目录

[一、 安装 1](#_Toc2807138)

[（一）Git客户端下载 1](#_Toc2807139)

[（二）安装过程 2](#_Toc2807140)

[二、 Git使用 9](#_Toc2807141)

[(一) 打开Git Bash，配置用户名和邮箱 9](#_Toc2807142)

[三、 时光机穿梭 11](#_Toc2807143)

[(1)版本回退 13](#_Toc2807144)

[(2)工作区和暂存区 15](#_Toc2807145)

[(3)管理修改 18](#_Toc2807146)

[(4)撤销修改 20](#_Toc2807147)

[(5)删除文件 23](#_Toc2807148)

[四、 远程仓库 24](#_Toc2807149)

[(1)添加远程库 24](#_Toc2807150)

[(2)从远程库克隆 29](#_Toc2807151)

[五、 分支管理 31](#_Toc2807152)

[(1)创建与合并分支 31](#_Toc2807153)

[(2)解决冲突 34](#_Toc2807154)

[(3) 分支管理策略。 37](#_Toc2807155)

[(4) bug分 39](#_Toc2807156)

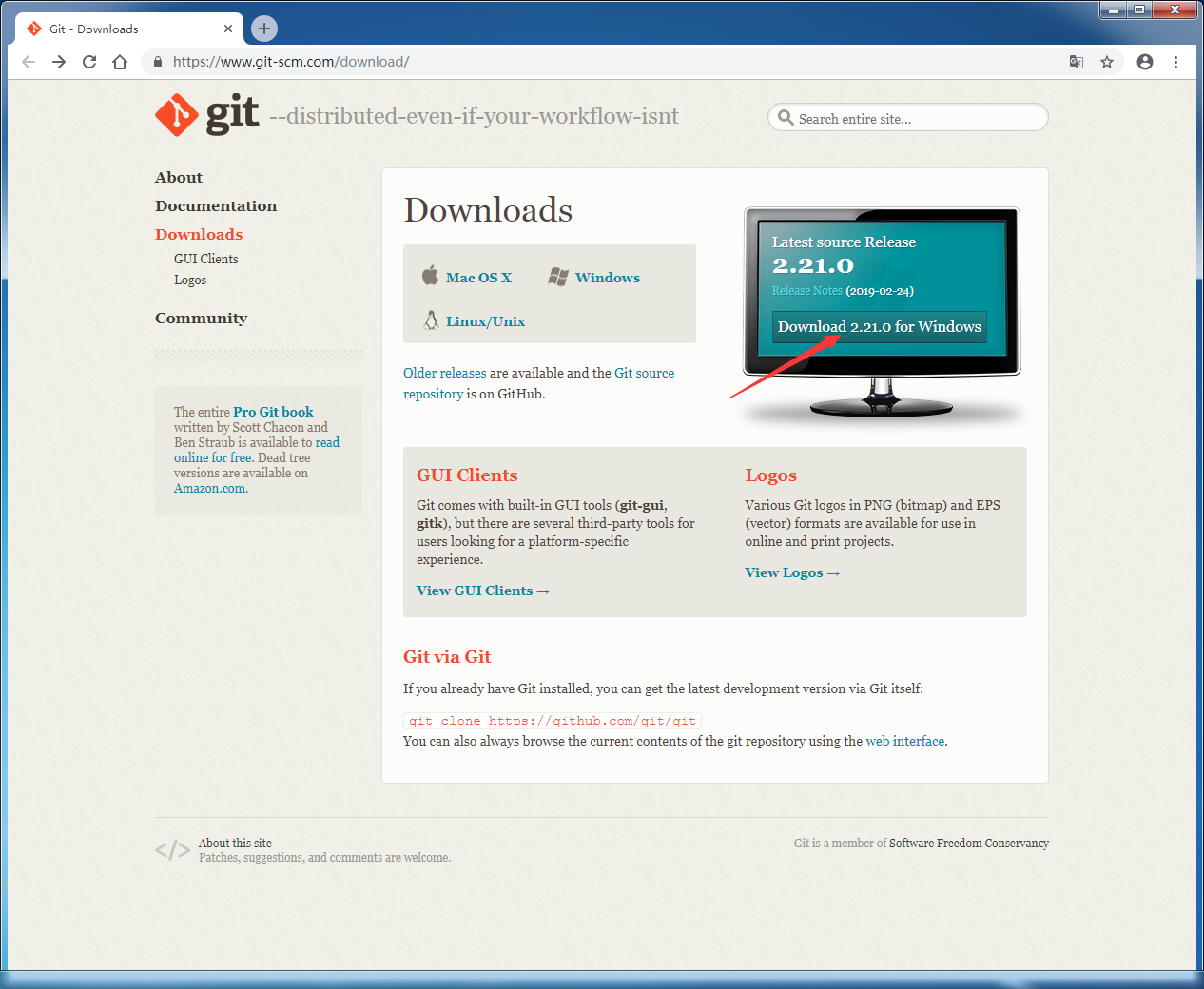
[(5)多人协作 41](#_Toc2807157)

# 

# 安装

### （一）Git客户端下载

https://www.git-scm.com/download/

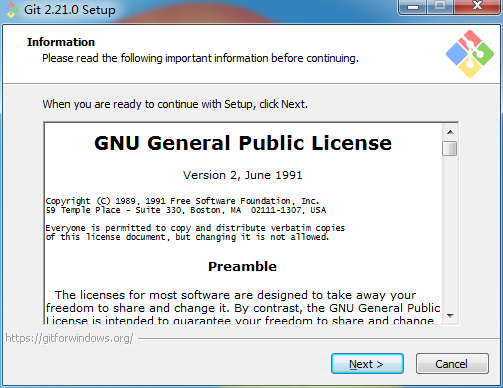


**图1.1.1**

### 

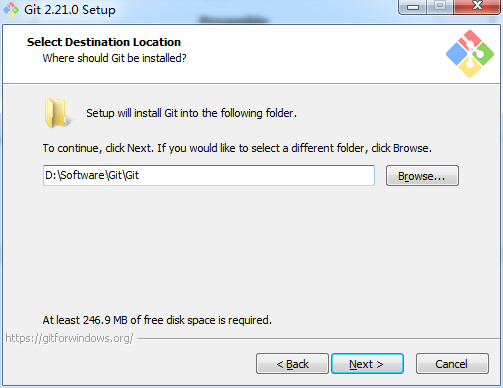
### （二）安装过程

1.1双击安装程序“Git-2.21.0-64-bit.exe”，显示截图如下：



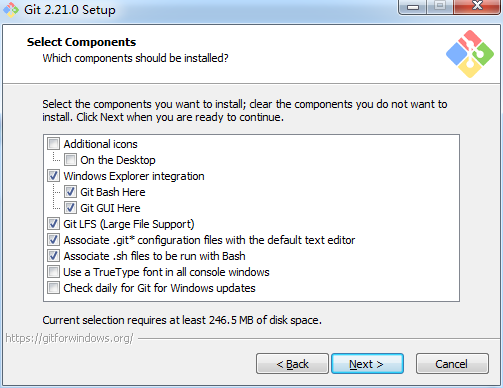
**图1.2.1**

1.2 点击“Next”，自定义选择安装位置，显示截图如下：

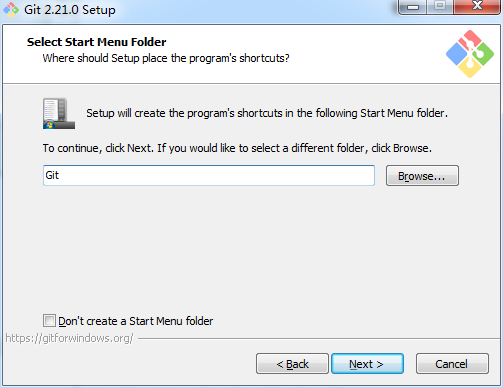


**图1.2.2**

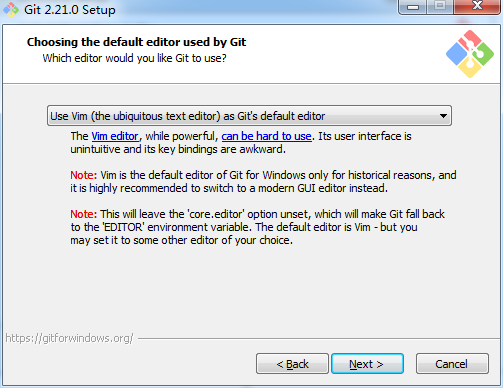
1.3 一直点击“Next”到安装成功，显示截图如下：



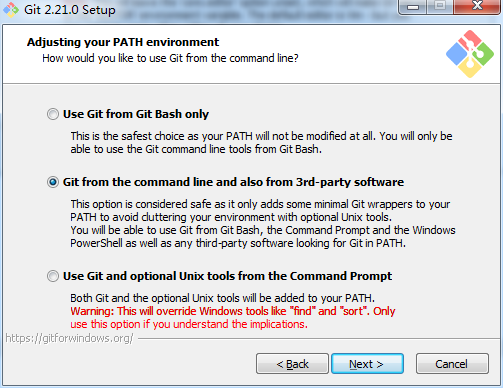
**图1.2.3**



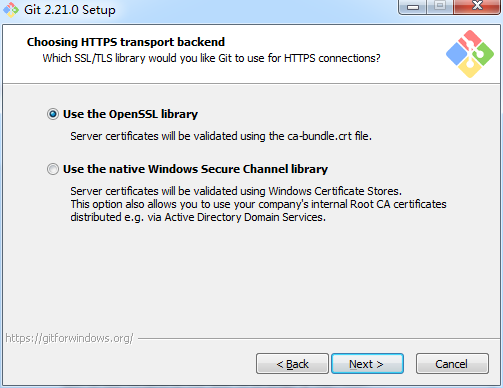
**图1.2.4**



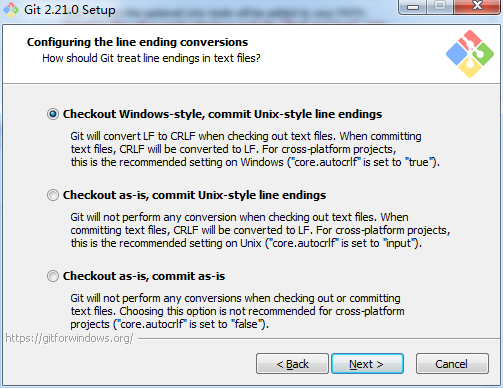
**图1.2.5**



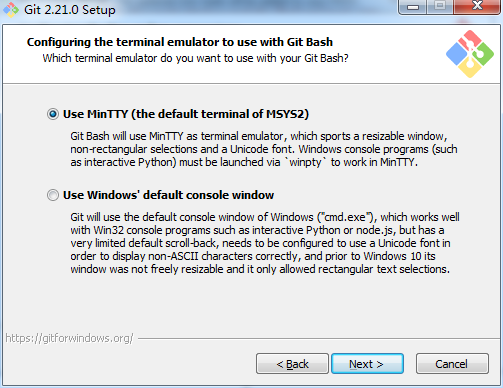
**图1.2.6**



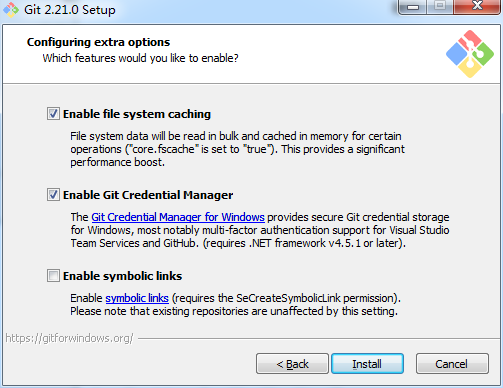
**图1.2.7**



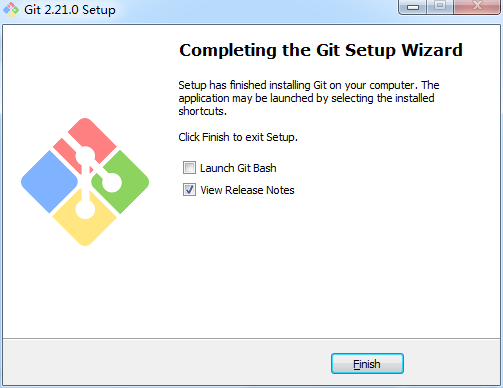
**图1.2.8**



**图1.2.9**



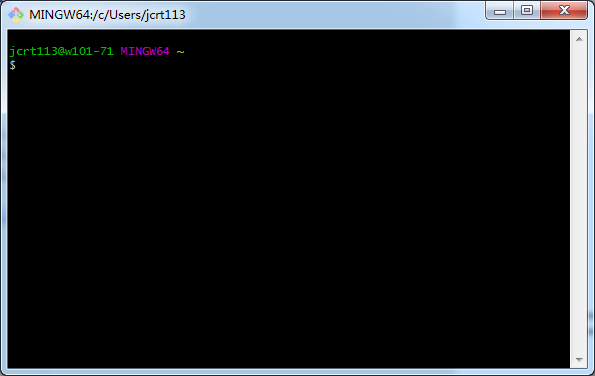
**图1.2.10**



**图1.2.11**

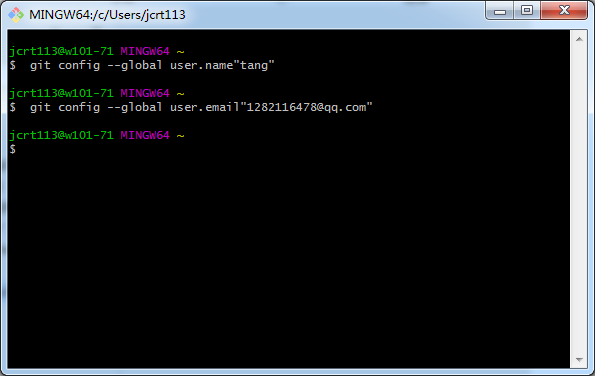
# Git使用

### 打开Git Bash，配置用户名和邮箱



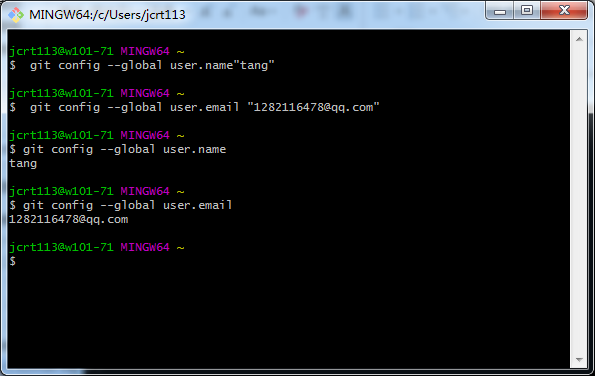
**图2.1.1**

配置用户名和邮箱



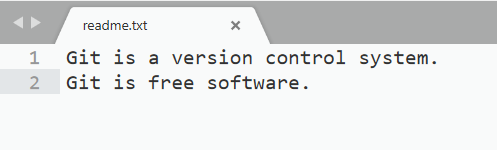
**图2.1.2**

查看配置的用户名和邮箱



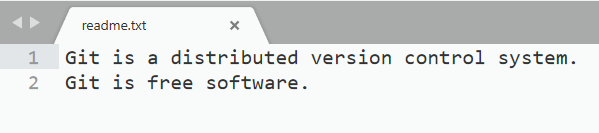
**图2.1.3**

# 时光机穿梭



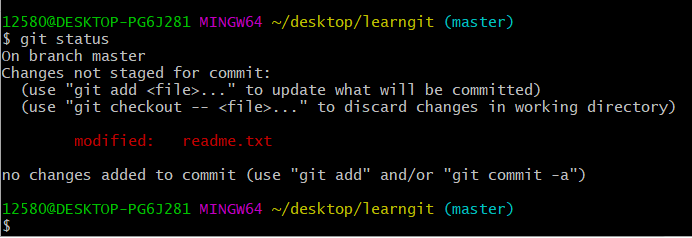
**图3.1 readme.txt文件**

已经提交进入git仓库的readme.txt文件，现在我们对它进行改动，改动如下：



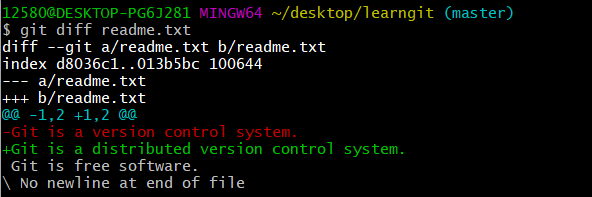
**图3.2修改后的readme.txt文件**

保存修改后，使用git status命令查看结果，git status命令可以让我们时刻掌握仓库当前的状态：



**图3.3 git status命令查看**

上面的命令输出告诉我们，readme.txt被修改过了，但还没有准备提交的修改。虽然Git告诉我们readme.txt被修改了，但如果能看看具体修改了什么内容，自然是很好的。此时我们可以使用git diff命令，当然，要看某一个文件的修改信息，需要加上文件名：



**图3.4 git diff命令查看具体修改信息**

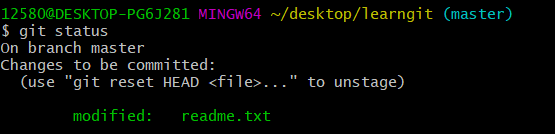
git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第一行添加了一个distributed单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add：



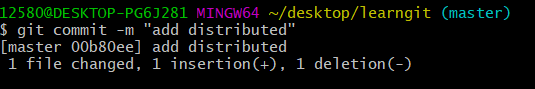
**图3.5 git add readme.txt**

没有任何输出，我们再次使用git status命令查看状态信息：



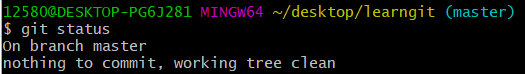
**图3.6 查看git状态信息**

对比上一次的状态信息，我们可以发现，readme.txt从红色变成了绿色，并且提示我们可以将其提交，接下来我们放心的执行提交命令：



**图3.7 git commit提交修改**

提交后，我们再用git status命令看看仓库的当前状态：



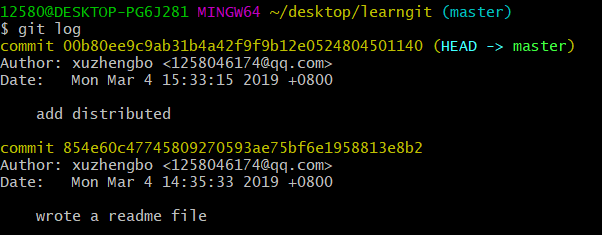
**图3.8 查看提交后的状态**

Git告诉我们当前没有需要提交的修改，而且，工作目录是干净（working tree clean）的。

### (1)版本回退

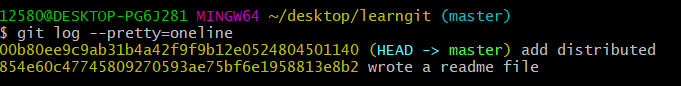
现在，我们已经完成了修改文件，然后把修改提交到Git版本库。每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

在之前的工作中，我们一共提交了两个版本的readme.txt文件，我们可以使用git log命令查看版本信息：



**图3.9 git log查看版本信息**

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：



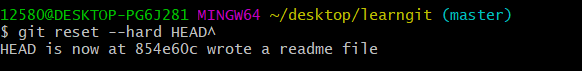
**图3.10 git log --pretty=oneline查看简略的版本信息**

需要友情提示的是，你看到的一大串类似00b80e...的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

好了，现在我们启动时光穿梭机，准备把readme.txt回退到上一个版本，也就是wrote a readme file的那个版本，怎么做呢？

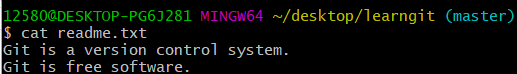
首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交00b80e...（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

现在，我们要把当前版本add distributed回退到上一个版本wrote a readme file，就可以使用git reset命令：



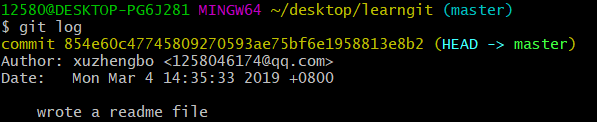
**图3.11 回退版本**

看看readme.txt的内容是不是版本wrote a readme file，我们可以使用cat命令：



**图3.12 查看回退版本后的readme.txt的内容**

从内容中我们可以看出，readme.txt已经被成功还原了。我们用git log再看看现在版本库的状态：



**图3.13 查看回退版本后的版本信息**

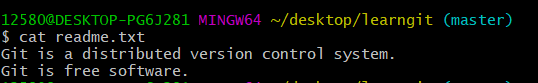
最新的那个版本add distributed已经看不到了！好比你从21世纪坐时光穿梭机来到了19世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个add distributed的commit id是00b80e...，于是就可以指定回到未来的某个版本：



**图3.14 指定到特定的某个版本**

版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。再小心翼翼地看看readme.txt的内容：



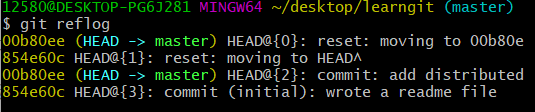
**图3.15查看readme.txt文件内容**

从内容我们可以看见版本已经指定到add distributed的版本。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向add distributed，改为指向wrote a readme file，然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到wrote a readme file版本时，再想恢复到add distributed，就必须找到add distributed的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：



**图3.16 查看head指针变动信息**

终于舒了口气，从输出可知，add distributed的commit id是00b80ee，现在，你又可以乘坐时光机回到未来了。

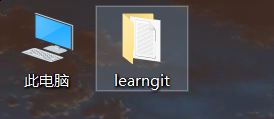
### (2)工作区和暂存区

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

先来看名词解释。

#### 3.2.1 工作区（Working Directory）

就是你在电脑里能看到的，设置为git仓库的目录，比如我的learngit文件夹就是一个工作区：



**图3.17 工作区**

#### 3.2.2 版本库（Repository）

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。

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：

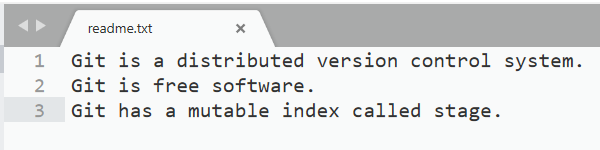
第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

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。

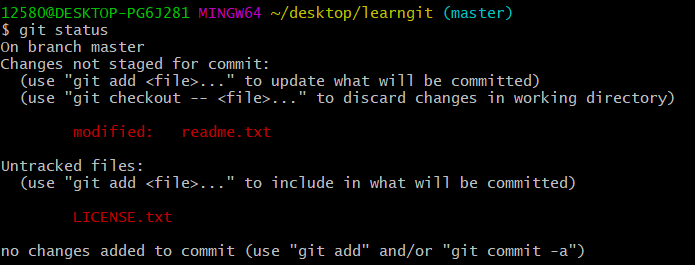
俗话说，实践出真知。现在，我们再练习一遍，先对readme.txt做个修改，比如加上一行内容：



**图3.18 加入一行内容**

然后，在工作区新增一个LICENSE文本文件（内容随便写）。

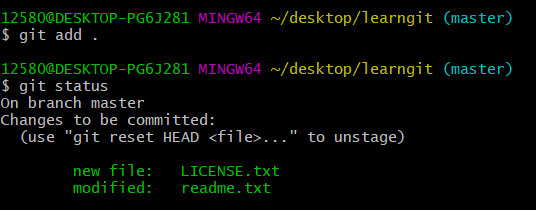
先用git status查看一下状态：



**图3.19 查看修改readme.txt文件以及加入LICENSE.txt文件后的状态**

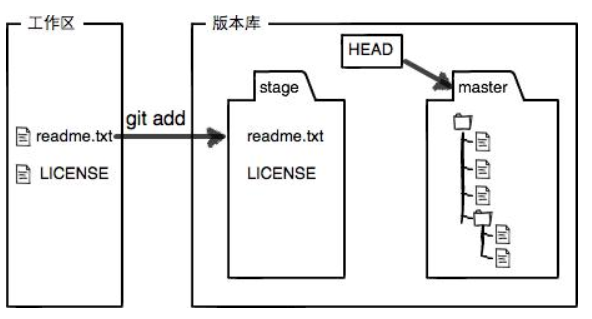
Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用命令git add . （此处的.意思是讲该文件夹下面的所有可add文件执行add操作）把readme.txt和LICENSE都添加后，用git status再查看一下：



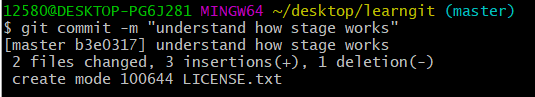
**图3.20 先添加，后查看状态**

现在，暂存区的状态就变成这样了：



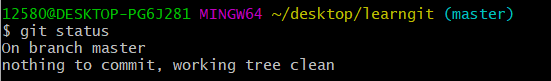
**图3.21 当前暂存区内容**

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：



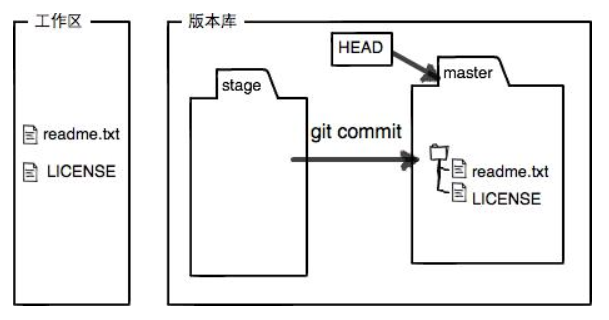
**图3.22提交所有修改**

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，使用git status进行查看：

****

**图3.23 查看状态**

现在版本库变成了这样，暂存区就没有任何内容了：



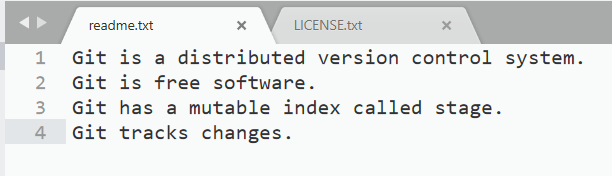
**图3.24 进行git commit操作后，暂存区与版本库示意图**

### (3)管理修改

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

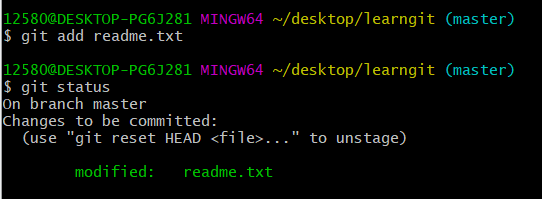
你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

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：



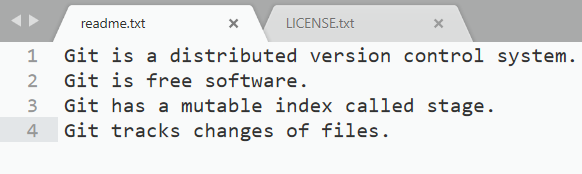
**图3.25 readme.txt增加一行**

然后，添加并查看当前状态：



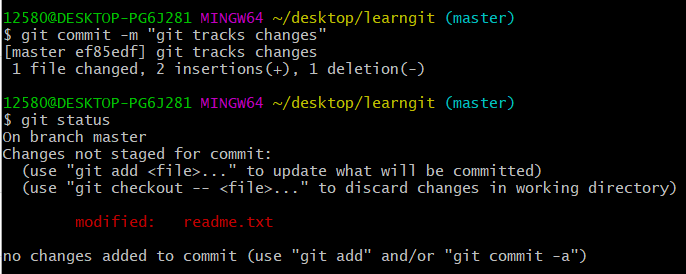
**图3.26 第一次修改并提交**

然后，再修改readme.txt：



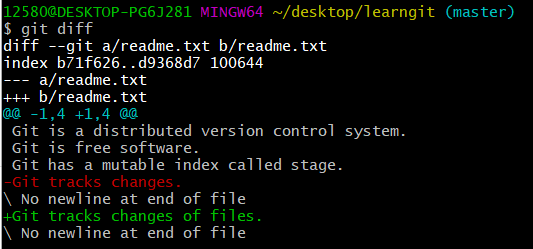
**图3.27 第二次修改**

进行提交，并查看状态：



**图3.28 提交，并查看状态**

这里就会发现问题，我们明明提交了，可是为什么还是显示有改变未被提交？我们使用git diff查看下具体的修改信息：



**图3.29 查看修改信息**

你是不是发现了，这里的修改信息是我们第二次进行修改时的修改信息，那么问题来了，为什么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

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

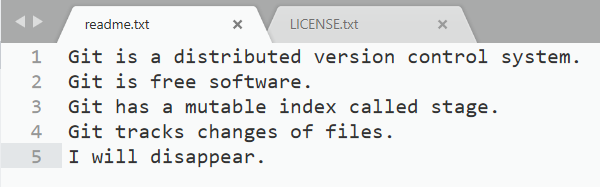
你看，我们前面讲了，Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

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

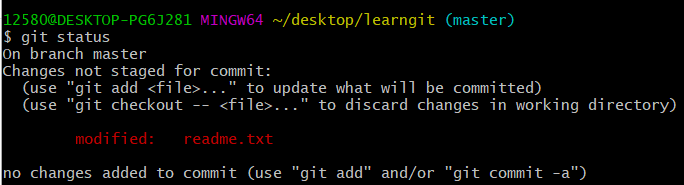
### (4)撤销修改

在readme.txt文件中增加一行：



**图3.30修改readme.txt文件**

查看当前状态：



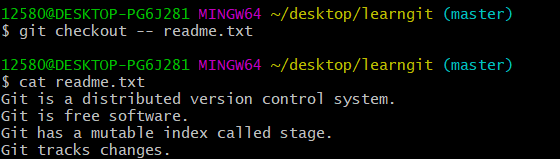
**图3.31 修改后的状态**

你可以发现，Git会告诉你，git checkout -- file可以丢弃工作区的修改，命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt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readme.txt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。

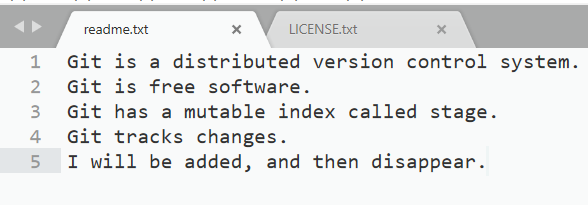
总之，就是让这个文件回到最近一次git commit或git add时的状态，进行git checkout -- readme.txt以及cat readme.txt操作：



**图3.32 执行撤销操作的readme.txt文件**

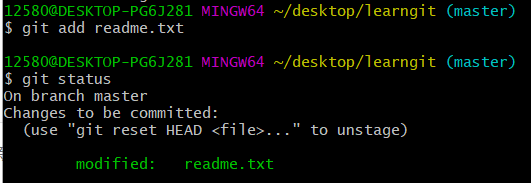
git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令。

刚才是没有add的情况下进行撤销操作，现在我们再学习下进行了add操作后的而撤销操作。首先在readme.txt文件中添加一句话：



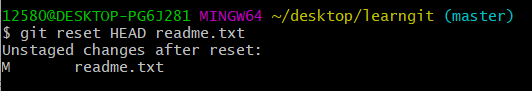
**图3.33 修改readme.txt文件**

添加，并查看状态：



**图3.34 查看状态**

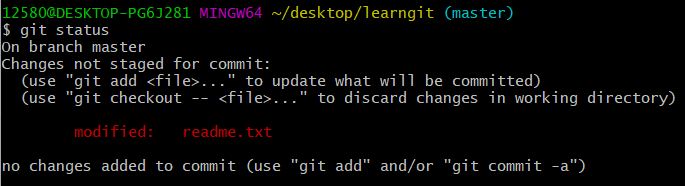
Git告诉我们，用命令git reset HEAD <file>可以把暂存区的修改撤销掉（unstage），重新放回工作区：



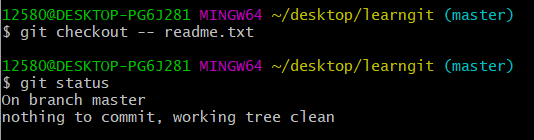
**图3.35 撤销暂存区修改，放回工作区**

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

再用git status查看一下，现在暂存区是干净的，工作区有修改：



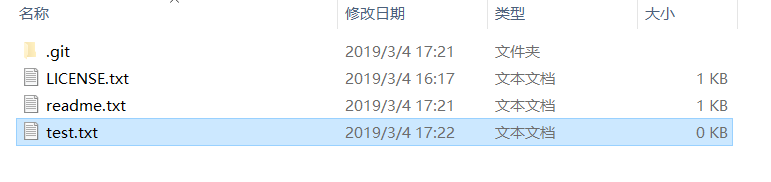
然后我们就可以使用刚才学到的git checkout -- readme.txt将工作区的修改撤销掉：



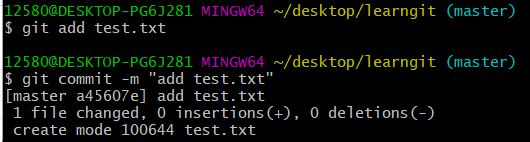
**图3.36 撤销工作区的修改**

### (5)删除文件

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

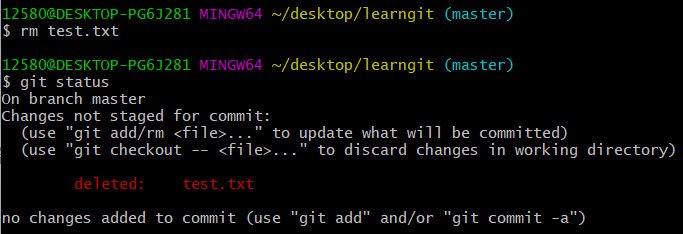


**图3.37 添加test.txt文件**



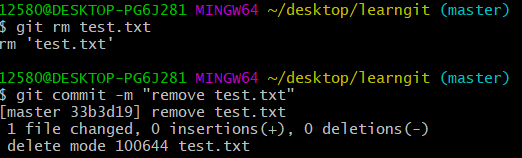
**图3.38 提交test.txt文件**

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了，执行rm操作，并查看状态：



**图3.39 删除test.txt文件，并查看状态**

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了。现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：



**图3.40 删除版本库中的test.txt文件**

另一种情况是删错了，因为版本库里还有呢，所以可以使用git checkout --test.txt很轻松地把误删的文件恢复到最新版本。

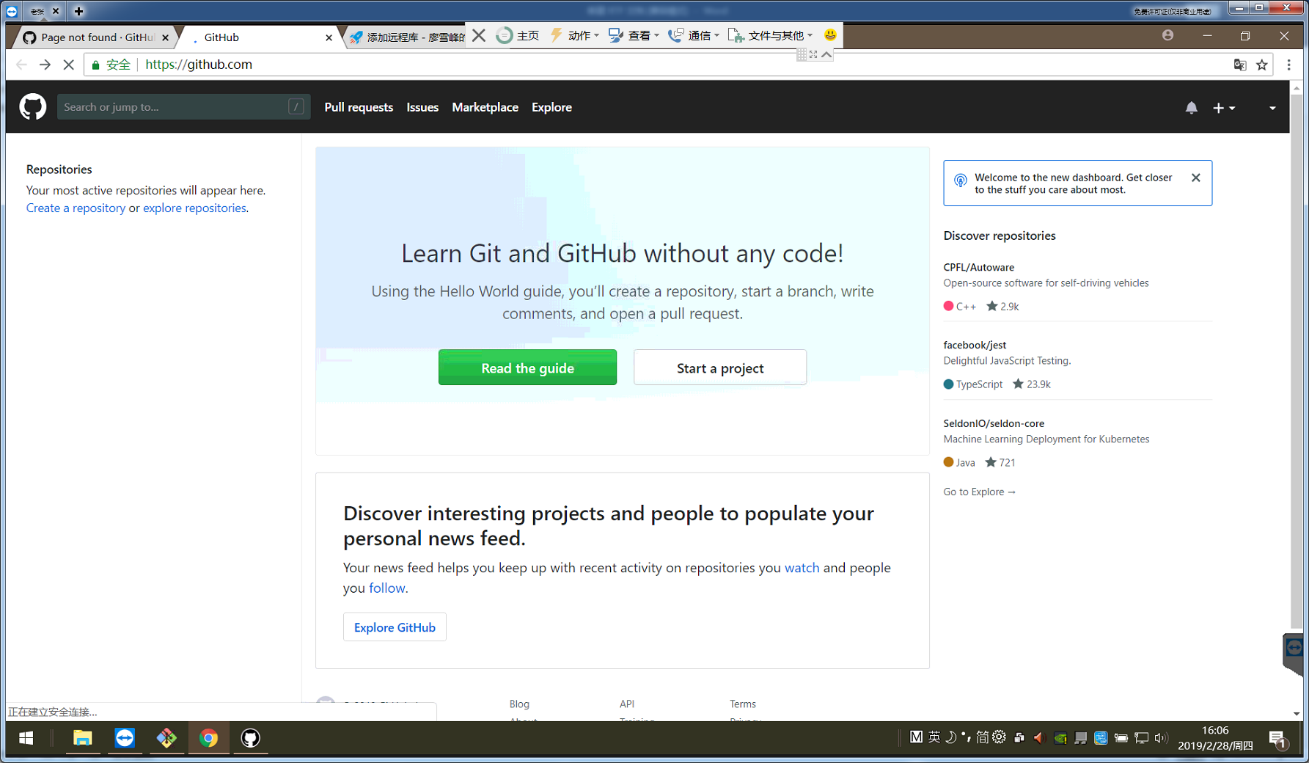
# 远程仓库

首先要自行注册github账号并登陆

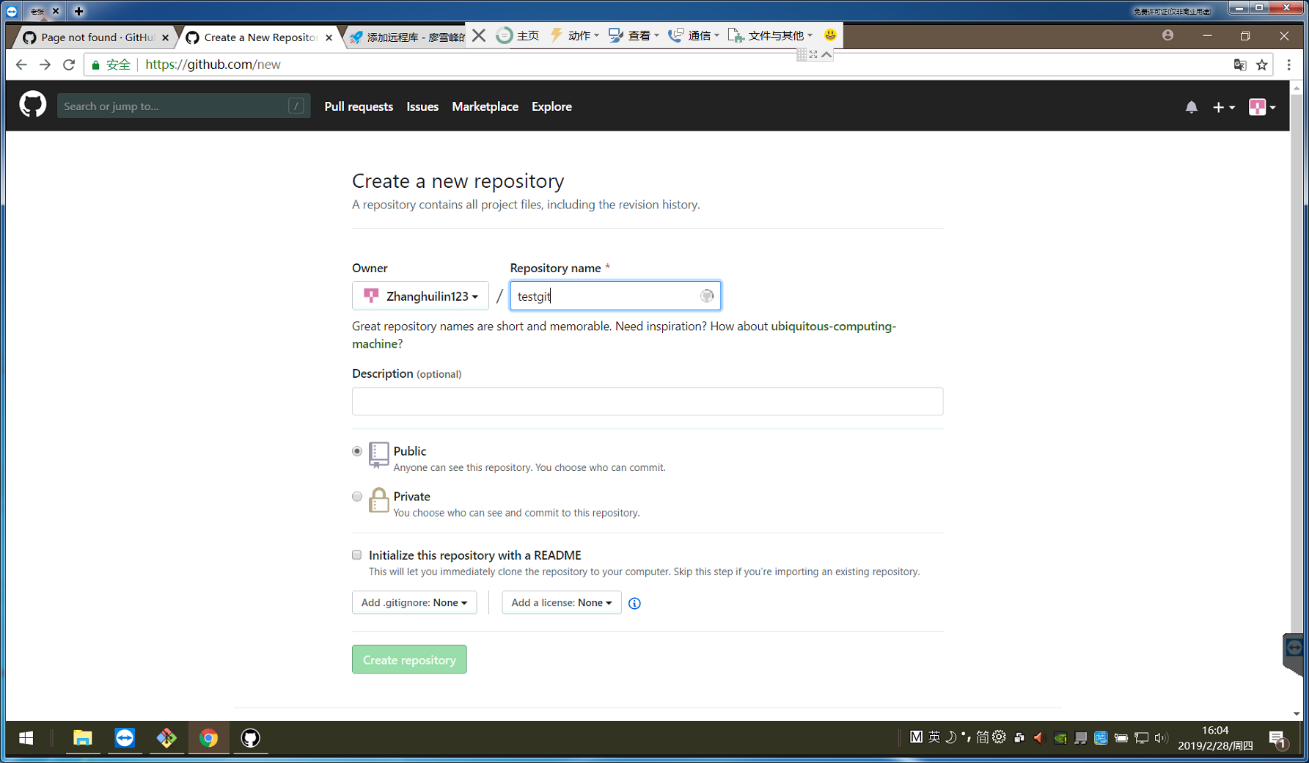
### (1)添加远程库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得

1.首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：

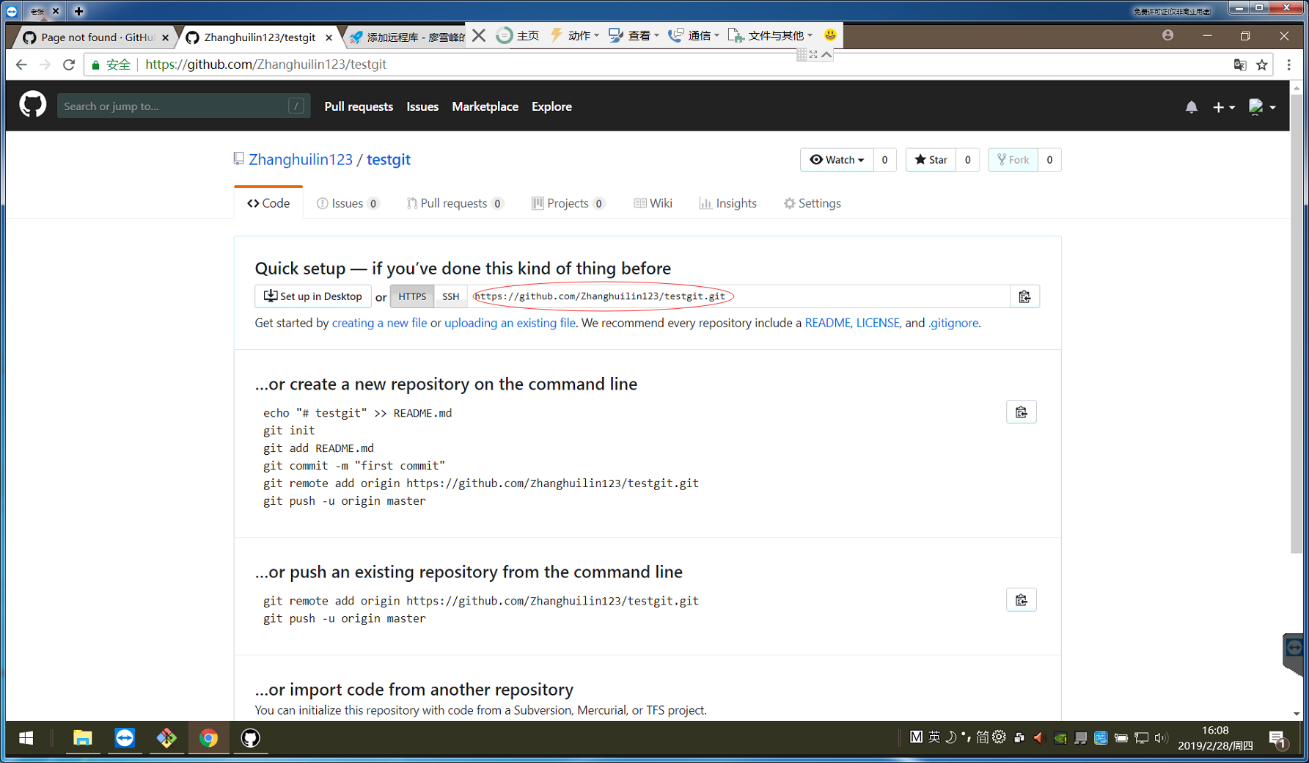


2.在Repository name填入testgit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



3.目前，在GitHub上的这个testgit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的testgit仓库下运行命令：



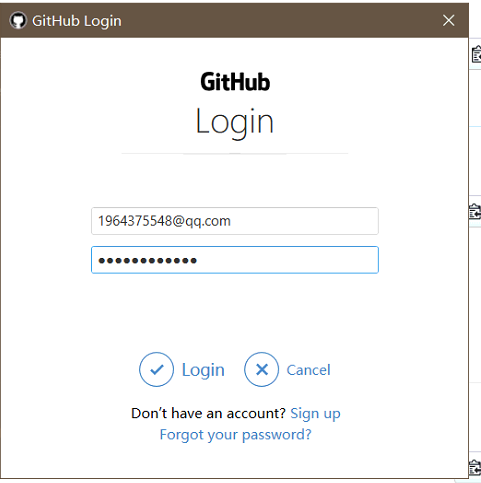
命令如下，即上图画圈部分地址（每个人的均不一样）



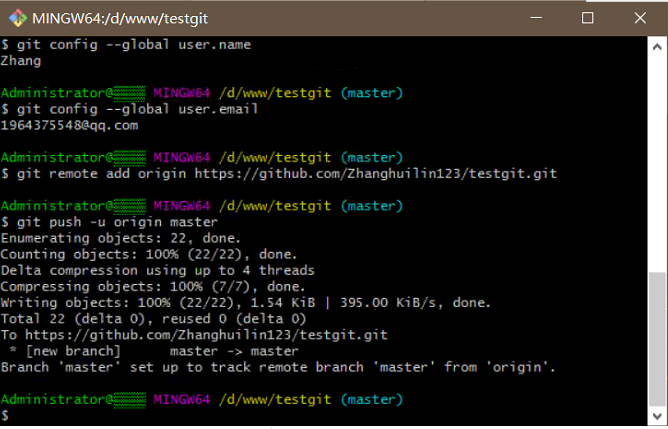
然后输入



之后会出现



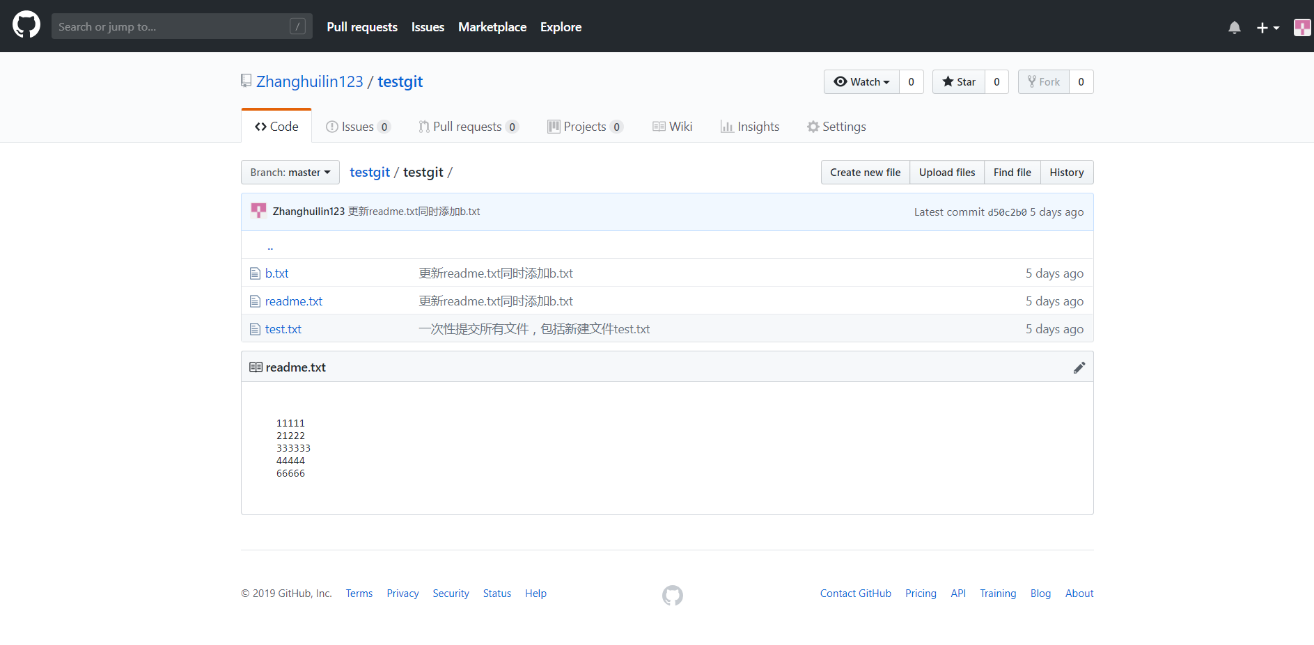
输入账号密码



4.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：



从现在起，只要本地作了提交，就可以通过命令：

git push origin master

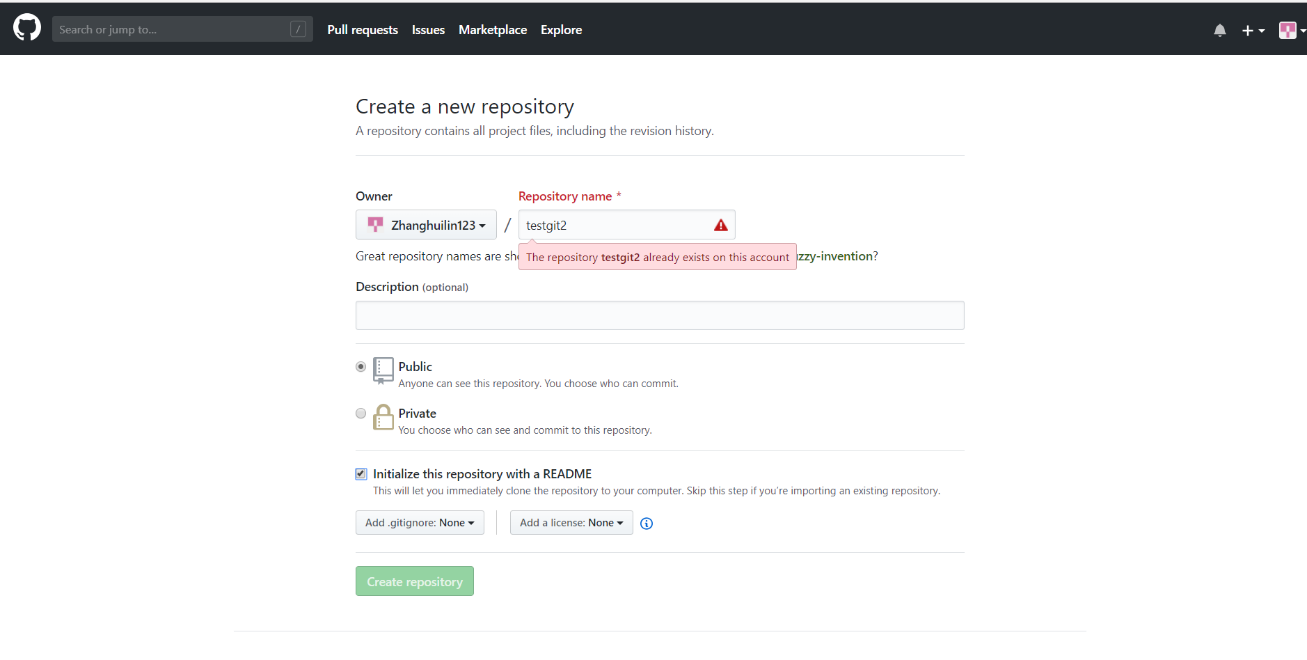
把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

### (2)从远程库克隆

我们上面已经知道先有本地库，后有远程库的时候，如何关联远程库。

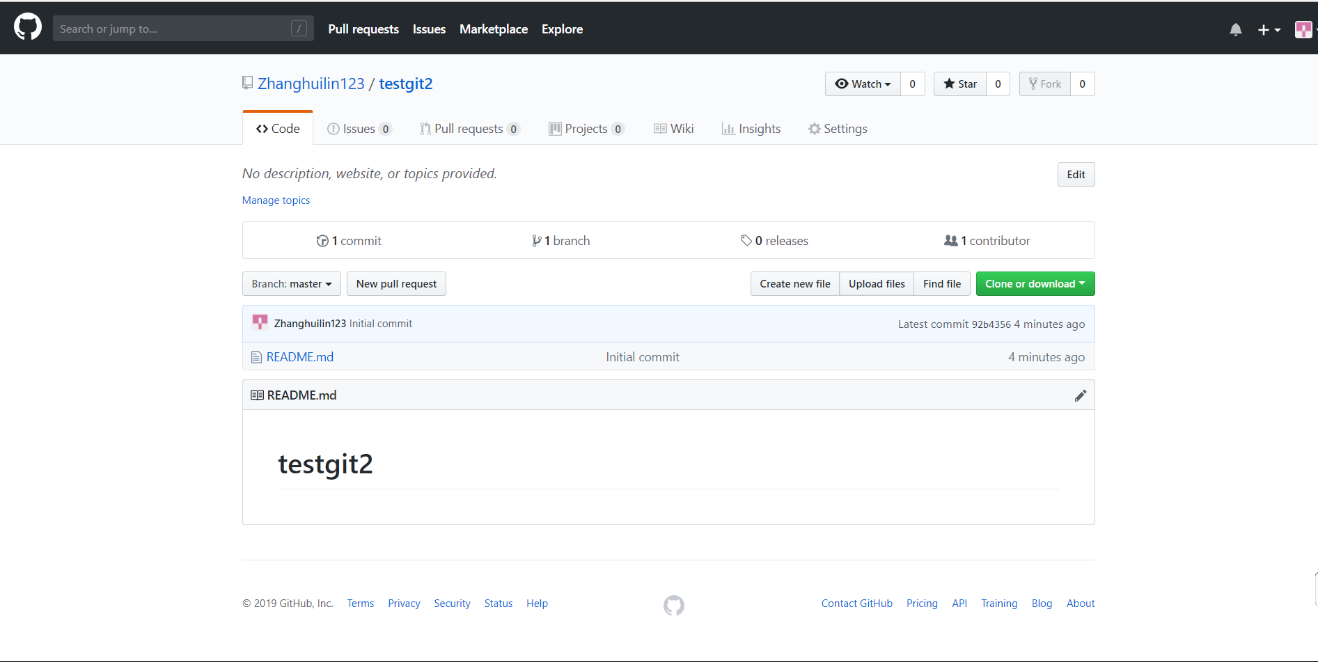
1.现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登陆GitHub，创建一个新的仓库，名字叫testgit2：



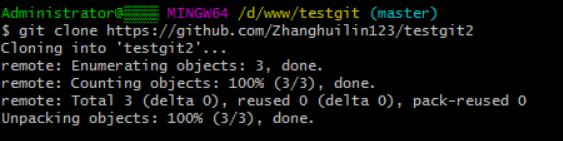
(注意因为我之前已经创建了testgit2, 所以github提示我已经存在testgit2文件)

2.我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：

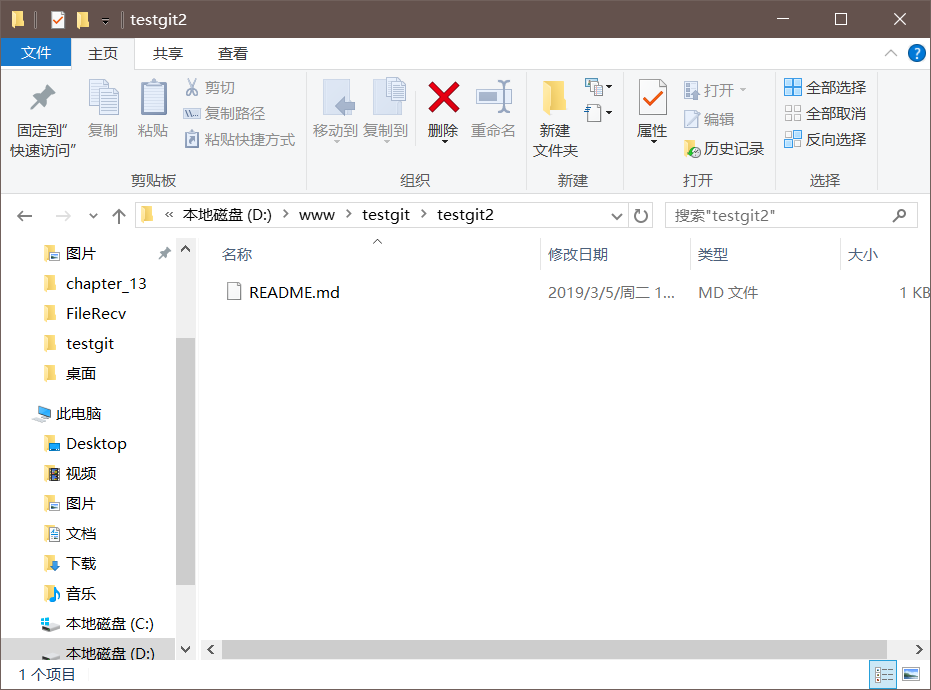


3.从远程库克隆

现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：



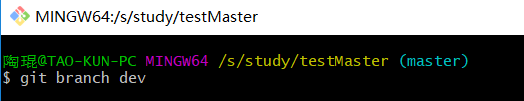
4.去本地目录下找到生成的testgit 2文件，就说明克隆成功了



# 分支管理

### (1)创建与合并分支

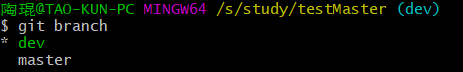
1.首先创建一个名叫dev分支git branch dev



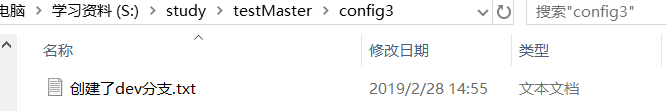
2.切换到dev分支 git checkout dev



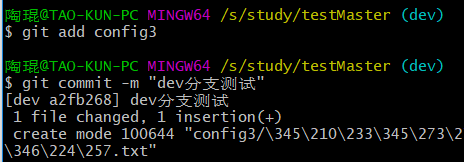
3.查看当前分支 git branch



4.创建config3并在其下创建一个txt文件



5.提交 git add config3

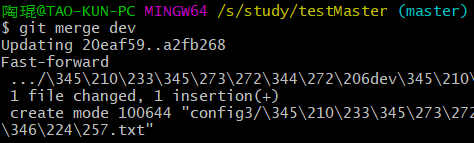


6．dev分支上工作完成，切回master分支，查看config3是否存在，不存在！





7.将dev分支的工作成果合并到master上



成功！



8.删除dev分支



成功！

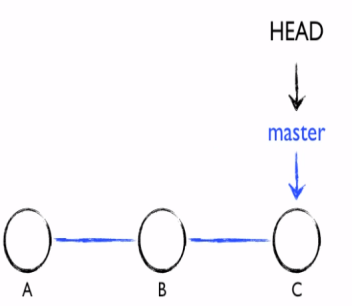
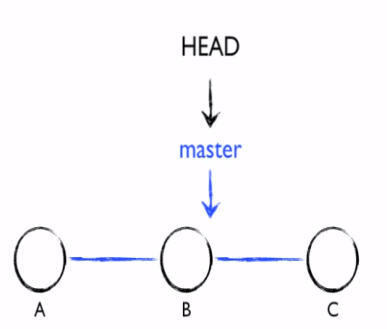
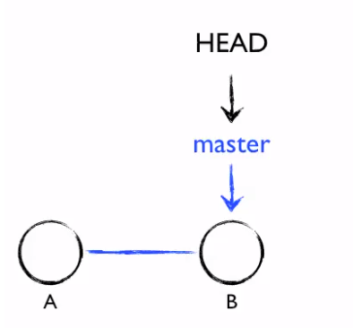


#### 5.1.1创建与合并分支总结

1. 一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



1. 每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：



1. 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：



1. 当给dev分支进行新的提交后dev指针向前移了一步，而master没有移动



1. 当完成dev分支上的操作，需要将dev合并到master分支上，这是就是把master指针指向dev分支



1. 合并完成后将dev分支删除

#### 5.1.2分支操作

* 1. 查看分支：git branch
  2. 创建分支：git branch <name>
  3. 切换分支：git checkout <name>
  4. 创建+切换分支：git checkout -b <name>
  5. 合并某分支到当前分支：git merge <name>
  6. 删除分支：git branch -d <name>

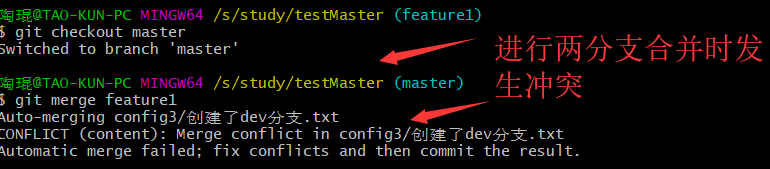
### (2)解决冲突

（1）新建feature1分支，修改config 中的txt文件

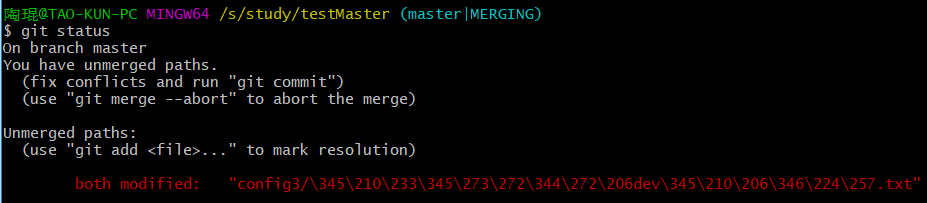


（2）切换到master分支

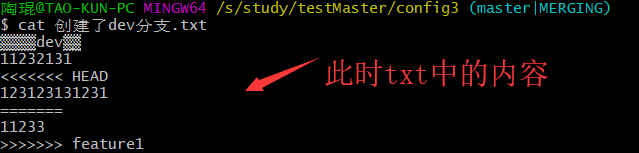
（3）在master分支上把txt文件的修改



（4）文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：



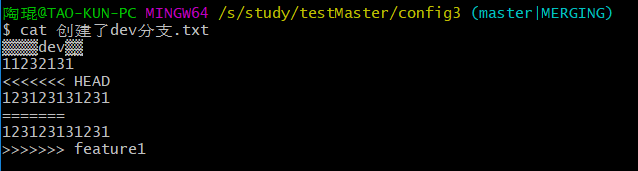
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容



现在，master分支和feature1分支各自都分别有新的提交，变成了这样



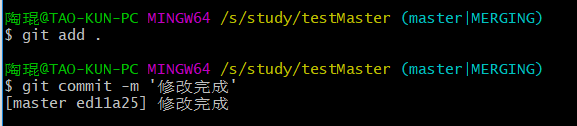
修改至内容一致



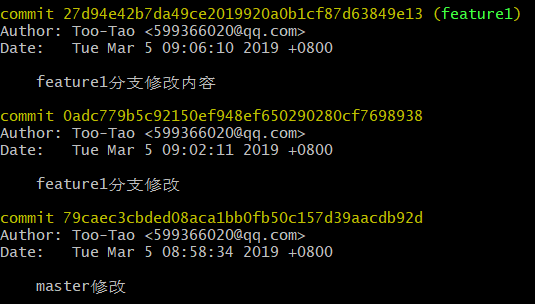
现在，master分支和feature1分支变成了下图所示：



提交



用带参数的git log也可以看到分支的合并情况



冲突解决，最后，删除feature1分支：

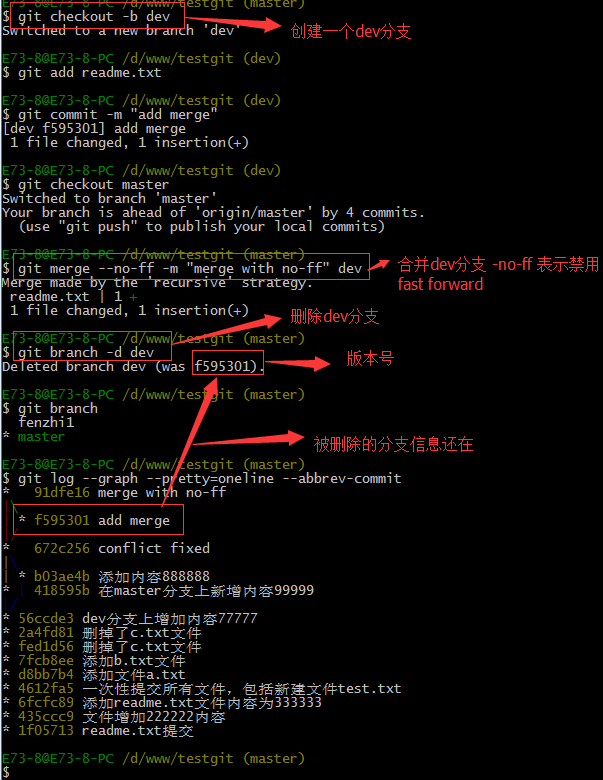


### (3) 分支管理策略。

通常合并分支时，git一般使用”Fast forward”模式，在这种模式下，删除分支后，会丢掉分支信息，现在我们来使用带参数 –no-ff来禁用”Fast forward”模式。首先我们来做demo演示下：

1. 创建一个dev分支。
2. 修改readme.txt内容。
3. 添加到暂存区。
4. 切换回主分支(master)。
5. 合并dev分支，使用命令 git merge –no-ff  -m “注释” dev
6. 查看历史记录

截图如下：

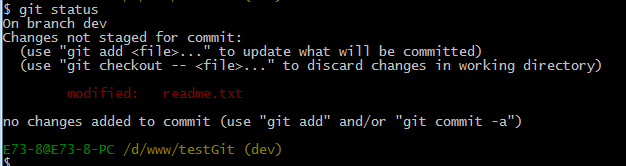


分支策略：首先master主分支应该是非常稳定的，也就是用来发布新版本，一般情况下不允许在上面干活，干活一般情况下在新建的dev分支上干活，干完后，比如上要发布，或者说dev分支代码稳定后可以合并到主分支master上来。

### (4) bug分支

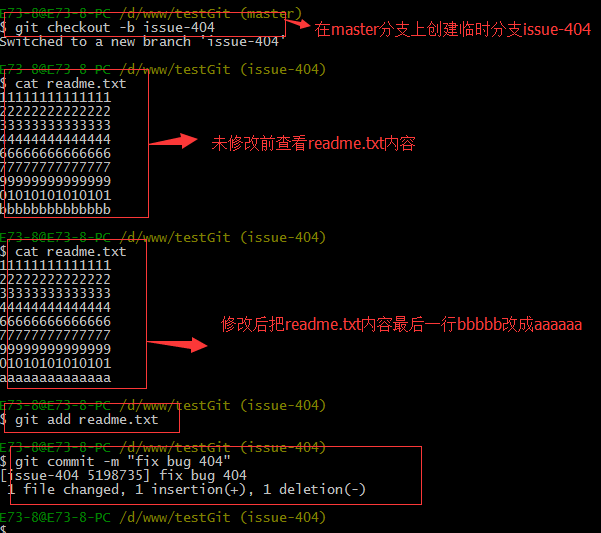
在开发中，会经常碰到bug问题，那么有了bug就需要修复，在Git中，分支是很强大的，每个bug都可以通过一个临时分支来修复，修复完成后，合并分支，然后将临时的分支删除掉。

比如我在开发中接到一个404 bug时候，我们可以创建一个404分支来修复它，但是，当前的dev分支上的工作还没有提交。比如如下：

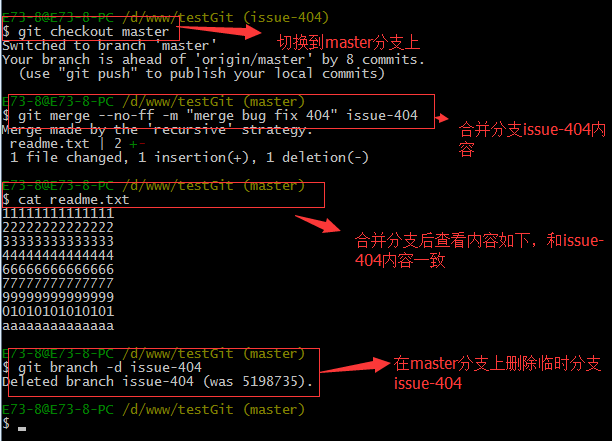


所以现在我可以通过创建issue-404分支来修复bug了。

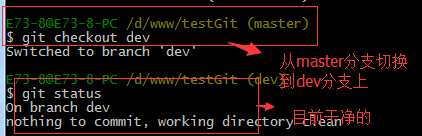
首先我们要确定在那个分支上修复bug，比如我现在是在主分支master上来修复的，现在我要在master分支上创建一个临时分支，演示如下：



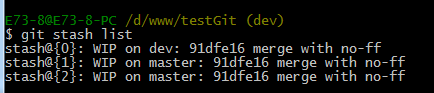
修复完成后，切换到master分支上，并完成合并，最后删除issue-404分支。演示如下：



现在，我们回到dev分支上干活了。



工作区是干净的，那么我们工作现场去哪里呢？我们可以使用命令 git stash list来查看下。如下：

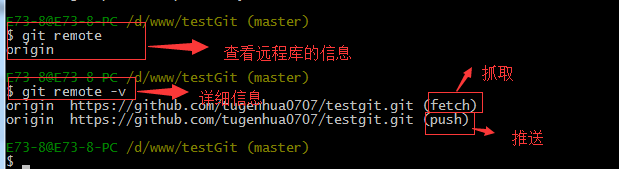


### (5)多人协作

当你从远程库克隆时候，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，并且远程库的默认名称是origin。

1. 要查看远程库的信息 使用 git remote
2. 要查看远程库的详细信息 使用 git remote –v

如下演示：



#### 5.5.1推送分支

推送分支就是把该分支上所有本地提交到远程库中，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：

使用命令 git push origin master

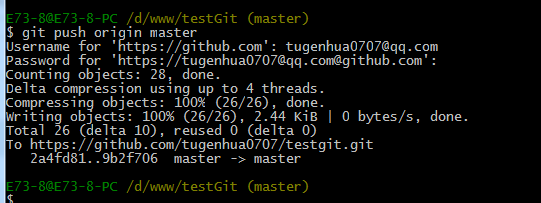
比如我现在的github上的readme.txt代码如下：



本地的readme.txt代码如下：



现在我想把本地更新的readme.txt代码推送到远程库中，使用命令如下：



我们可以看到如上，推送成功，我们可以继续来截图github上的readme.txt内容 如下：



可以看到 推送成功了，如果我们现在要推送到其他分支，比如dev分支上，我们还是那个命令 git push origin dev

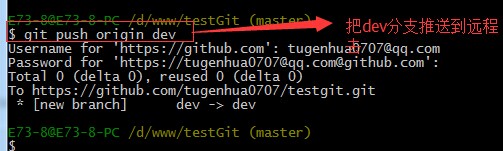
那么一般情况下，那些分支要推送呢？

1. master分支是主分支，因此要时刻与远程同步。
2. 一些修复bug分支不需要推送到远程去，可以先合并到主分支上，然后把主分支master推送到远程去。

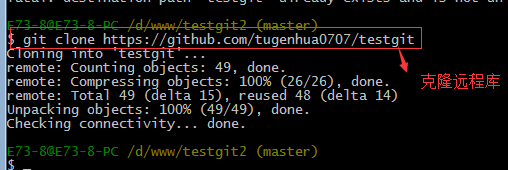
#### 5.5.2抓取分支

多人协作时，大家都会往master分支上推送各自的修改。现在我们可以模拟另外一个同事，可以在另一台电脑上（注意要把SSH key添加到github上）或者同一台电脑上另外一个目录克隆，新建一个目录名字叫testgit2

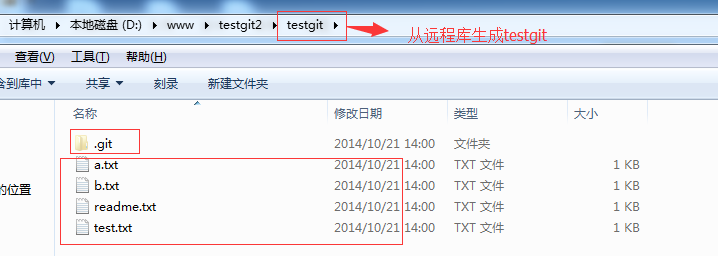
但是我首先要把dev分支也要推送到远程去，如下



接着进入testgit2目录，进行克隆远程的库到本地来，如下：



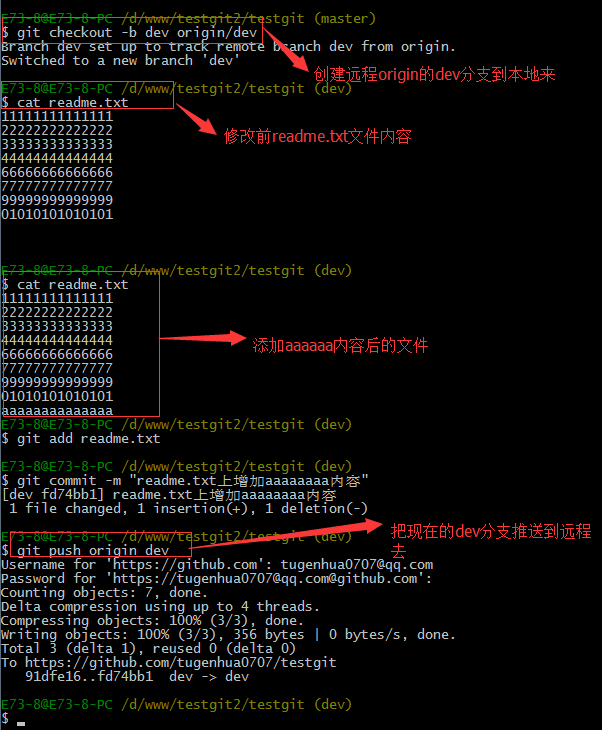
现在目录下生成有如下所示：



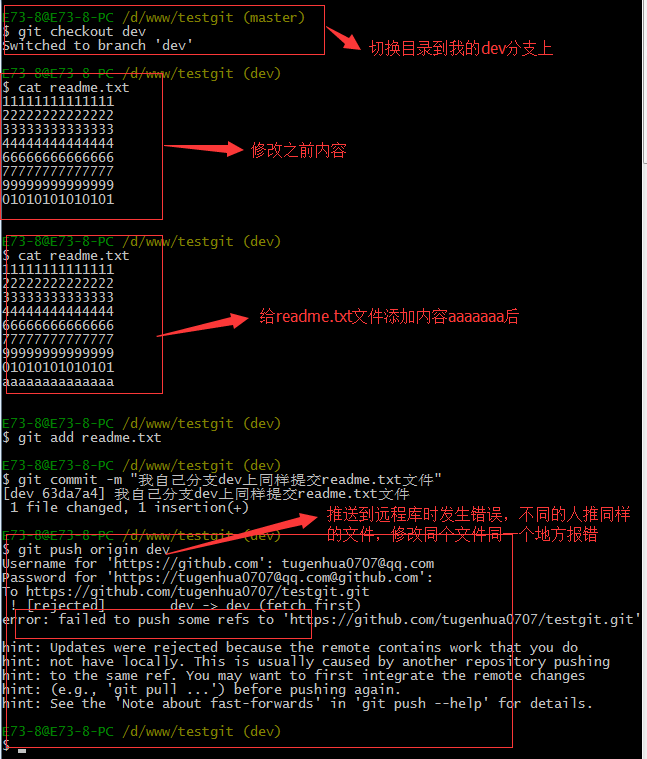
现在我们的小伙伴要在dev分支上做开发，就必须把远程的origin的dev分支到本地来，于是可以使用命令创建本地dev分支：git checkout  –b dev origin/dev

现在小伙伴们就可以在dev分支上做开发了，开发完成后把dev分支推送到远程库时。

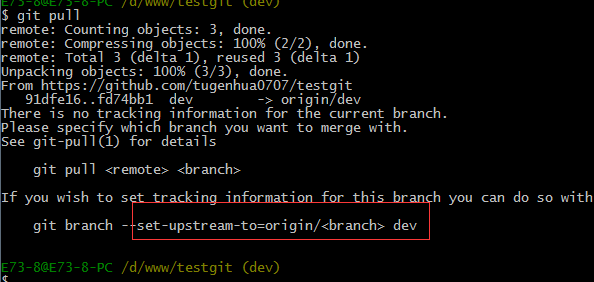
如下：



小伙伴们已经向origin/dev分支上推送了提交，而我在我的目录文件下也对同样的文件同个地方作了修改，也试图推送到远程库时，如下：



由上面可知：推送失败，因为[我的小伙伴](https://www.baidu.com/s?wd=%E6%88%91%E7%9A%84%E5%B0%8F%E4%BC%99%E4%BC%B4&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)最新提交的和我试图推送的有冲突，解决的办法也很简单，上面已经提示我们，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。

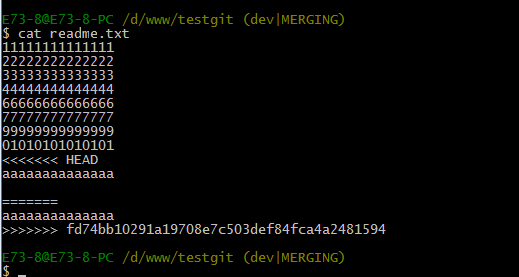


git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：如下

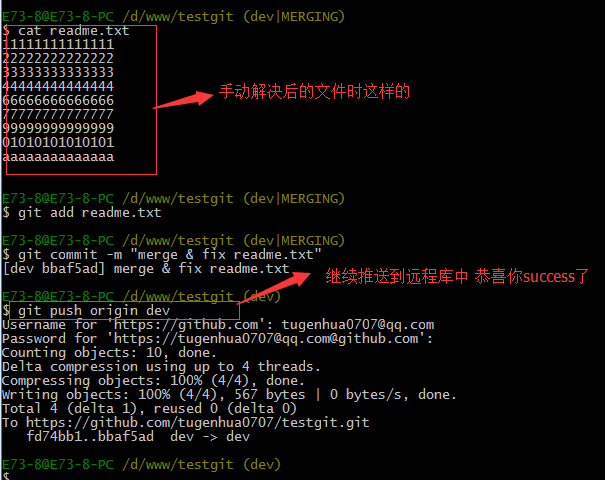


这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的 解决冲突完全一样。解决后，提交，再push：

我们可以先来看看readme.txt内容了。



现在手动已经解决完了，我接在需要再提交，再push到远程库里面去。如下所示：



因此：多人协作工作模式一般是这样的：

1. 首先，可以试图用git push origin branch-name推送自己的修改.
2. 如果推送失败，则因为远程分支比你的本地更新早，需要先用git pull试图合并。
3. 如果合并有冲突，则需要解决冲突，并在本地提交。再用git push origin branch-name推送。