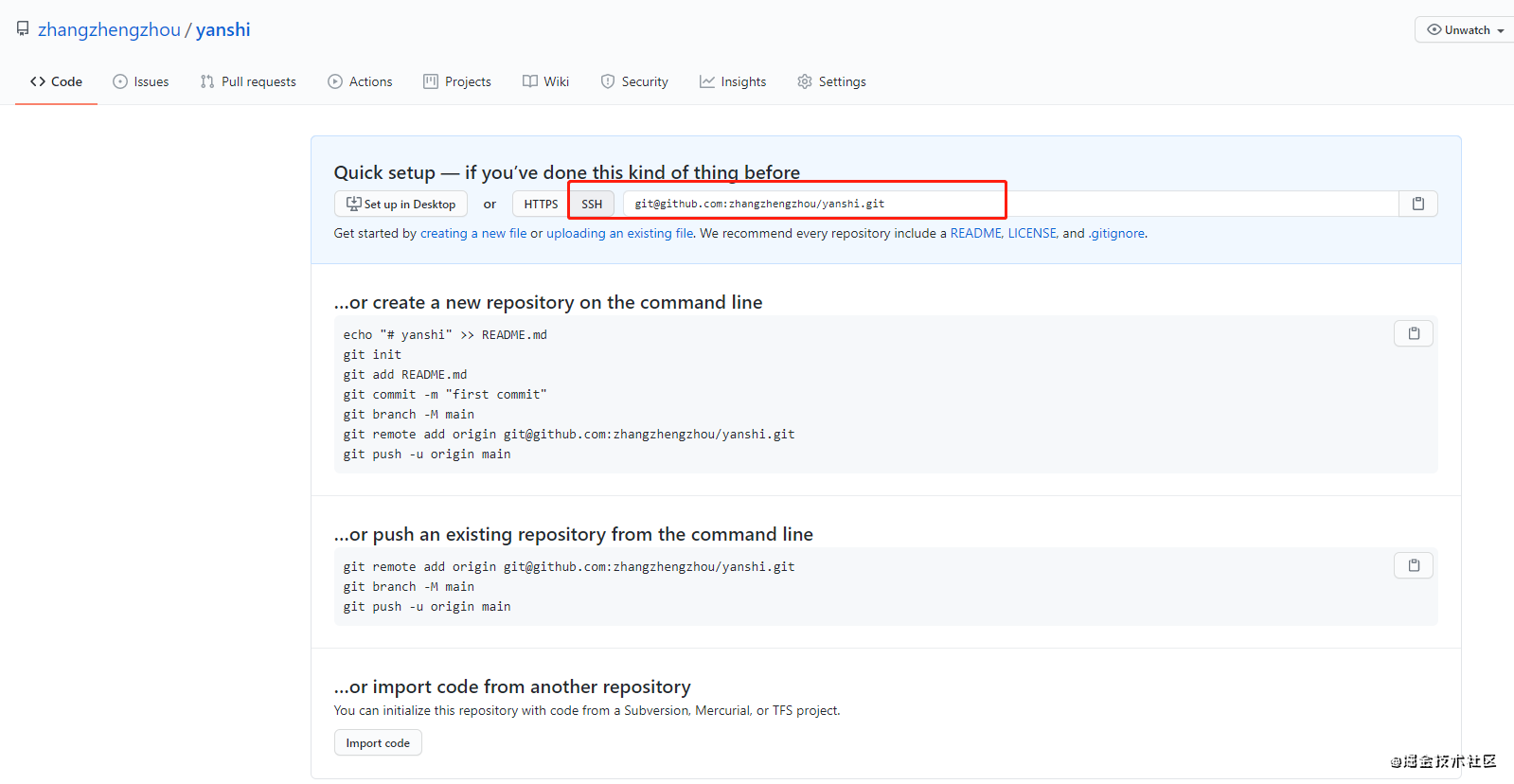






**4.1.1 本地仓库与远程仓库关联**

在仓库页面，我们有两种关联仓库的方式，我们使用SSH的方式来关联。

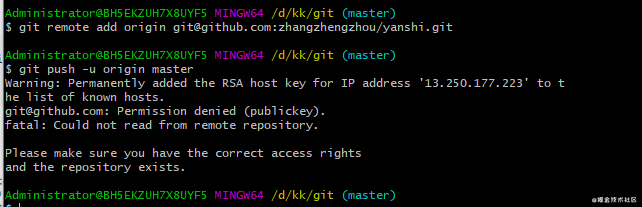


执行下面命令：

git remote add origin git@github.com:zhangzhengzhou/yanshi.git

git push -u origin main（第一次提交一定要带上-u）

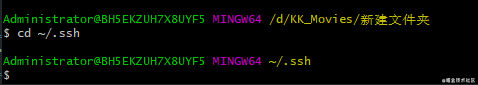
执行完成后会报一个权限验证不通过的错误。



要消除这个错误，需要把我们服务器的SSH公钥放到github上

**4.1.2 生成ssh公钥**

我们使用下面命令生成公钥和私钥。  
1.验证下本地有没有生成.ssh文件，如果没提示说明本地有了 反之要去生成.ssh

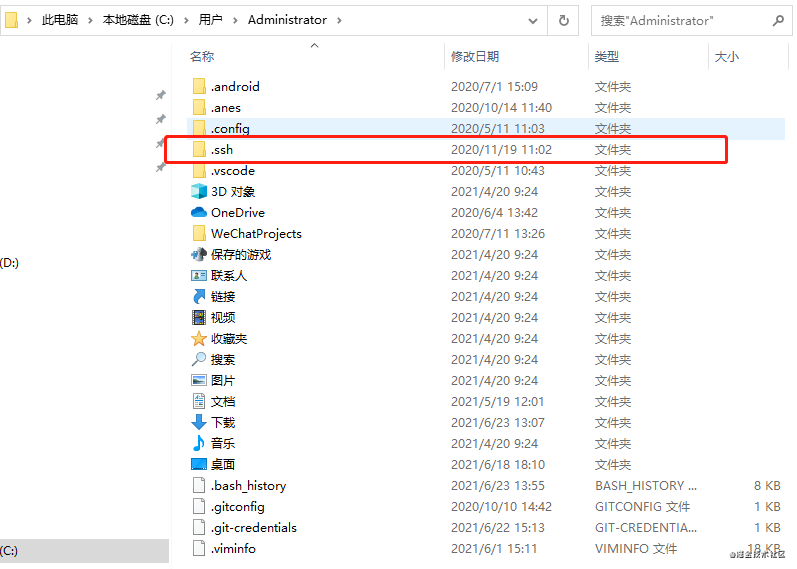


2.本地没有ssh 执行以下命令

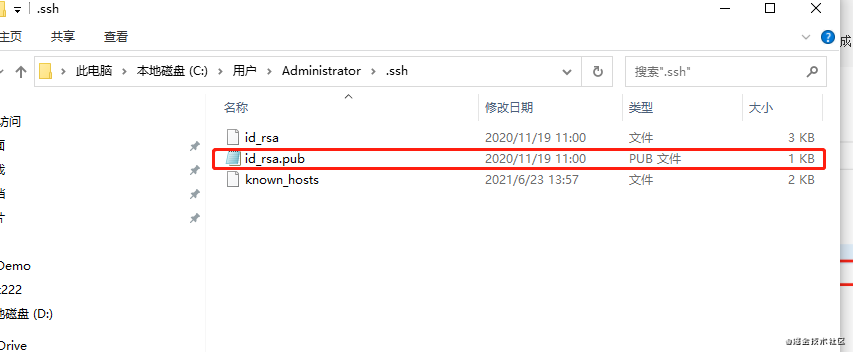
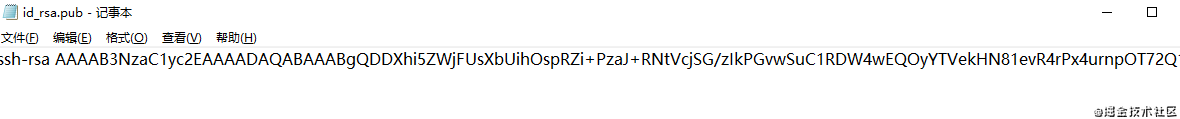
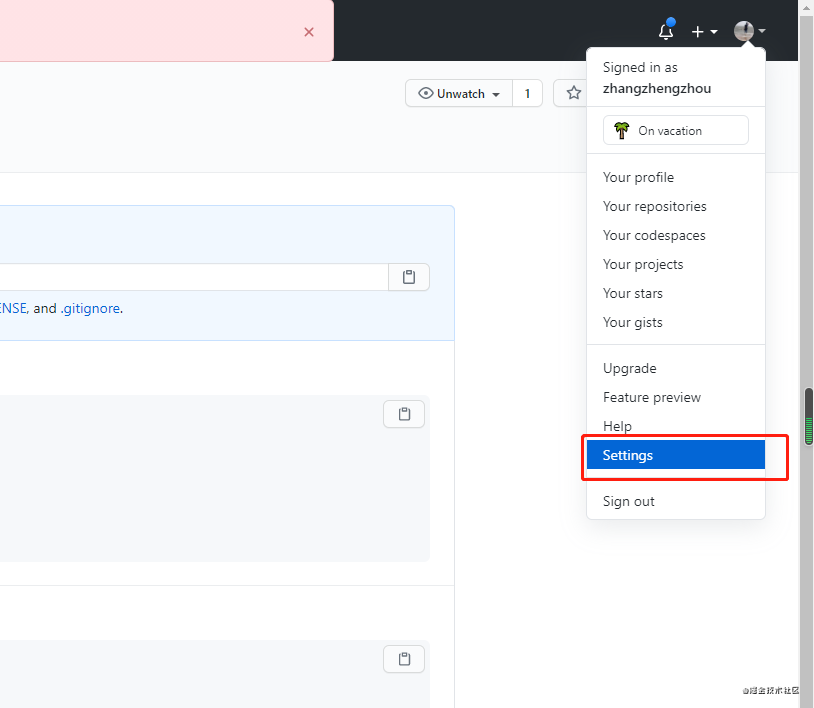
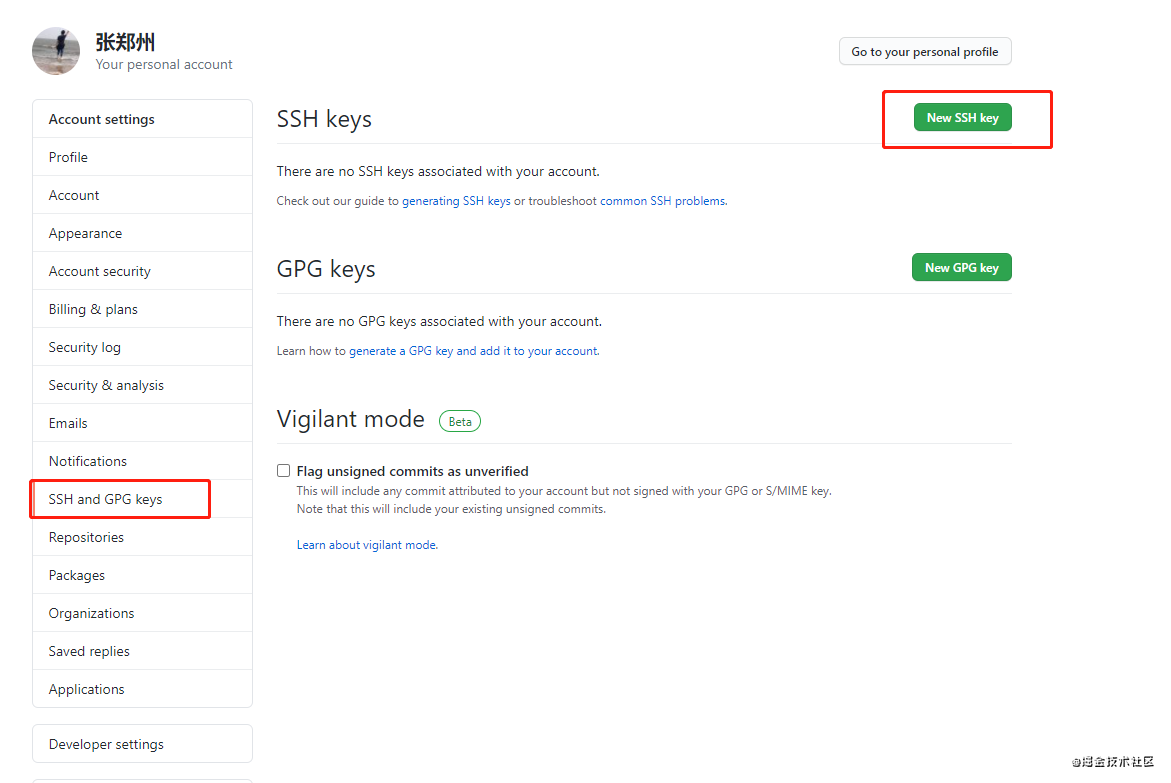
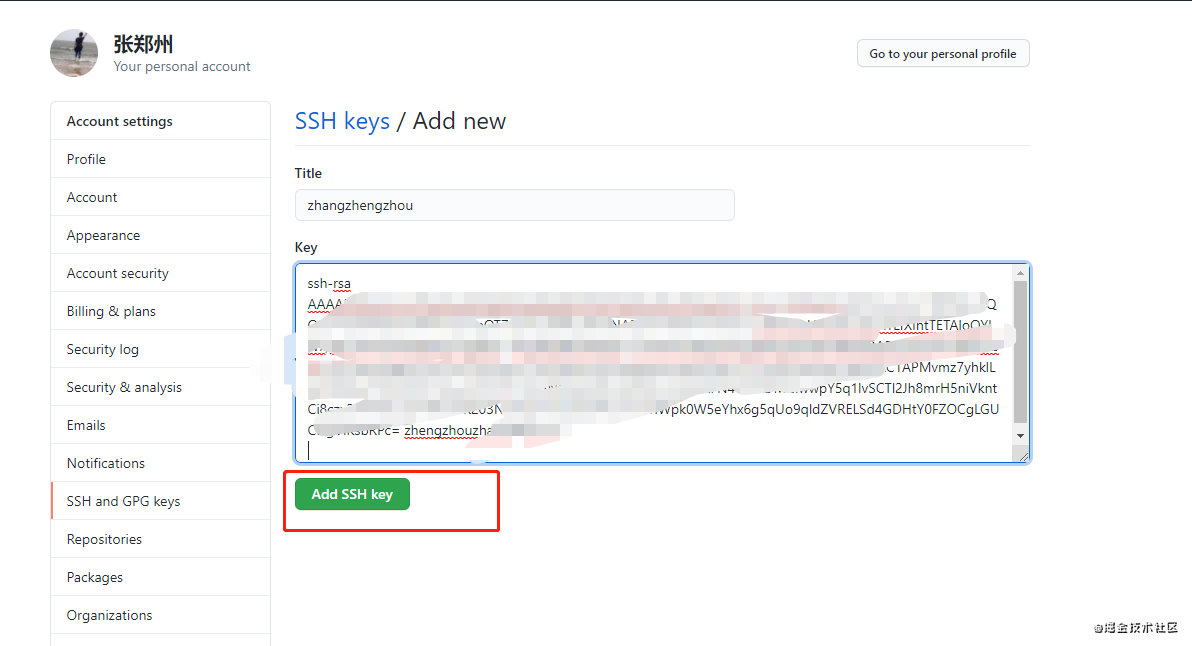
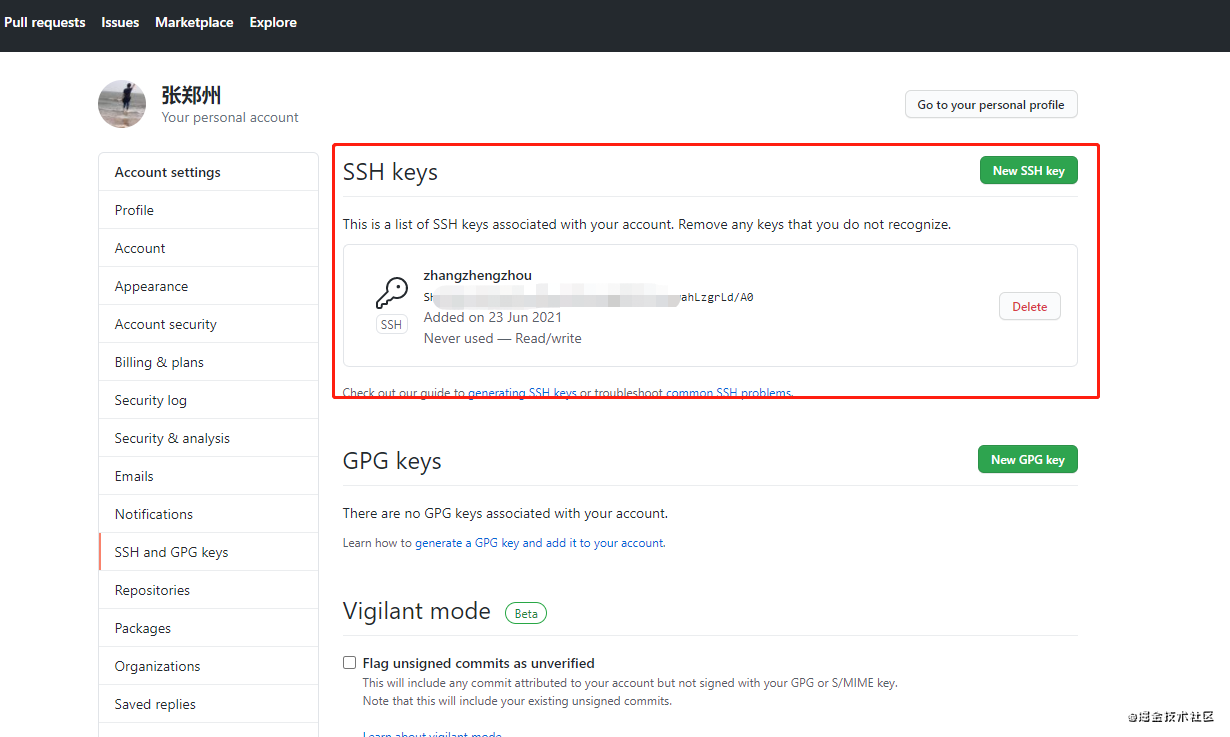
ssh-keygen -t rsa 或者 ssh-keygen -t rsa -C "your@email.com"

之后执行 cd ~/.ssh 去查看有没有提示 没提示说明已生成

一般生成的路径如下

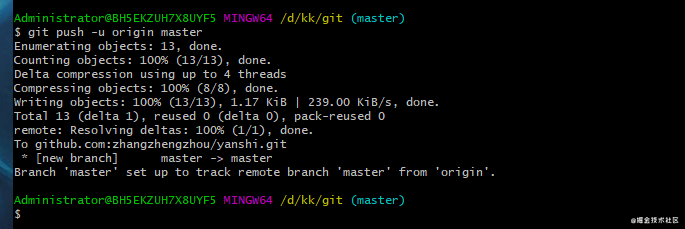


把公钥的内容复制到gitthub上。

1. 
2. 
3. 
4. 
5. 
6. 

我们再执行一次,就可以把master 分支推到远程仓库中去了。

git push -u origin master

* u :代表关联的主机
* origin：代表远程仓库源  
  

**小结：**

* 我们使用 git remote add origin <远程地址> 关联远程仓库
* 第一次使用 git push -u origin master 推送
* 使用SSH协议，需要配置SSHkey
* 以后再次推送直接使用 git push 即可

**4.1.3 推送远程分支和删除远程分支**

我们把qa分支推送到远程仓库：

git checkout qa

git push origin qa //第一次使用和远程qa分支想关联

使用下面命令来删除远程分支：

git push origin --delete qa

**4.1.4 从远程仓库克隆**

如果我们本地没有仓库，需要把线上的仓库克隆下来，可以使用下面的命令,这里我们使用http的方式

git clone https://github.com/.git -b qa yanshi

我们来解释一下里面的两个参数

* -b qa: 表示克隆qa分支，如果不加，默认拷贝master分支
* yanshi: 把克隆下来的内容放在"yanshi"文件夹下面  
  小结：
* 使用 git clone git@github.com:zhangzhengzhou/yanshi.git [-b 分支名] [目录名] 来克隆仓库
* 我们不建议使用htts的方式，因为https除了慢，还要经常输入口令(即github或者gitlab的登录账号和密码)非常麻烦

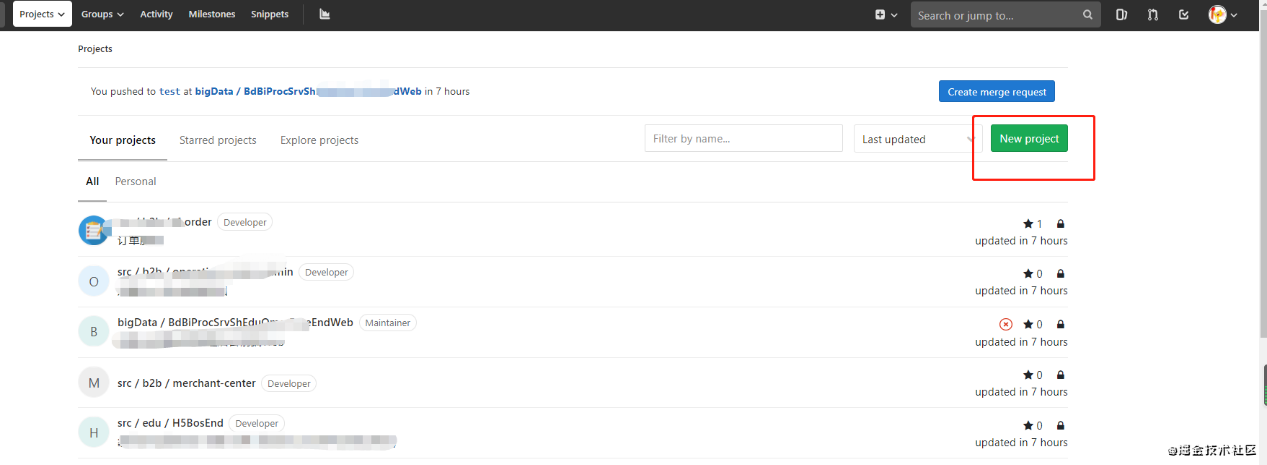
**思考题**

如果我们远程分支推错了怎么办？

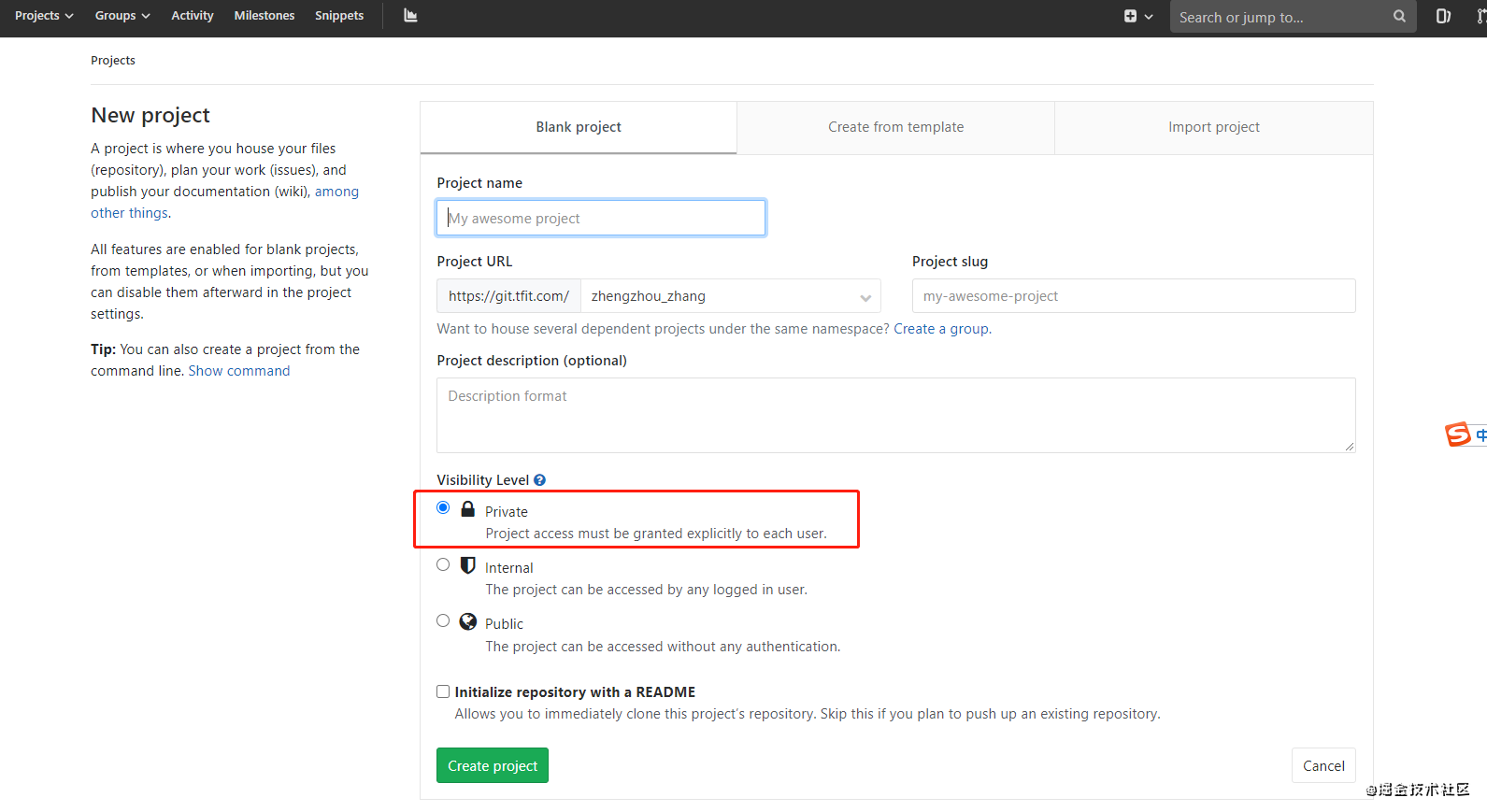
**4.2托管平台（gitlab 主讲）**

**4.2.1 创建项目**

1)创建项目

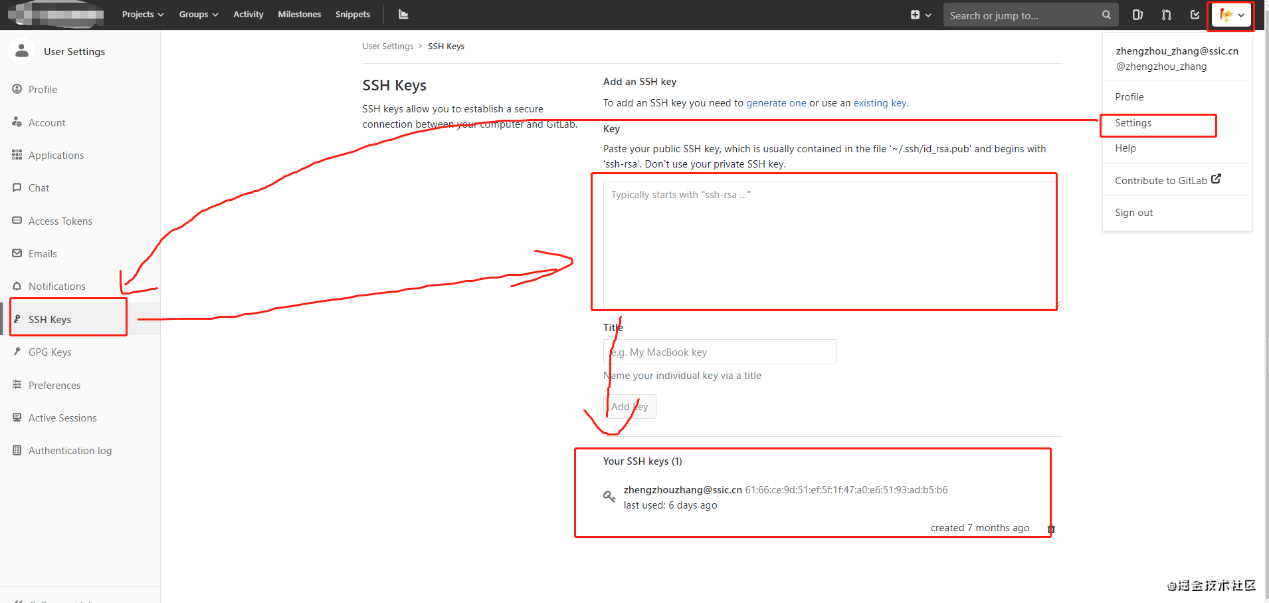
  
**选择Private项**

* Private：项目访问权必须明确授予每个用户
* Internal：任何登录的用户都可以访问该项目
* Public: 无需任何身份验证即可访问该项目



**4.2.2 本地仓库与远程仓库关联同目录4.1.1**

**4.2.3 同生成ssh公钥目录 4.1.2**



4）同4.1.4 从远程仓库克隆

**5 Pull Request 的命令行管理**

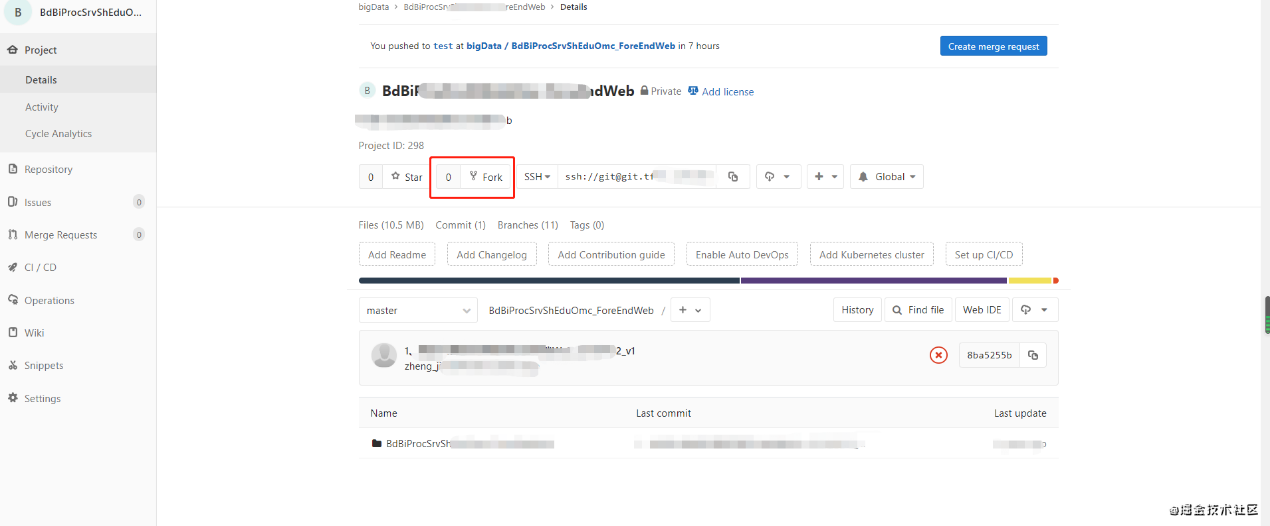
**5.1 Pull Request介绍**

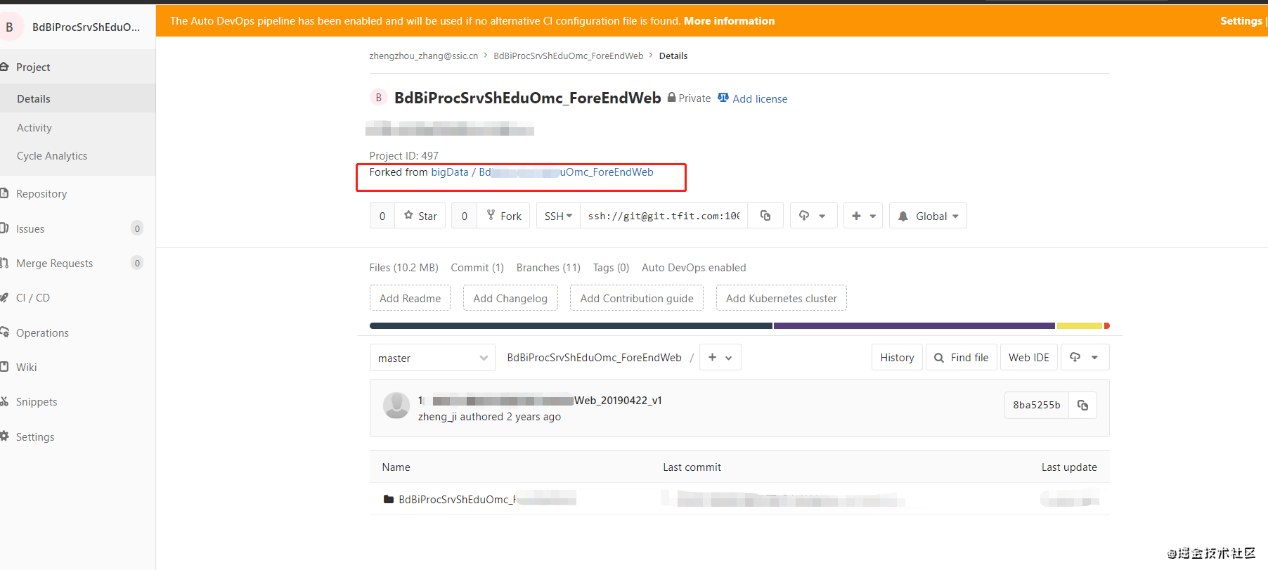
"Pull Request 是一种通知机制。你修改了他人的代码，将你的修改通知原来的作者，希望他合并你的修改，这就是 Pull Request。

**5.2 Pull Request 的流程**

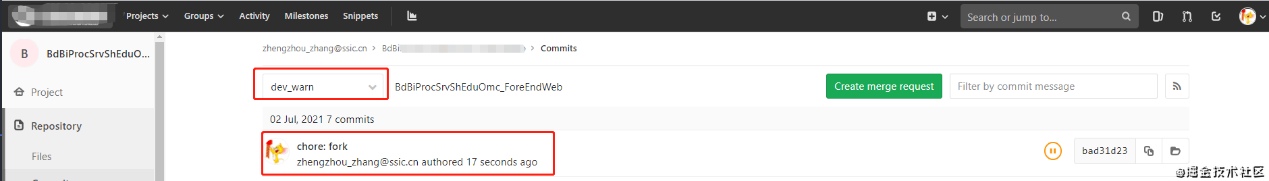
第一步，你需要把别人的代码，克隆clone到你自己的仓库，Github 的术语叫做 fork。

比如定义别人的仓库定义为A，克隆到你自己的仓库B

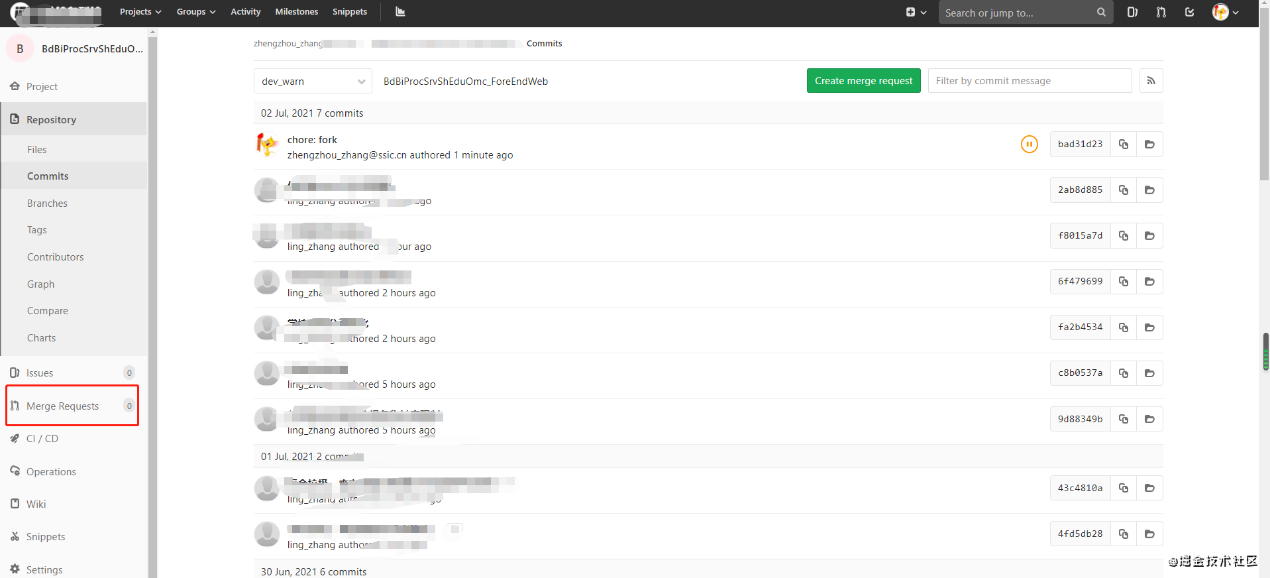


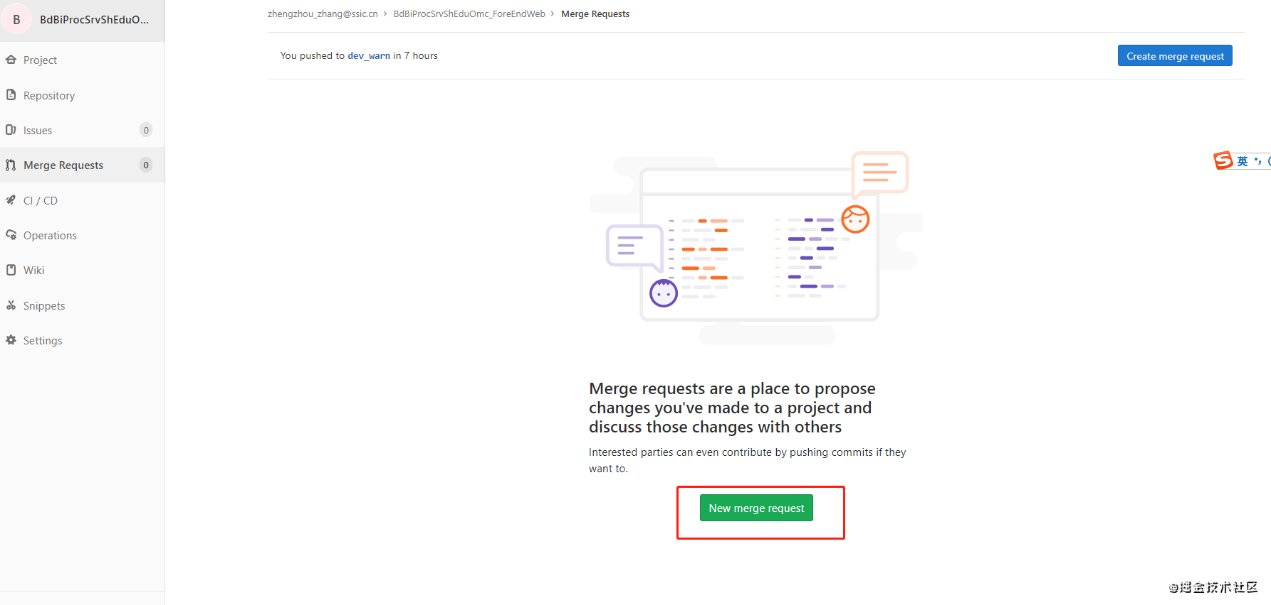


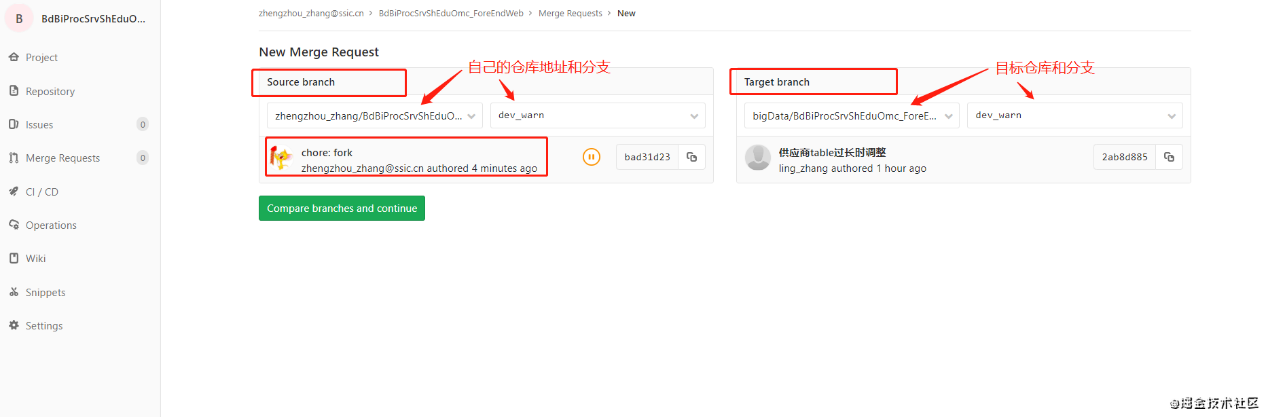
接着我们要在自己的仓库里面clone到本地，进行一些列的add、commit、push修改提交



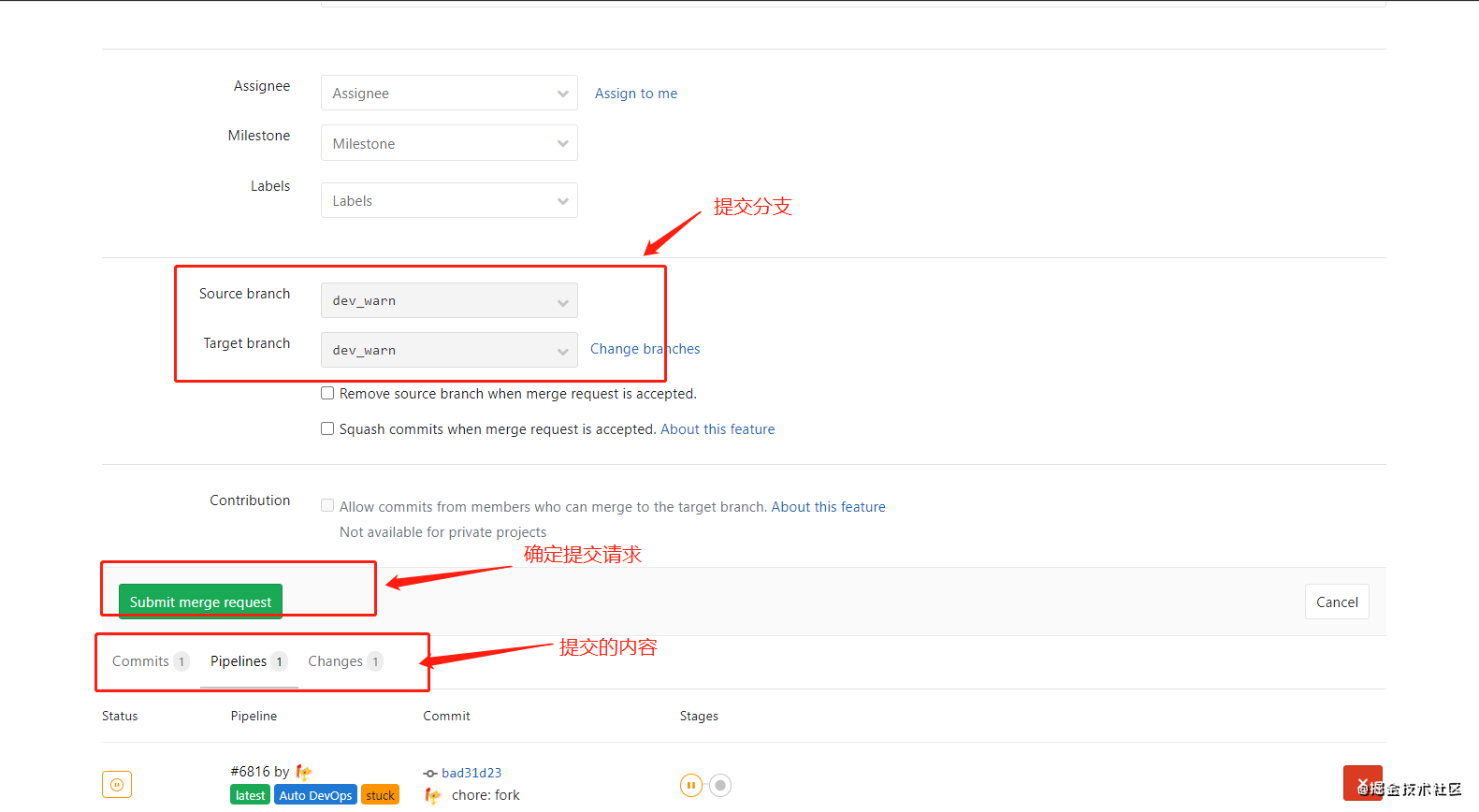
然后进行merge requests





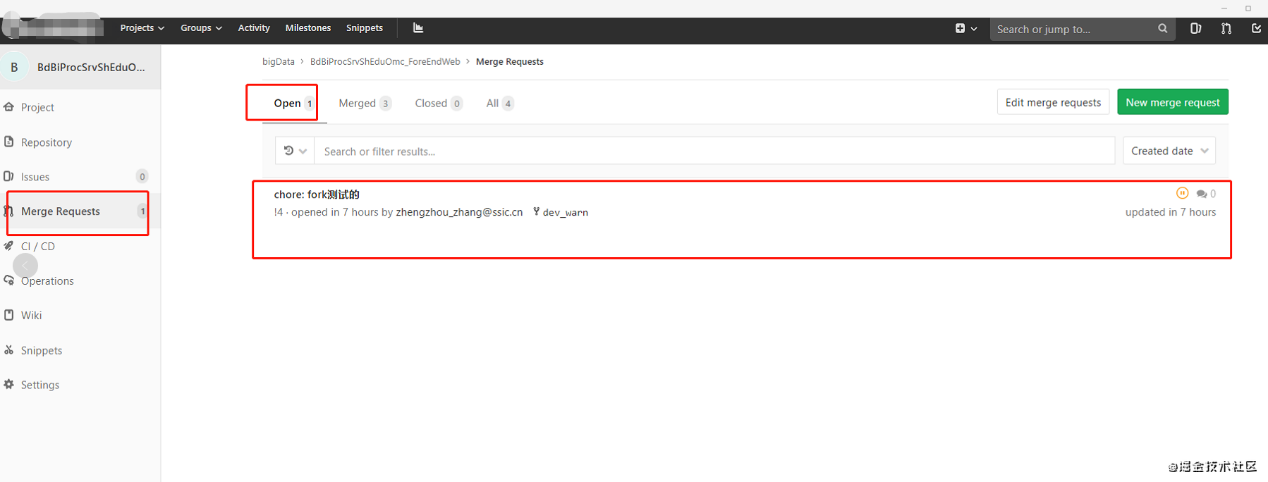


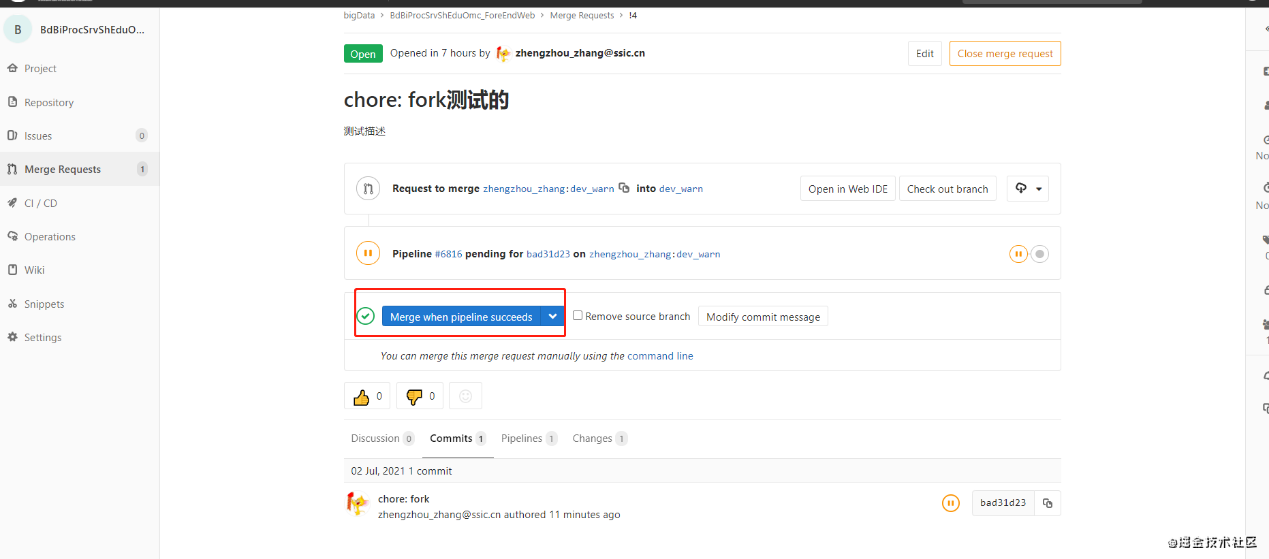
确认提交信息

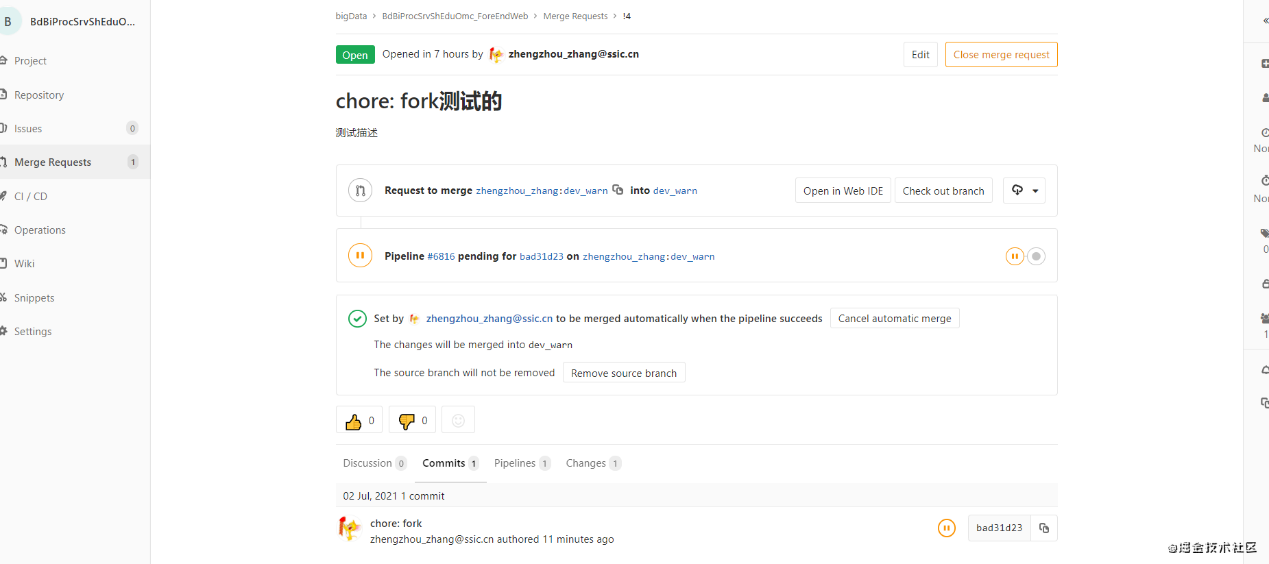


接下来我们就可以在A 也就是别人的仓库里面就可以收到merge requests 的请求记录列表，

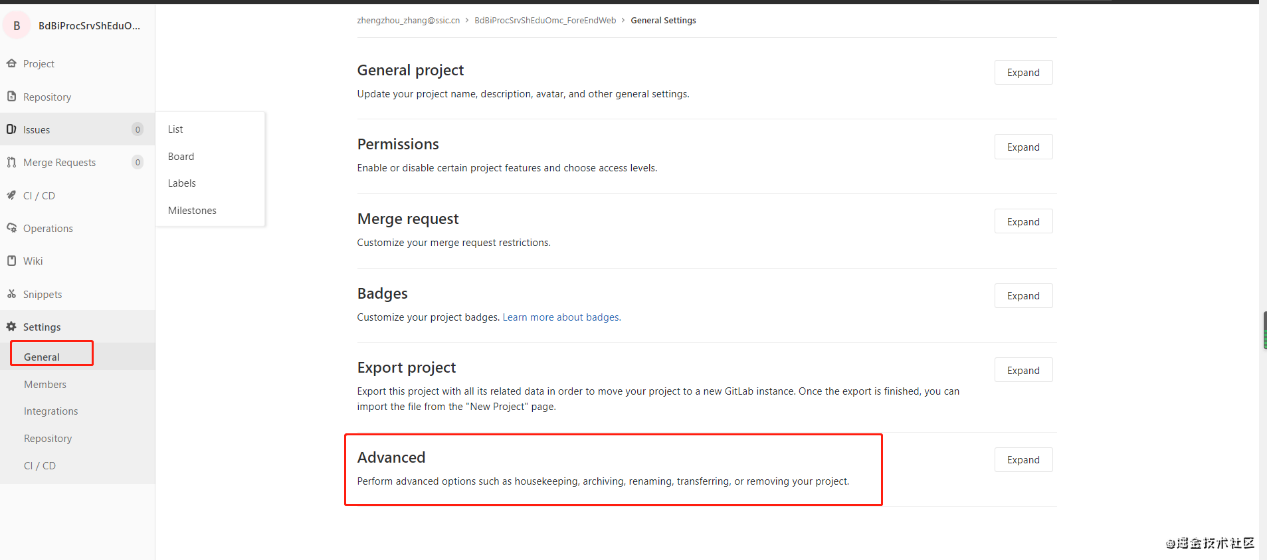
查看是否进merge和并



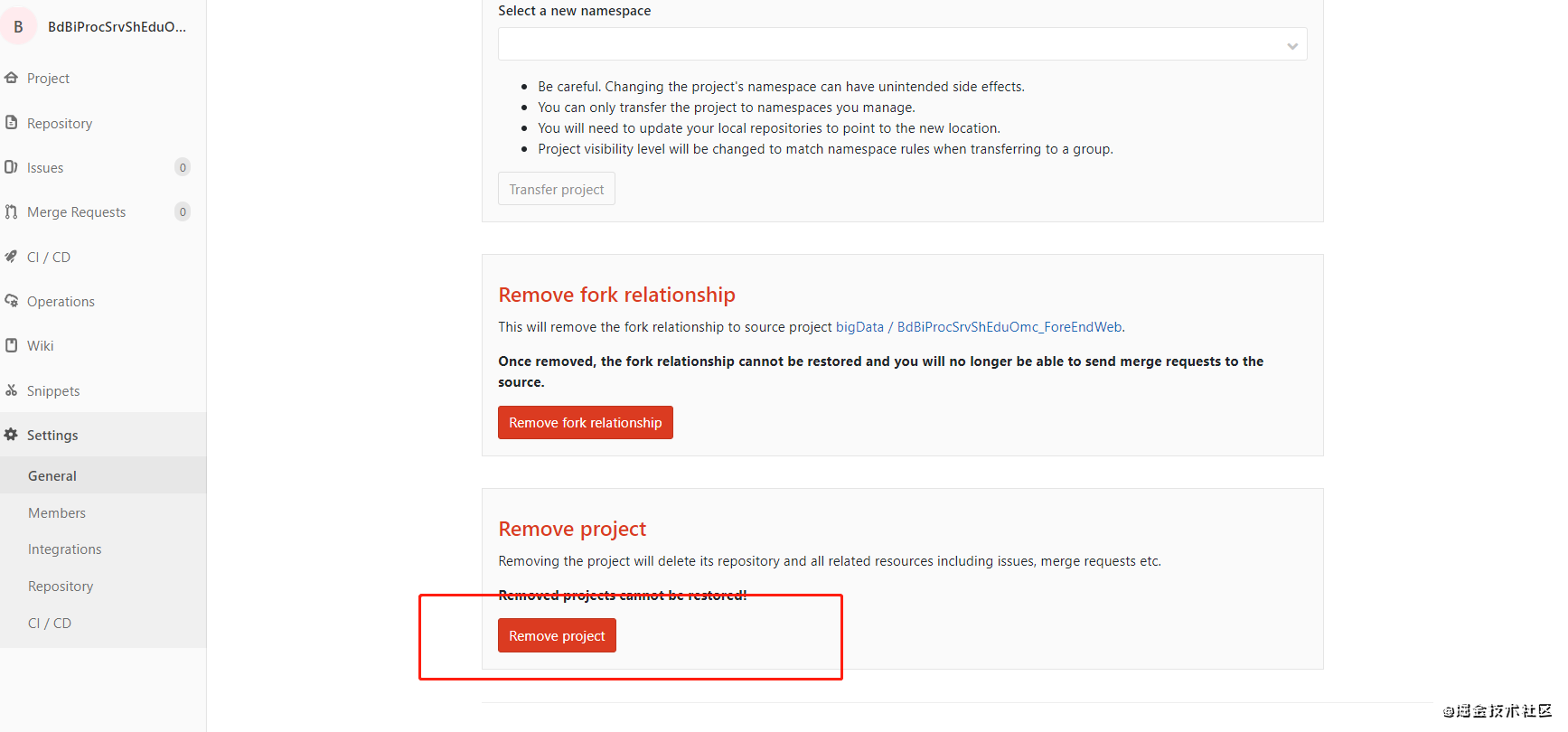


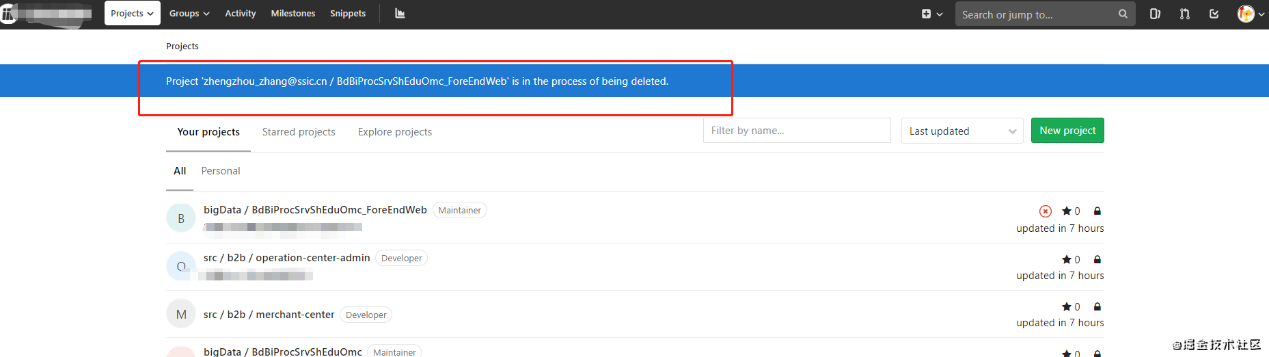


**5.3 删除自己仓库fork的项目**



点击删除的弹框里面 一定要输入项目名称例如BdBiProcSrvShEduOmc\_ForeEndWeb 不然不能confirm确认

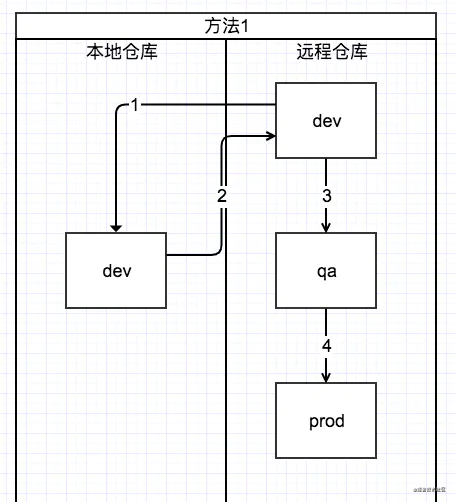




**6. 操作git的两种工作流程**

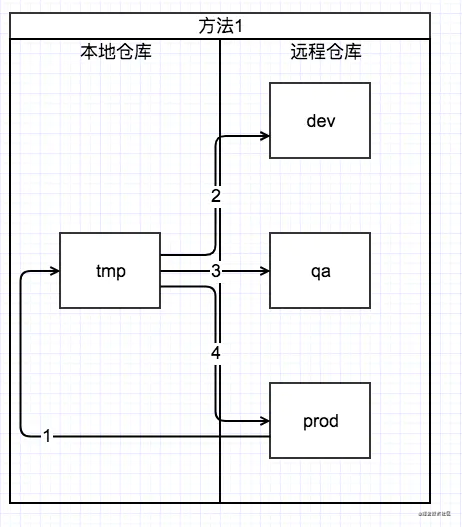
**1）第一种 把远程的dev分支拷贝到本地**

1. 把远程的dev分支拷贝到本地
2. 修改dev分支，并push给远程仓库
3. 远程仓库把dev分支合并到qa
4. 远程仓库把qa合并到prod



**2）第二种 从远程prod分支check出一个tmp分支**

1. 从远程prod分支check出一个tmp分支
2. 修改tmp分支，把tmp分支合并到远程仓库的dev
3. 把tmp分支合并到远程仓库的qa
4. 把tmp分支合并到远程仓库的prod



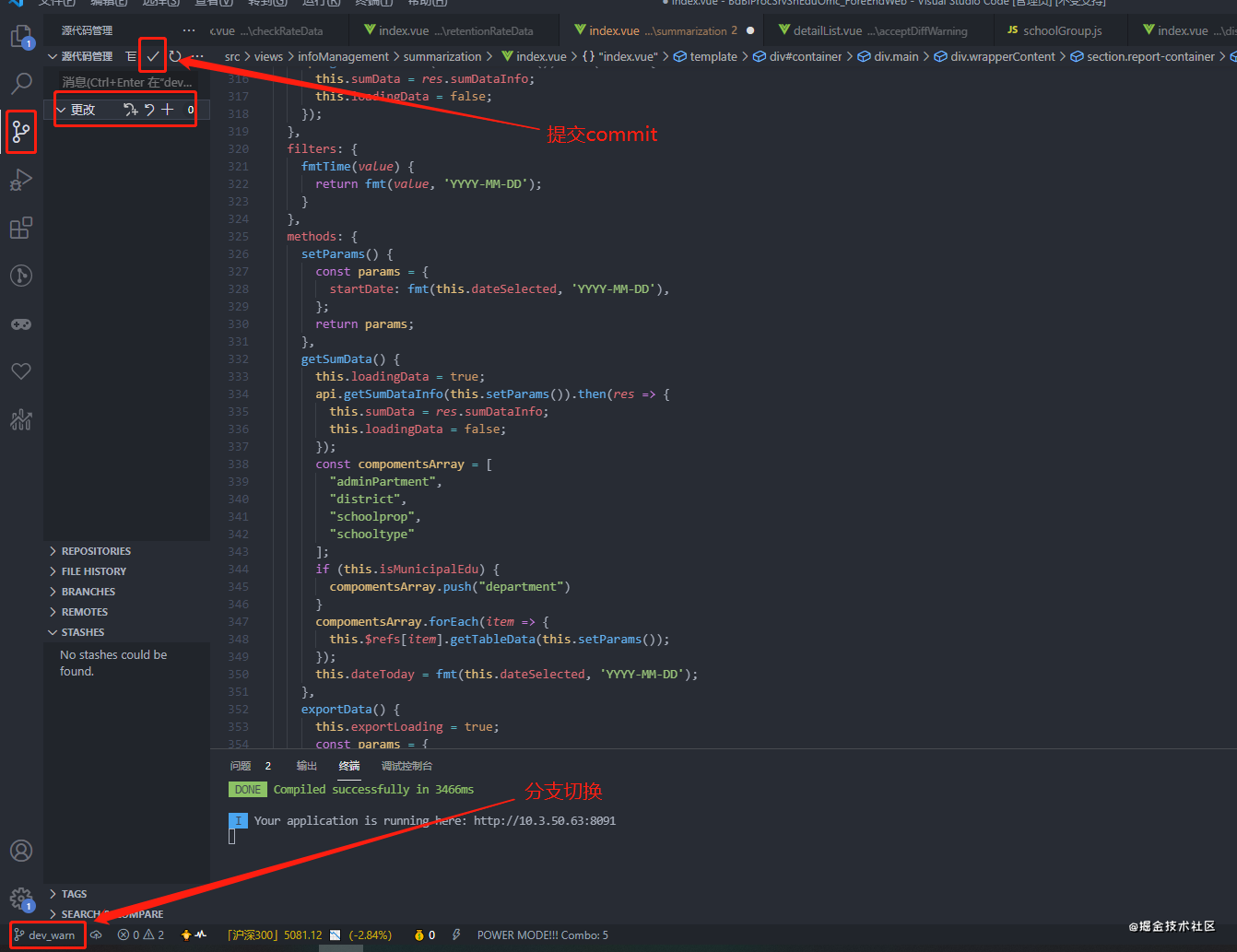
**思考题：**

这两种方式孰优孰劣？我们应该如何选择？

**7 拓展： vscode的安装和使用git**

地址：https://code.visualstudio.com/

vscode有内置的git插件，所以无需安装，当然你也可以使用其他git插件辅助（比如SourceTree客户端）



SourceTree客户端



