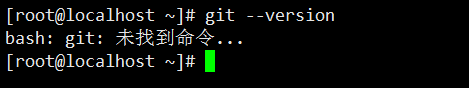
# git教程

本次学习使用centos7系统平台

## 查看是否系统中存在git



## 安装git

yum install git

## 登录git

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

git config --global user.name "renkaige"

git config --global user.email "renkaige6735@qq.com"

注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

## 创建版本库

什么是版本库呢？版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

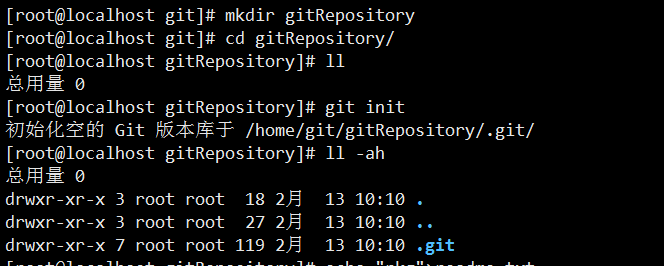
所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

git init

瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

如果你没有看到.git目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看见。

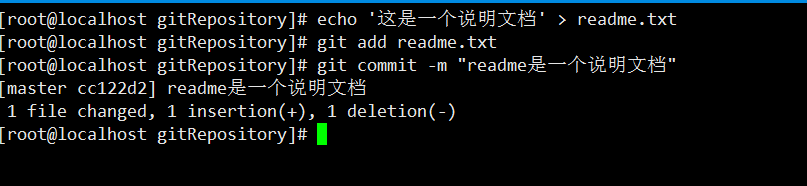


## 上传文件

在git本地仓库中创建文件并提交

git add readme.txt ##提交文件

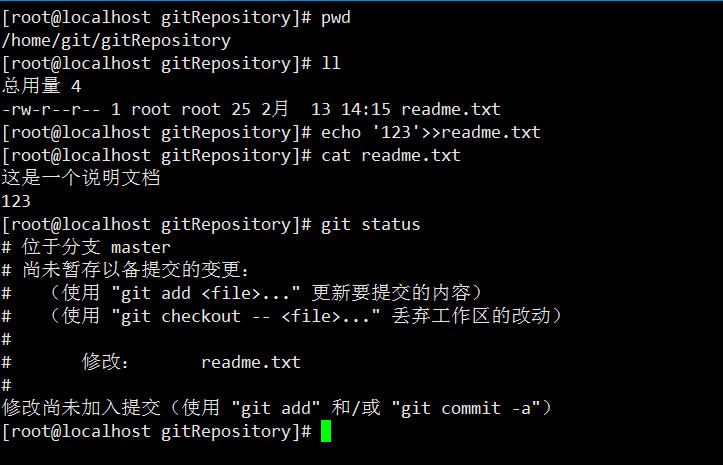
git commit –m “本次提交的注释说明,相当于svn中的注释”



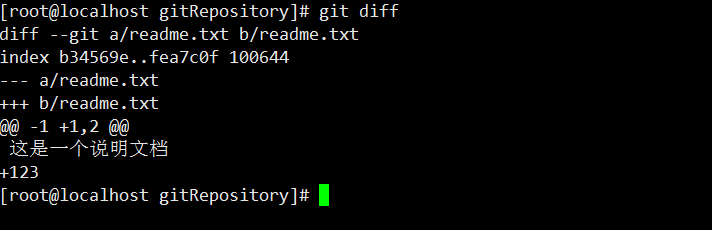
## 查看工作区中的文件变化

为readme.txt追加123

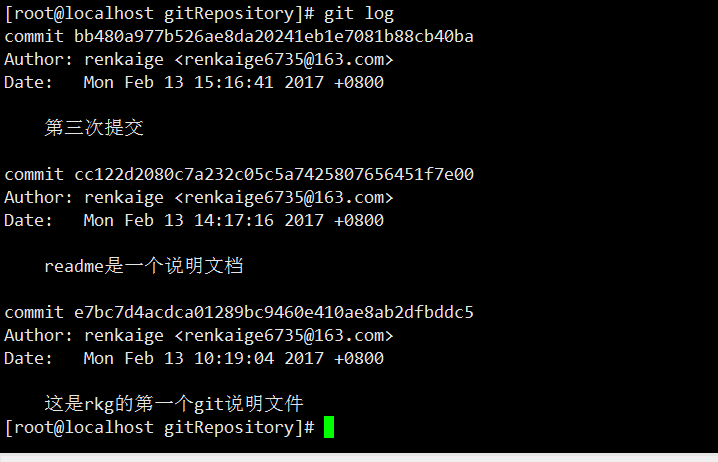
使用git status命令可以发现提示readme.txt修改但是没有提交



使用git diff查看当前仓库中文件更改了哪些内容,可以发现+123是新增的内容删除内容会在内容前面加一个减号

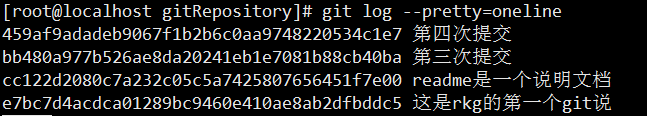


## 查看git历史记录



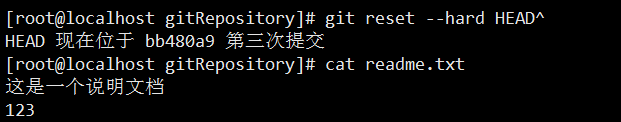
## 将文件会文件夹回退到上个版本(任意版本)

可以使用git log --pretty=oneline命令查看缩略版的git日志



在Git中，用HEAD表示当前版本，也就是最新的提交(第四次提交)，上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

### 回退到上一个版本(第三次提交) git reset --hard HEAD^



### 再次回退到上个版本(readme是一个说明文档) git reset --hard HEAD~1



### 回退到当前版本git reset --hard HEAD或者git reset --hard HEAD~0





### 回退到后面的版本(最新版)

假设现在处于第一个版本,那么该如何回到最新版本呢

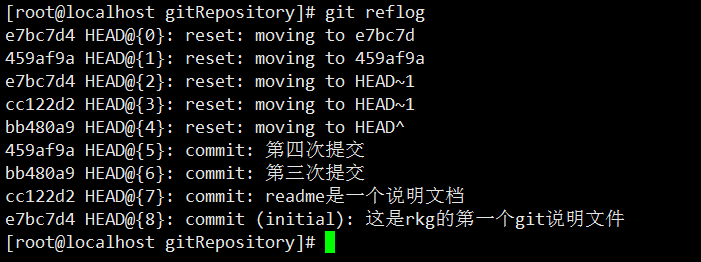
git为我们提供了回到指定版本的命令

git reset --hard 版本号(前面git log命令中显示的版本号)

但是通过git log命令可以发现在处于第一个版本时git的历史中没有后面的历史记录(交心的版本)



Git提供了一个命令git reflog用来记录你的每一次命令



这样就可以使用前面的git reset --hard 版本号(版本号可以不用写全,但是也不能只写两三位,开头两三位可能重复git不知道查找哪一个版本)

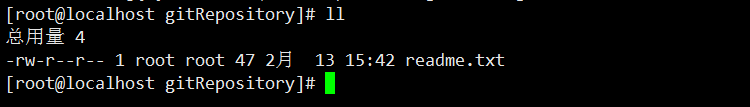
### 注意

HEAD~后面的数字必须是0或正整数,且不能超过当前版本的历史版本(假设总共有4个版本当前处于第二个版本中,所以git历史中当前版本前面只有一个版本,既HEAD~不能大于1)

## 暂存区与工作区

### 工作区

就是你在电脑里能看到的目录，比如下面的GitRepository文件夹就是一个工作区：



### 版本库（Repository）&暂存区(stage)

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。

s

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

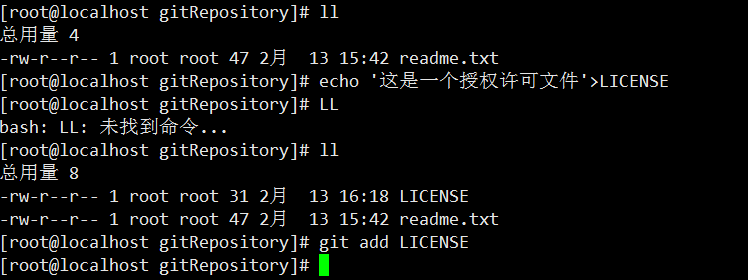
因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

俗话说，实践出真知。现在，我们再练习一遍，先对readme.txt做个修改，比如加上一行内容：

vim readme.txt

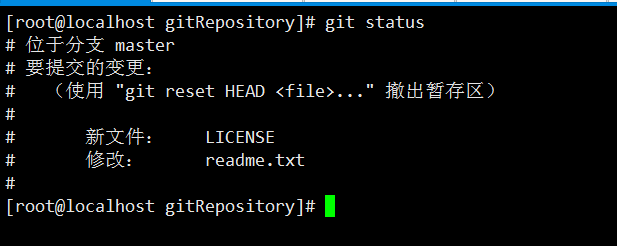
然后，在工作区新增一个LICENSE文本文件（内容随便写）。



先用git status查看一下状态：Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过,未跟踪的文件



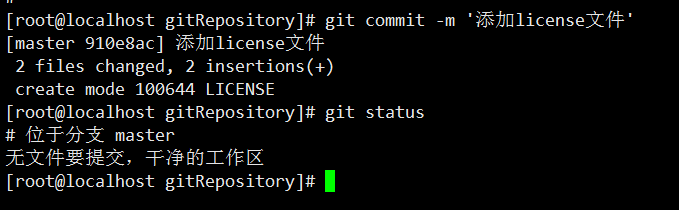
现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：



现在，暂存区的状态就变成这样了：可以发现暂存区中新增了LICENSE文件修改了readme.txt



所以，git add命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行git commit就可以一次性把暂存区的所有修改提交到分支, 一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：现在版本库变成了这样，暂存区就没有任何内容了：

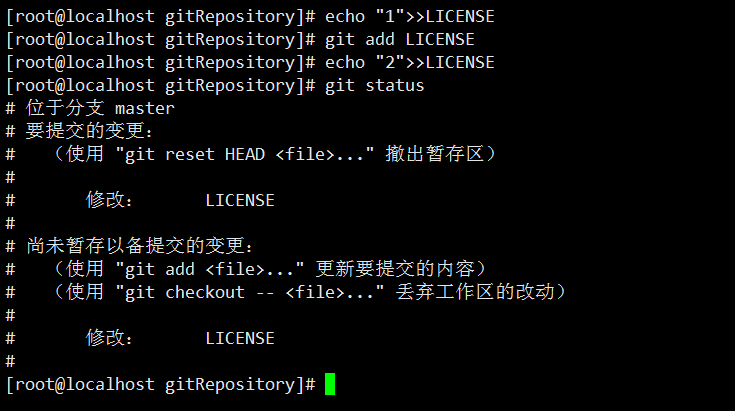


## git只提交暂存区内容

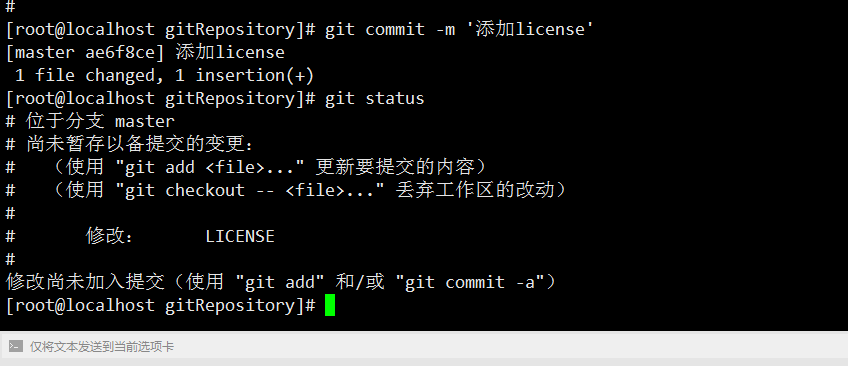
git管理的是修改,git每次提交只会将暂存区的内容commit不会将未添加进工作区的内容commit

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

为什么说Git管理的是修改，而不是文件呢？我们还是做实验。第一步，对readme.txt做一个修改，比如加一行内容,然后git add 然后在添加一行内容然后查看状态可以发现有两个变更,第一个在暂存区需要提交,第二个不在暂存区需要添加进暂存区

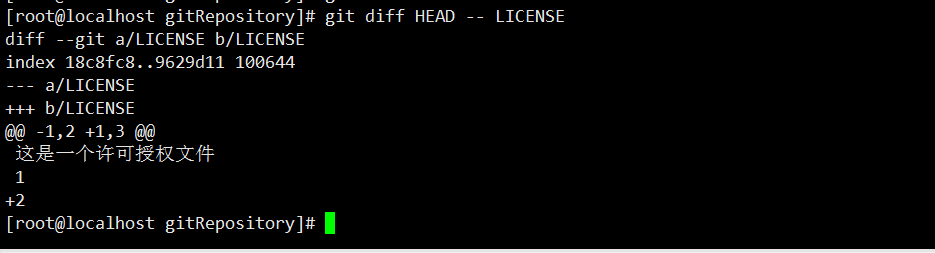


使用git commit后查看状态发现在暂存区的那个更改被提交了,不在暂存区的那个更改没有被提交



你看，我们前面讲了，Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的区别



可见，第二次修改确实没有被提交。

那怎么提交第二次修改呢？你可以继续git add再git commit，也可以别着急提交第一次修改，先git add第二次修改，再git commit，就相当于把两次修改合并后一块提交了：

第一次修改 -> git add -> 第二次修改 -> git add -> git commit

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了Git是如何跟踪修改的，每次修改，如果不add到暂存区，那就不会加入到commit中。

## 撤销更改

可以使用git status查看上面修改后的状态,发现可以使用git checkout -- <file>来丢弃工作区的改动



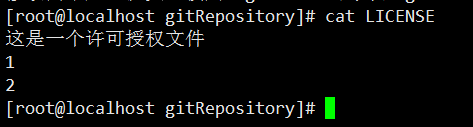
命令git checkout LICENSE意思就是，把LICENSE文件在工作区的修改全部撤销，这里有两种情况：

一种是LICENSE自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是LICENSE已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

现在，看看LICENSE的文件内容：



文件内容果然复原了。

git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

### 总结

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD file，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考版本回退一节，不过前提是没有推送到远程库。

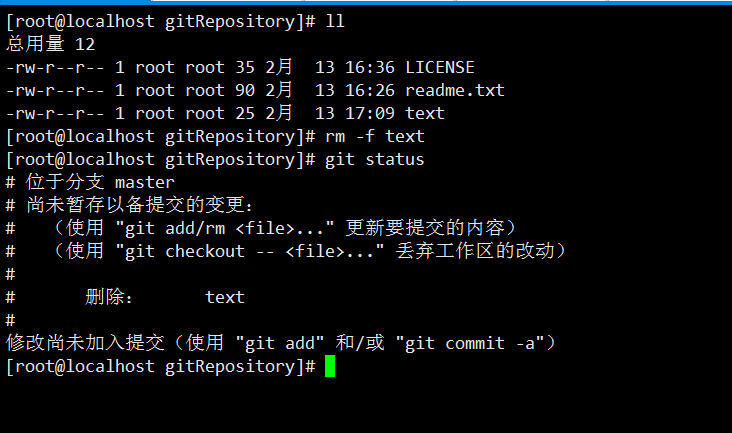
## 删除文件

先添加待删除的测试文件text

然后直接rm –f text

查看状态:发现会提示git rm <file>来删除文件,如果真的想删除文件可使用git rm text即可

如果是误操作删除,可使用git checkout -- file来还原删除操作



git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

小结

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

# 远程仓库

## 远程连接到github

### 生成密钥对

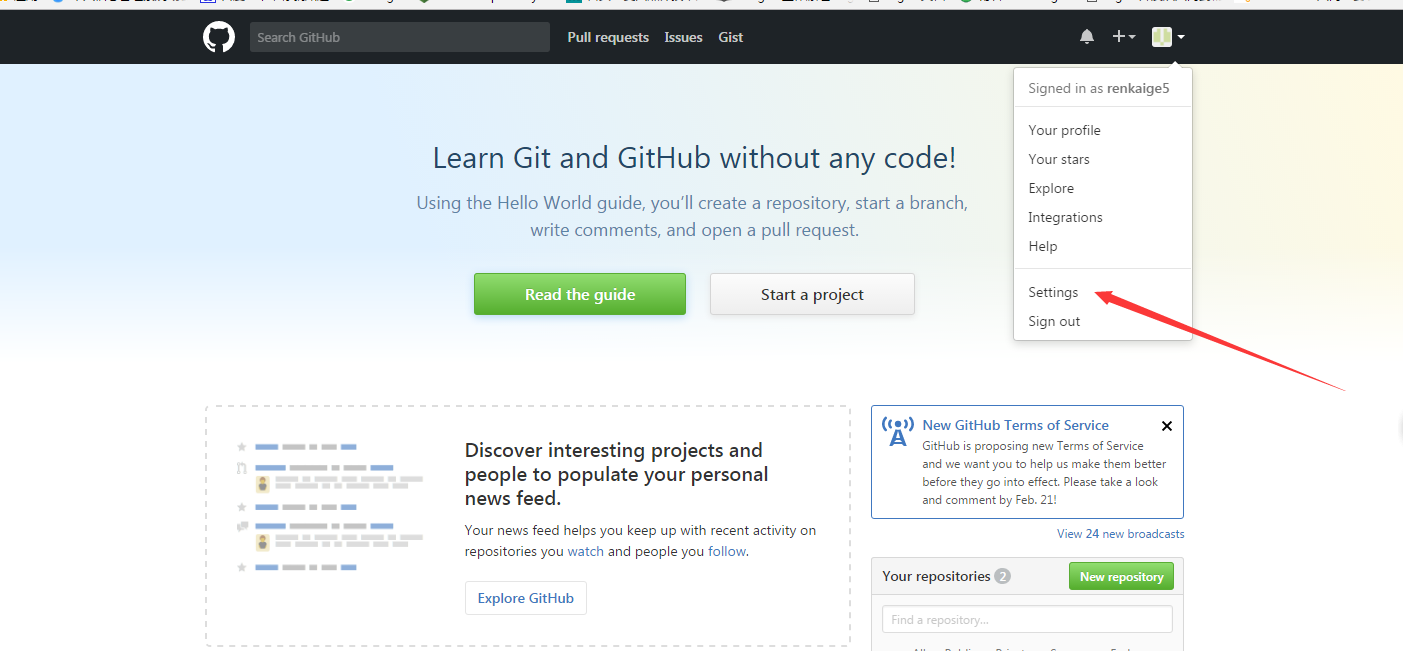
生成ssh免密登录密钥,将自己的公钥发送给github然后在每次上传到github时就不用输入账号密码了

ssh-keygen -t rsa -C "邮箱地址"

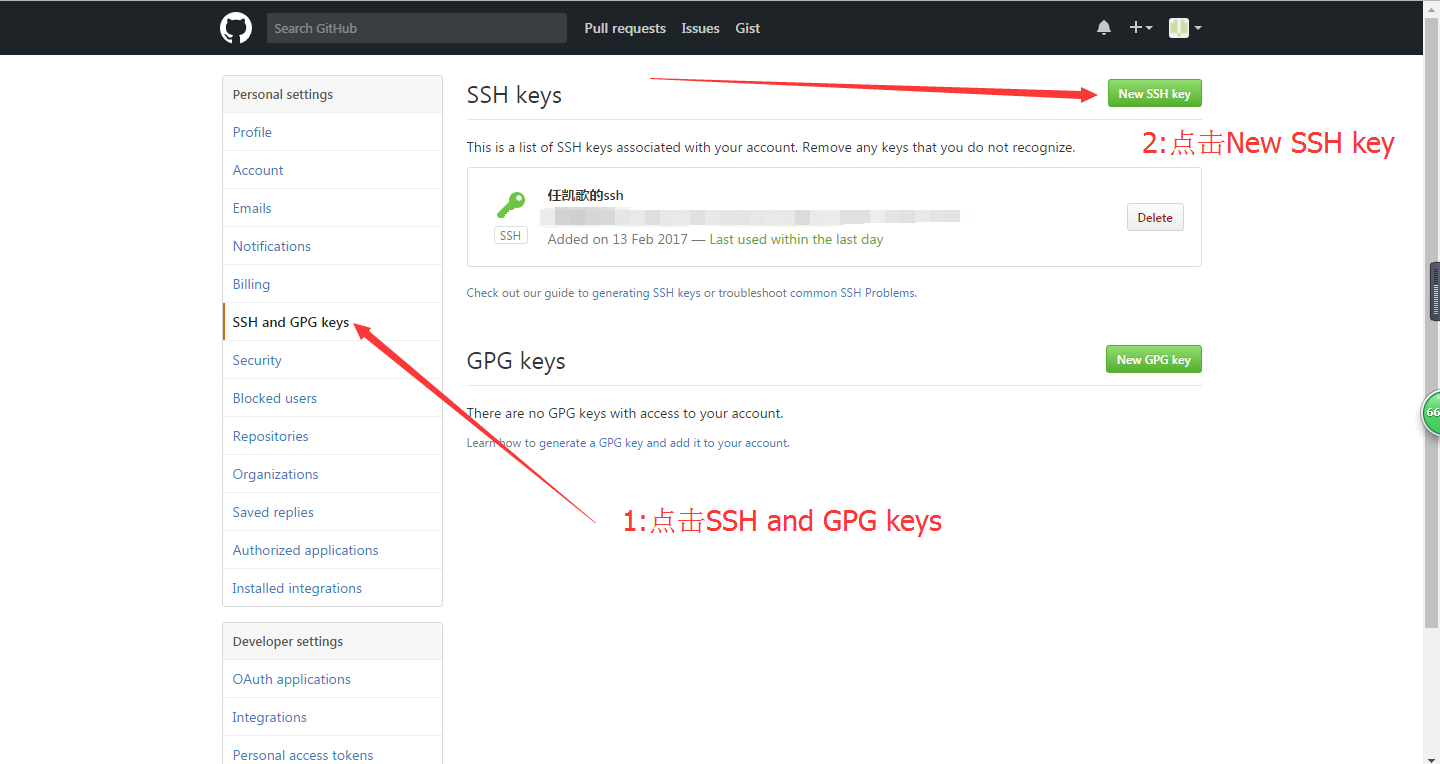
然后去~/.ssh/目录下里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

### 将公钥上传到github中

登录到github,点击setting,



点击ssh and gpg keys-->new ssh key

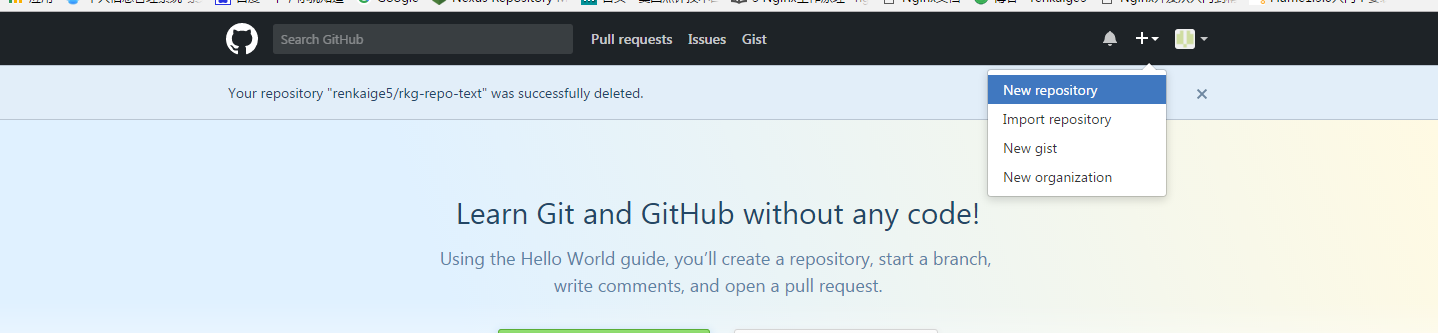


将生成的公钥内容复制到key中,title可以随意写,然后点击add ssh key

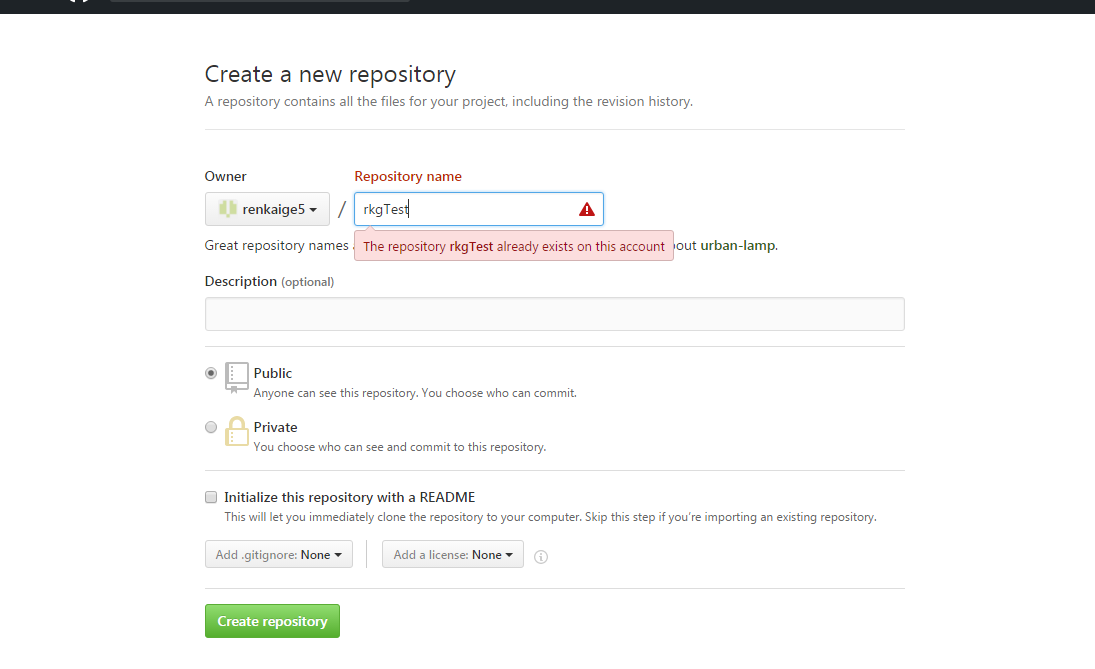


### 添加远程库

点击new repository



输入repository name



根据下面提示在本地仓库下运行如下命令

#### 添加远程库命令

git remote add origin [git@github.com:renkaige5/rkgTest.git](mailto:git@github.com:renkaige5/rkgTest.git)

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

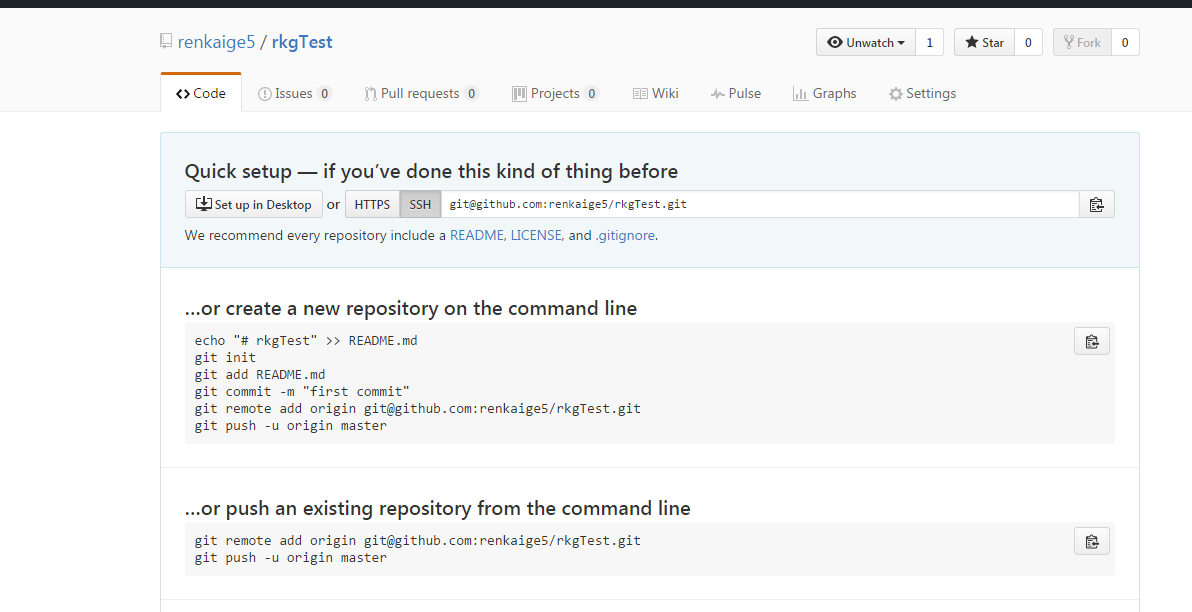
把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：

#### 同步文件到远程库

git push -u origin master

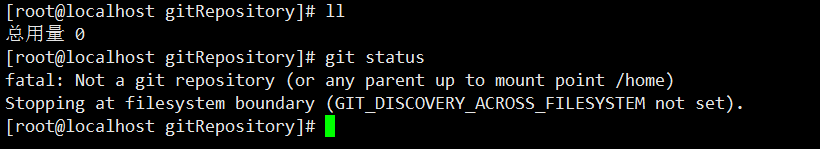


#### 移出本地仓库与远程库的关联

git remote rm origin

### 从远程库克隆

假设本地仓库目前什么东西都没有,可以看出目前gitRepository不是git的工作区里面没有任何东西,就是一个普通的文件夹

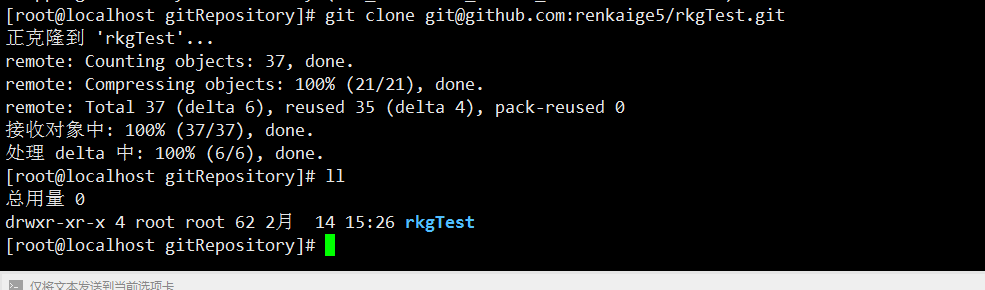


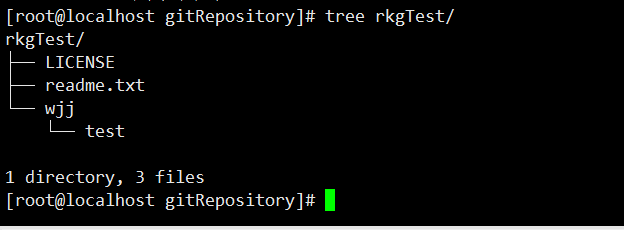
#### 克隆远程仓库

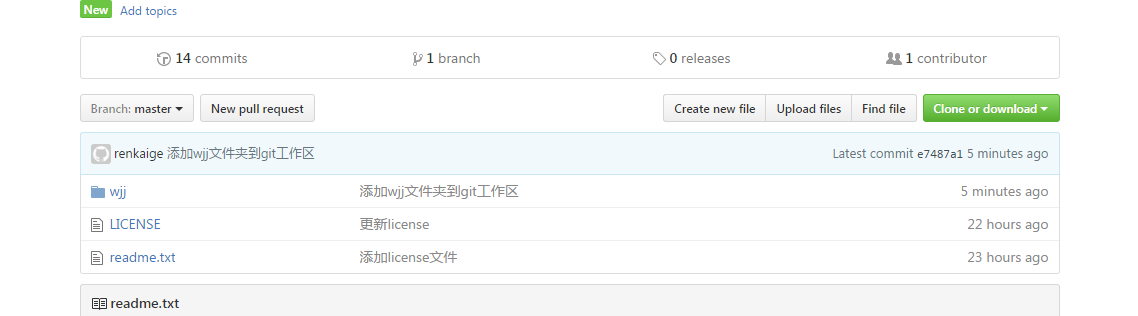
git clone git@github.com:renkaige5/rkgTest.git

语法 git clone [git@github.com:git](mailto:git@github.com:git)hub仓库地址

可以看出clone出来的目录结构和github中目录结构是一样的







#### 推送文件到远程仓库

git push -u origin master

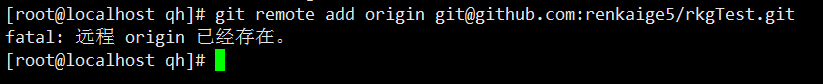
#### 移出本地仓库与远程库的关联

git remote rm origin

### 常见错误

远程origin已经存在

可使用git remote rm origin命令来解决



# 分支管理

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。

## 创建分支

首先，我们创建dev分支，然后切换到dev分支：

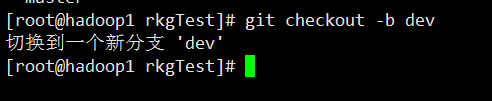
git checkout -b dev

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

git checkout dev

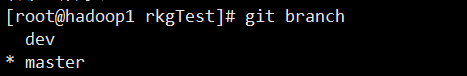
然后，用git branch命令查看当前分支：



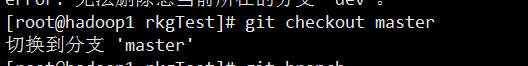
## 查看当前分支

带\*的标示当前所在分支

git branch



## 切换分支

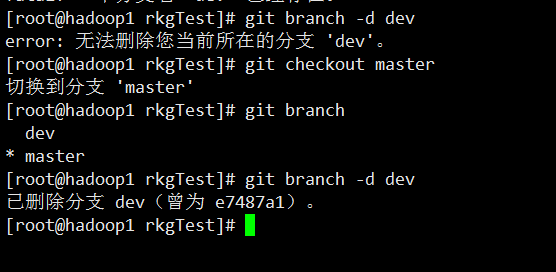


## 删除分支

删除前先保证删除的分支未被使用,如被使用先切换到其他分支

git checkout master

git branch –d dev



## 修改dev分支内容并在master分支查看

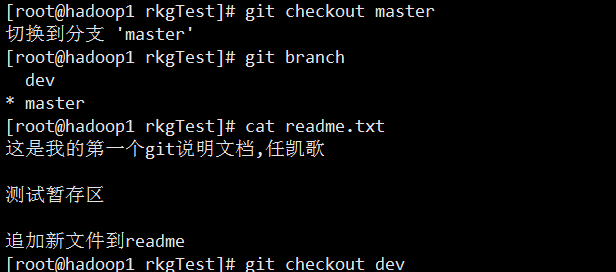
在dev分支中修改readme.txt添加 这是dev分支

然后提交readme.txt

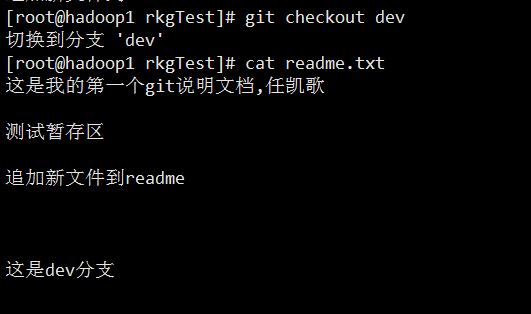
git add readme.txt

git commit -m '测试dev分支'

切换到mster分支并查看readme.txt,可以发现刚才修改的readme.txt在这个里面没有被修改

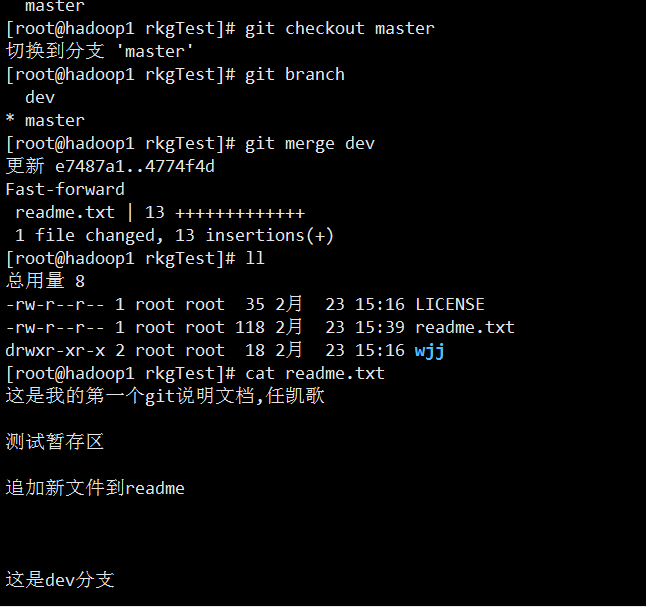


查看dev分支



## 合并分支(将dev修改的内容同步到master)

将dev的内容合并到master上,则需要先切换到master分支上,然后git merge dev即可



## 解决冲突

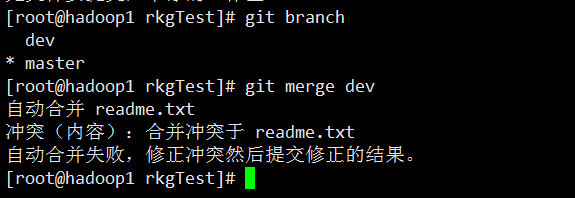
### 创建冲突

在dev分支上修改readme.txt最后一行内容然后提交

在master分支上修改readme.txt最后一行内容然后提交

这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看

此时尝试合并分支会告诉我们readme.txt冲突,自动合并失败

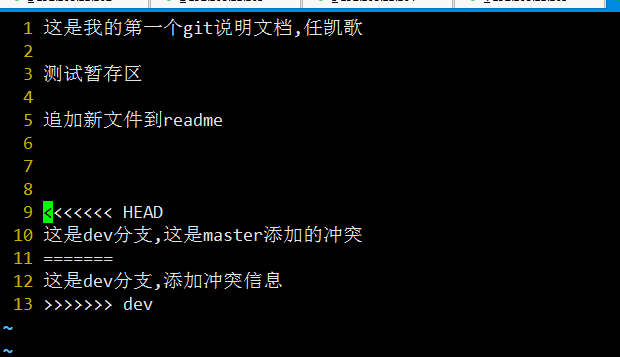


### 使用git status可以发现给出readme.txt冲突的提示



### 我们可以直接查看readme.txt

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

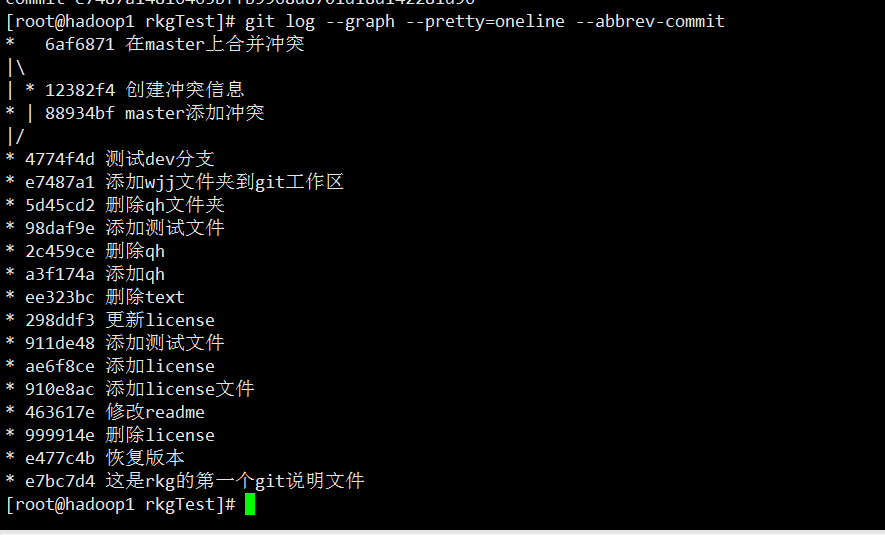


### 解决冲突后,在提交

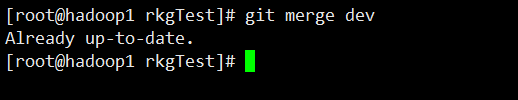


### 查看日志

git log --graph --pretty=oneline --abbrev-commit



### 此时在合并代码会提示已经合并过了



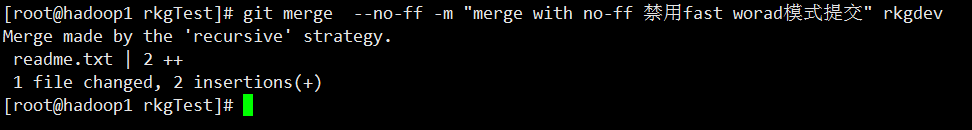
### 小结

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。

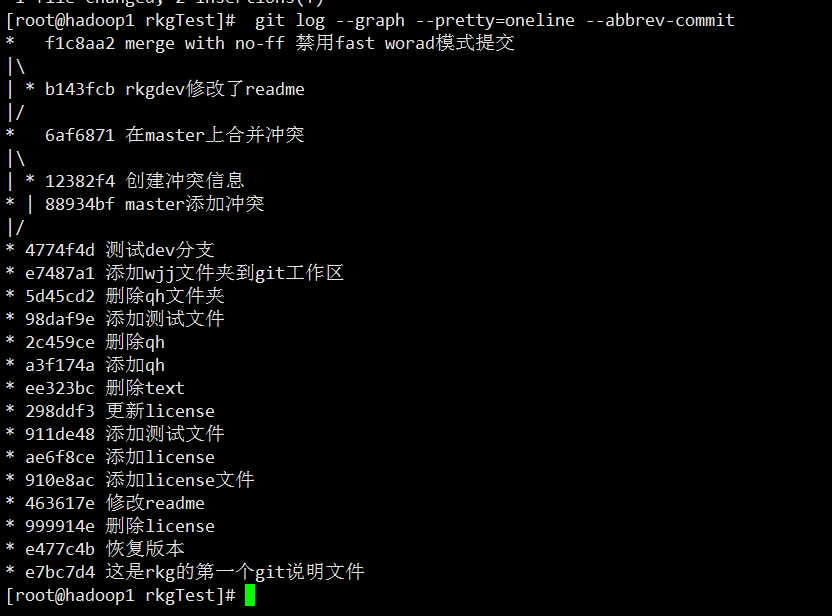
用git log --graph命令可以看到分支合并图。

## 分支管理策略

### 创建新分支rkgdev,然后修改readme,提交,然后切换到master分区,合并分区

git merge --no-ff -m "merge with no-ff 禁用fast worad模式提交" rkgdev

### 查看日志



### 小结

Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

## 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。

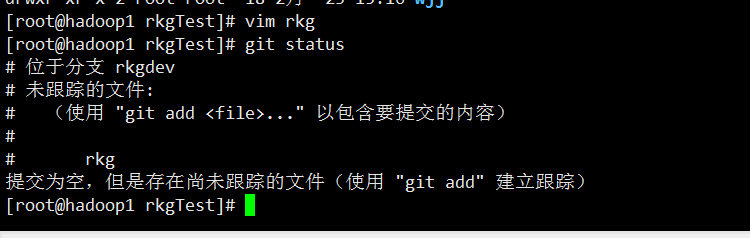
当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

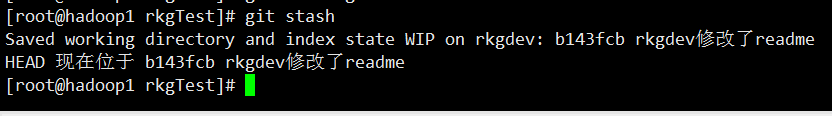
幸好，Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

### 创建一个rkg文件,用于测试

git add rkg ##添加rkg文件到暂存区,但是不提交到分支上



### git stash(将当前的版本储藏起来)



### git status

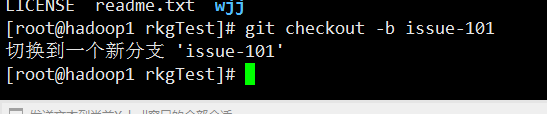
现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。



### 创建临时修改bug的分支

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

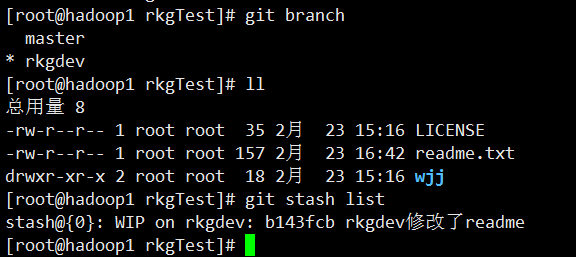
git checkout -b issue-101



### 然后修改bug,然后add到暂存区,然后提交到issue-101分支,然后在master上面合并issue-101分支,合并完分支后可以将issue-101分支删除了

### 现在切换到rkgdev分支继续工作

可以发现刚才创建的rkg文件不在了,

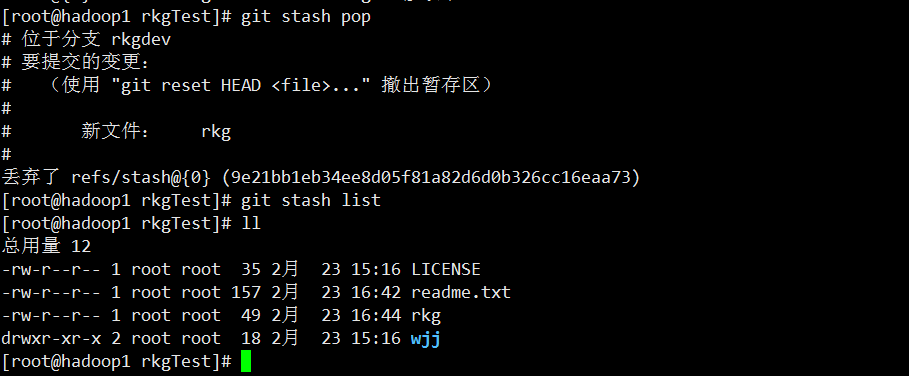


工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

再用git stash list查看，就看不到任何stash内容了：



你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

git stash apply stash@{0}

### 小结

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。

## feature分支

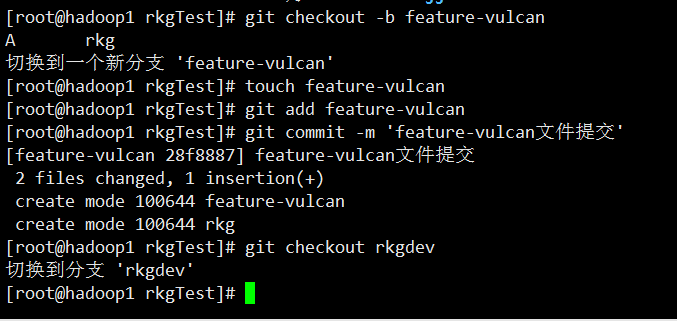
软件开发中，总有无穷无尽的新的功能要不断添加进来。

添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

创建feature-vulcan分支,然后修改提交,然后切换到rkgdev分支



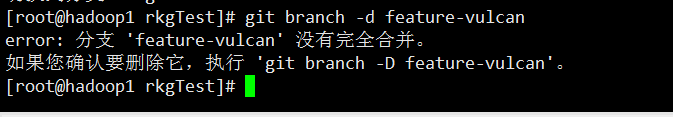
一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，

就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

执行删除操作feature-vulcan



销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令git branch -D feature-vulcan。

git branch -D feature-vulcan



终于删除成功！

小结

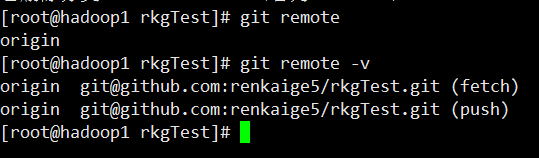
开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过git branch -D <name>强行删除。

## 多人协作

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：或者，用git remote -v显示更详细的信息：

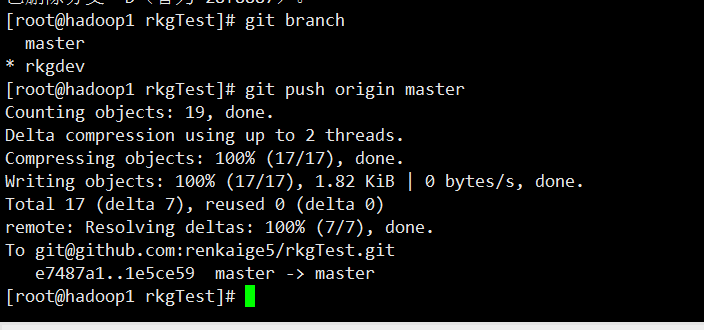


上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

### 推送分支

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

git push origin master



如果要推送其他分支，比如rkgdev，就改成：

git push origin rkgdev

但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

master分支是主分支，因此要时刻与远程同步；

rkgdev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；

feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

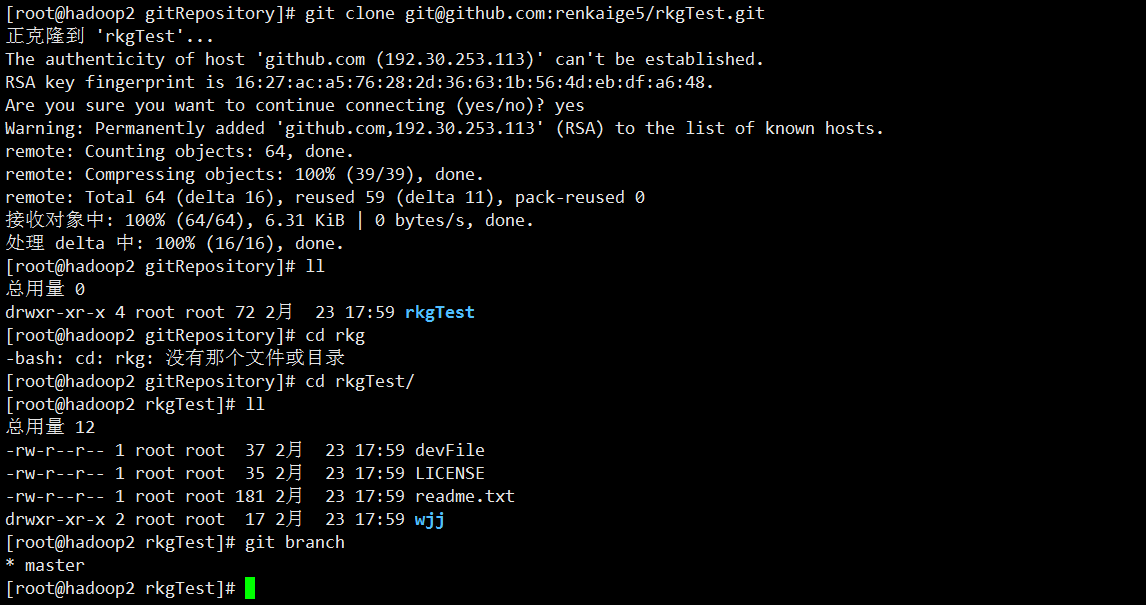
总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

### 抓取分支

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：



现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

162克隆master后创建dev分支并修改了readme文件并push到github上了

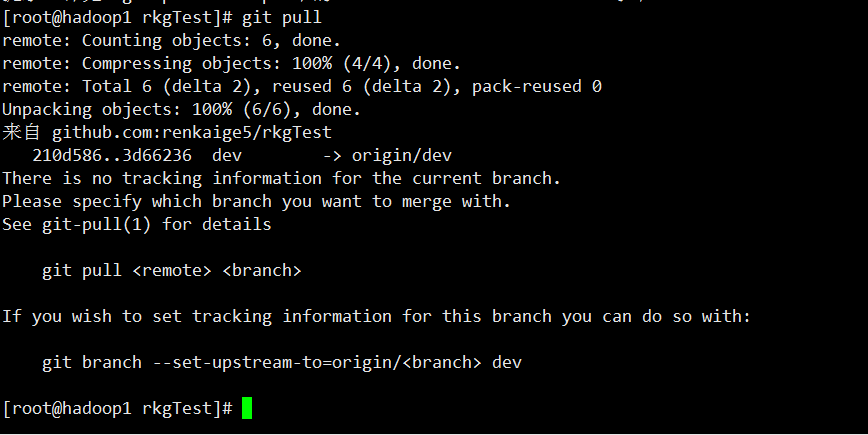


此时161也创建了dev分支并修改readme文件,然后push到github上此时提示

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

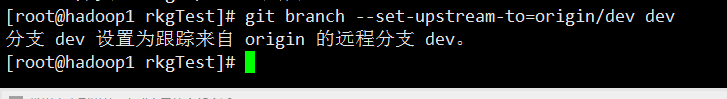


#### git pull(从远程服务器更新代码到本地分支)



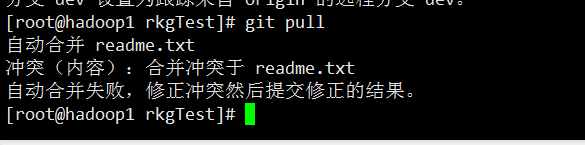
#### git branch --set-upstream-to=origin/dev dev

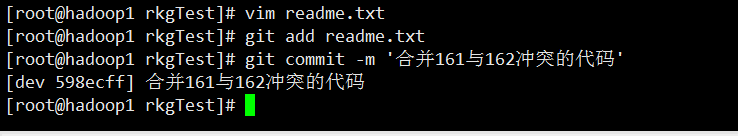
git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

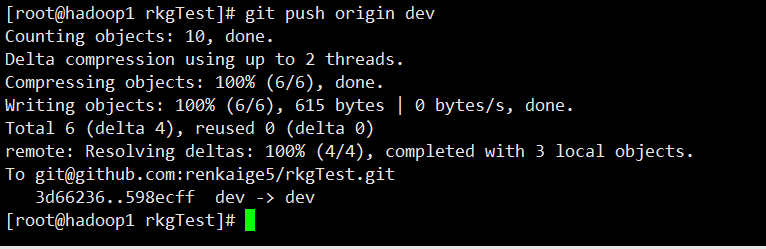


#### 再次git pull

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：







#### 因此，多人协作的工作模式通常是这样：

首先，可以试图用git push origin branch-name推送自己的修改；

如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；

如果合并有冲突，则解决冲突，并在本地提交；

没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

#### 小结

查看远程库信息，使用git remote -v；

本地新建的分支如果不推送到远程，对其他人就是不可见的；

从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；

在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；

建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；

从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

# 标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

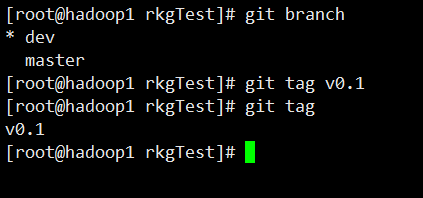
所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

## 创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

创建标签:git tag 标签名称

查看标签(所有的分支):git tag



默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

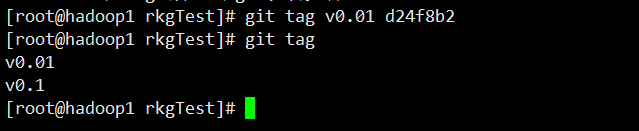
方法是找到历史提交的commit id，然后打上就可以了：

git log --pretty=oneline --abbrev-commit



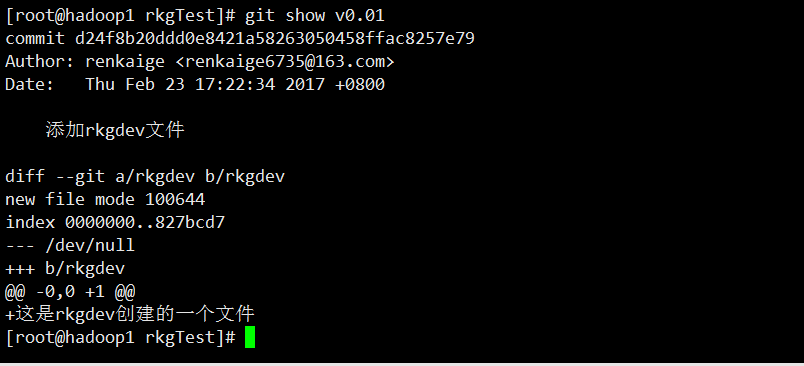
比方说要对rkgdev文件这次提交打标签，它对应的commit id是d24f8b2，敲入命令：

git tag v0.01 d24f8b2



注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

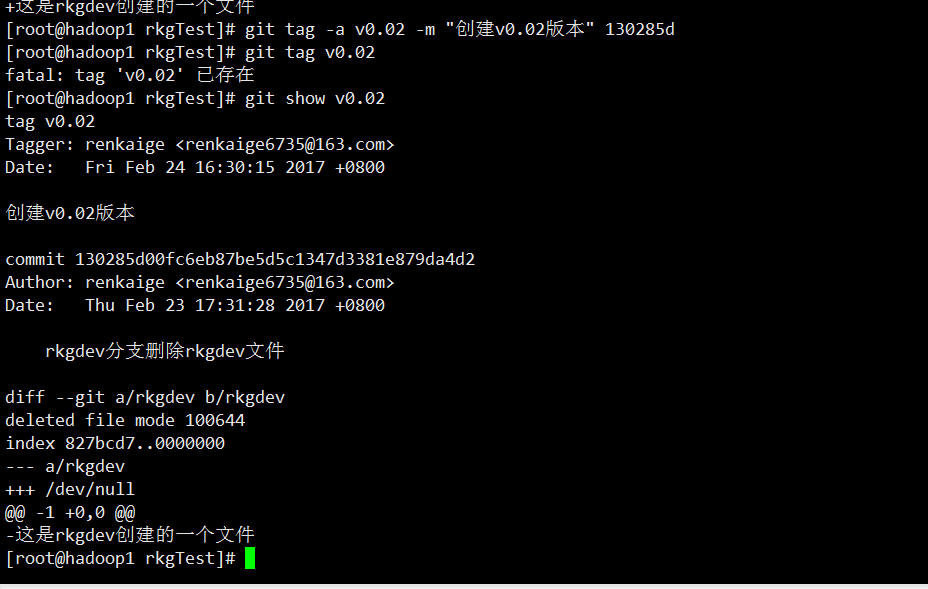
git show v0.01



可以看到，v0.01确实打在rkgdev这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

git tag -a v0.02 -m "创建v0.02版本" 130285d



小结

命令git tag <name>用于新建一个标签，默认为HEAD，也可以指定一个commit id；

git tag -a <tagname> -m "blablabla..."可以指定标签信息；

git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；

命令git tag可以查看所有标签。

## 操作标签

### 删除(本地分支)标签

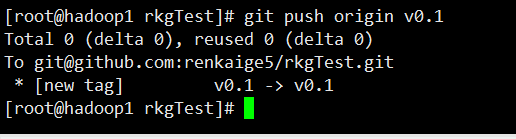
git tag –d 标签名



因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

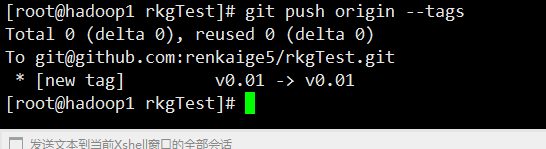
### 推送标签到github

如果要推送某个标签到远程，使用命令git push origin <tagname>：



### 或者，一次性推送全部尚未推送到远程的本地标签：

git push origin --tags



### 删除远程(github)标签

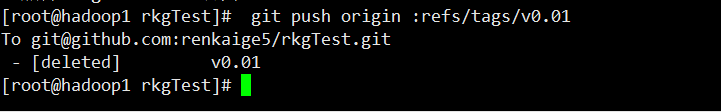
如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

git tag -d v0.01



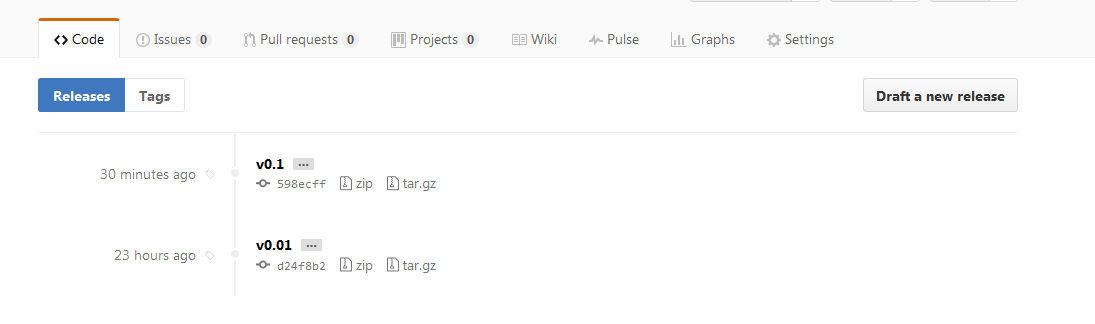
然后，从远程删除。删除命令也是push，但是格式如下：

git push origin :refs/tags/v0.01

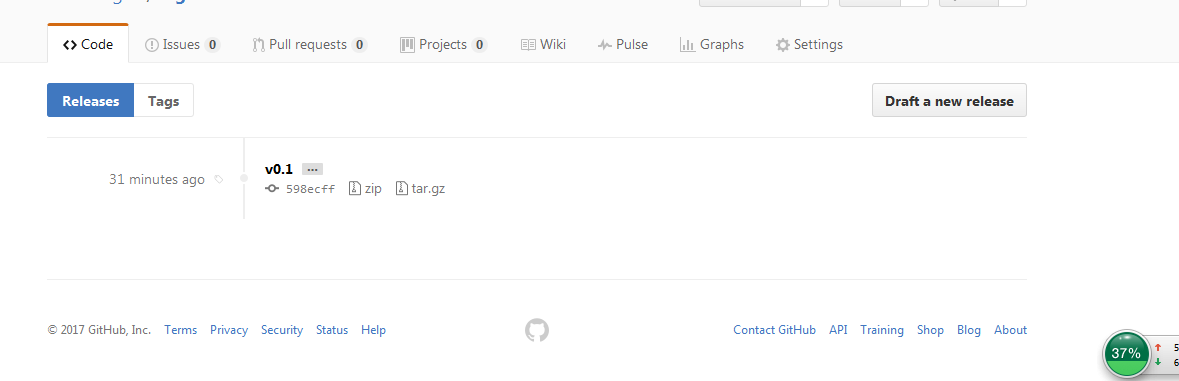


要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

删除前



删除后



### 小结

命令git push origin <tagname>可以推送一个本地标签；

命令git push origin --tags可以推送全部未推送过的本地标签；

命令git tag -d <tagname>可以删除一个本地标签；

命令git push origin :refs/tags/<tagname>可以删除一个远程标签。

# 使用github

我们一直用GitHub作为免费的远程仓库，如果是个人的开源项目，放到GitHub上是完全没有问题的。其实GitHub还是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

在GitHub出现以前，开源项目开源容易，但让广大人民群众参与进来比较困难，因为要参与，就要提交代码，而给每个想提交代码的群众都开一个账号那是不现实的，因此，群众也仅限于报个bug，即使能改掉bug，也只能把diff文件用邮件发过去，很不方便。

但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页https://github.com/twbs/bootstrap，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

git clone git@github.com:michaelliao/bootstrap.git

一定要从自己的账号下clone仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址git@github.com:twbs/bootstrap.git克隆，因为没有权限，你将不能推送修改。

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：

github-repos

如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

如果你没能力修改bootstrap，但又想要试一把pull request，那就Fork一下我的仓库：https://github.com/michaelliao/learngit，创建一个your-github-id.txt的文本文件，写点自己学习Git的心得，然后推送一个pull request给我，我会视心情而定是否接受。

小结

在GitHub上，可以任意Fork开源仓库；

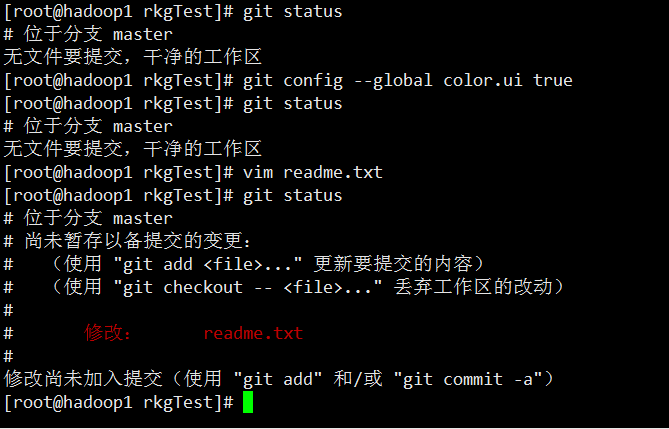
自己拥有Fork后的仓库的读写权限；

可以推送pull request给官方仓库来贡献代码。

# 自定义git

修改git显示颜色

git config --global color.ui true



## 忽略特殊文件

有些时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files ...，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要组合一下就可以使用了。所有配置文件可以直接在线浏览：https://github.com/github/gitignore

忽略文件的原则是：

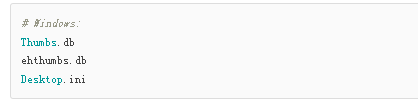
忽略操作系统自动生成的文件，比如缩略图等；

忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；

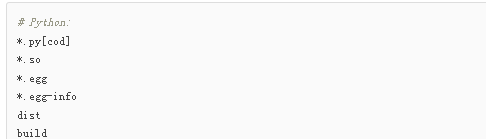
忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

举个例子：

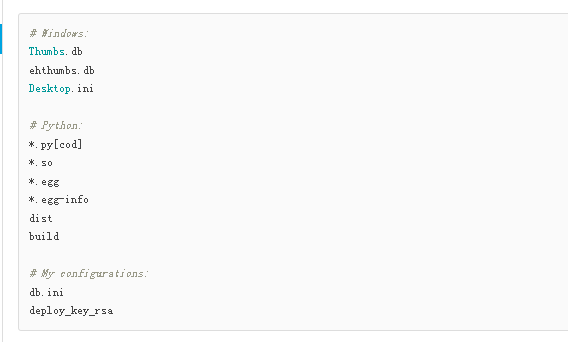
假设你在Windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有Desktop.ini文件，因此你需要忽略Windows自动生成的垃圾文件：



然后，继续忽略Python编译产生的.pyc、.pyo、dist等文件或目录：



加上你自己定义的文件，最终得到一个完整的.gitignore文件，内容如下：



最后一步就是把.gitignore也提交到Git，就完成了！当然检验.gitignore的标准是git status命令是不是说working directory clean。

使用Windows的童鞋注意了，如果你在资源管理器里新建一个.gitignore文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。

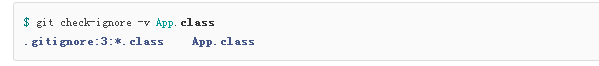
有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：



如果你确实想添加该文件，可以用-f强制添加到Git：



或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check-ignore命令检查：



Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

小结

忽略某些文件时，需要编写.gitignore；

.gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

## 配置别名

有没有经常敲错命令？比如git status？status这个单词真心不好记。

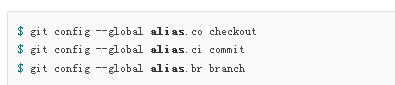
如果敲git st就表示git status那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：



好了，现在敲git st看看效果。

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：



以后提交就可以简写成：



--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

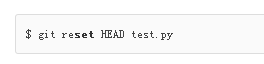
在[撤销修改](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/001374831943254ee90db11b13d4ba9a73b9047f4fb968d000)一节中，我们知道，命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：



当你敲入命令：



实际上Git执行的是：



配置一个git last，让其显示最后一次提交信息：



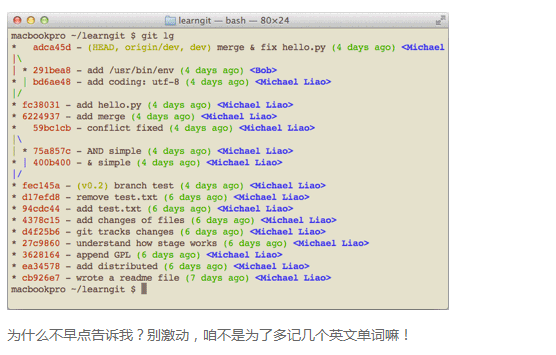
这样，用git last就能显示最近一次的提交：



甚至还有人丧心病狂地把lg配置成了：



来看看git lg的效果：



### 配置文件

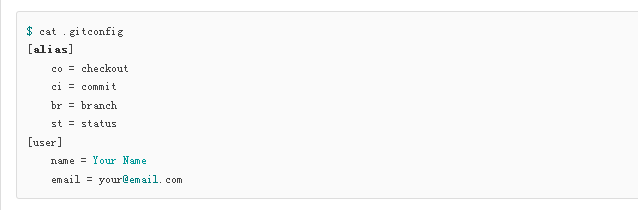
配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中：



别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中：



配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

### 小结

给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。

## 搭建git服务器