GitHub学习

# 一.创建版本库（仓库，文件夹下所有文件都能被Git管理）

在Git Bash创建（小黑框）

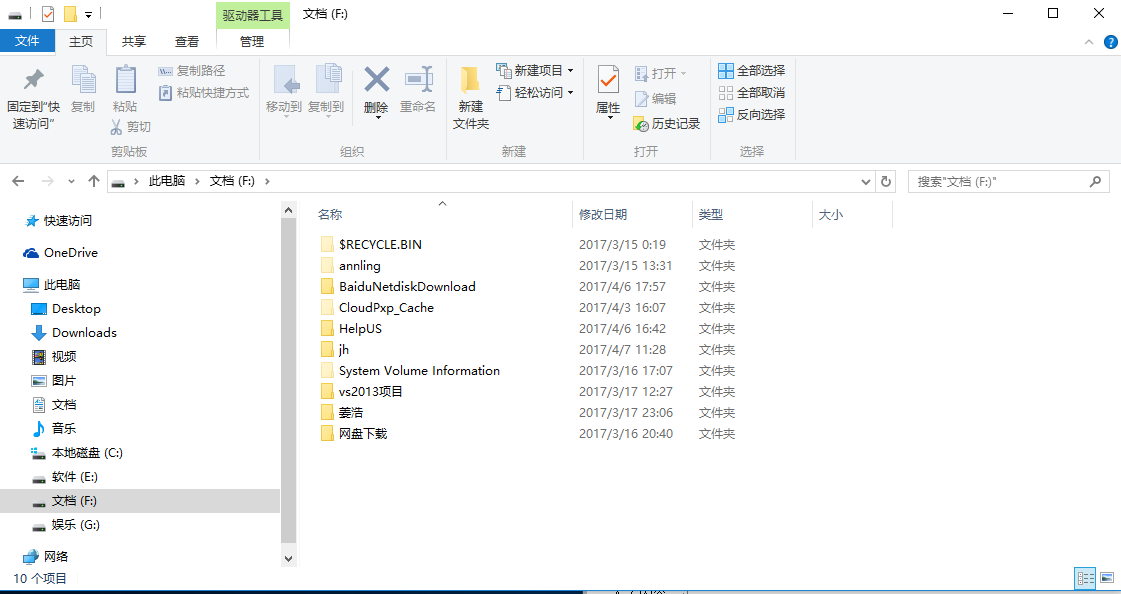
## 1.创建目录

cd + 目标相对路径---->指定所在目录路径

mkdir + 目录名 ---->创建目录

pwd ---->查看所在目录

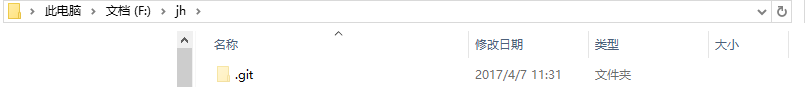




## 2. 通过git init命令把这个目录变成Git可以管理的仓库

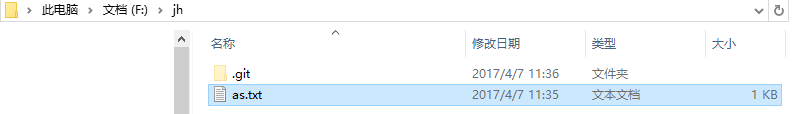


查看文件夹，多一个.git目录（是Git来跟踪管理版本库的，不能改）

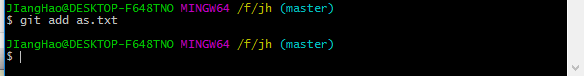


这个目录是隐藏的，如果没有，用 ls –ah查看（如果还没有电脑设置显示隐藏文件夹）

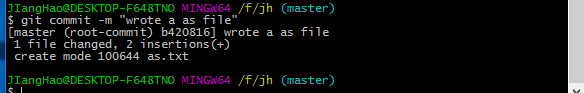
## 3.把文件添加到版本库（目的是记录文件的每一次改动）

编写一个.txt文件（随便写内容），放在指定目录

用git add命令告诉Git，把文件添加到了仓库



用命令git commit命令告诉Git，把文件提交到仓库



-m ” ”是本次说明注释

git commit命令执行成功后会告诉你，1个文件被改动（我们新添加了.txt文件），插入了内容。

为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

$ git add file1.txt

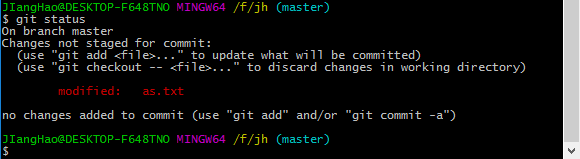
$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

# 二．查看改动信息

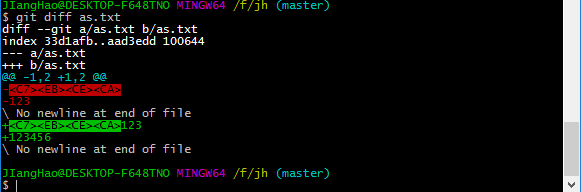
已经成功添加并提交一个.txt文件，现在修改.txt文件里面内容

运行git status命令查看结果



git status命令掌握当前仓库状态，红色行表示.txt已经改动，但还没准备提交修改，不能看修改的内容

运行git diff命令查看修改内容

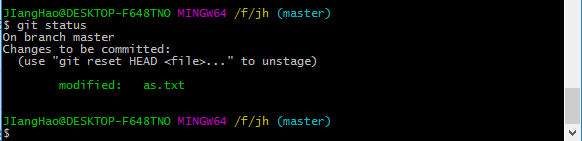


红色行表示被修改的内容，绿色行表示修改后的内容。知道对文件的修改，就可以放心提交到仓库了，提交修改和提交新文件一样两步

用git add命令

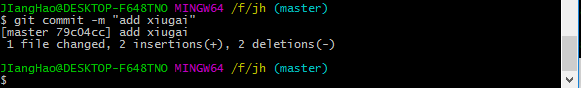


没有任何输出，在执行git commit之前，运行git status查看当前仓库状态

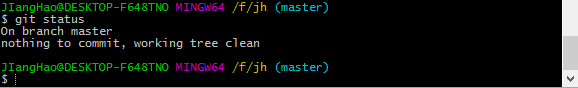


git status告诉我们，将要被提交的修改包括.txt文件（绿色行），下一步，我们就可以放心提交了

git commit命令提交

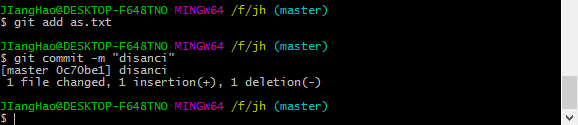


提交后，再用git status命令查看仓库当前状态，Git告诉我们当前没有需要提交的修改，而且工作目录干净（nothing to commit,working tree clean）

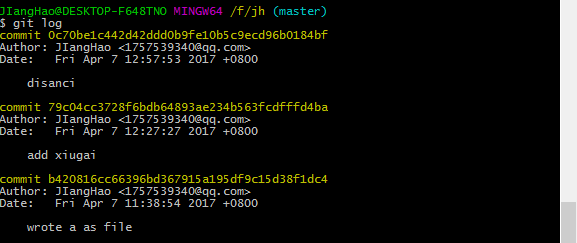


# 三．版本退回

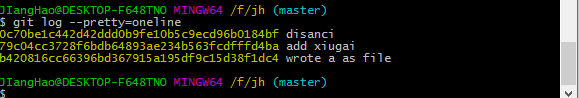
尝试再次修改，目的是用git记录修改，一旦把文件改乱或者误删，可以从最近一个commit恢复，然后尝试提交修改



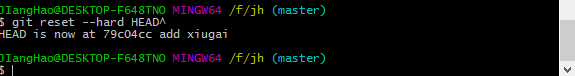
用git log命令查看历史记录，修改的次数



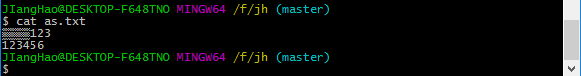
git log命令显示从最近到最远的提交日志，如果嫌输出信息太多，可以加上—pretty=oneline参数



一大串黄色数字是commit id（版本号） 在Git中，用HEAD表示当前版本（最新版本），上一版本就是HEAD^,上上一本版本就是HEAD^^,当然往上100个版本写100个^不现实，所以写成HEAD~100，现在把当前版本回退到上一个版本，使用git reset命令



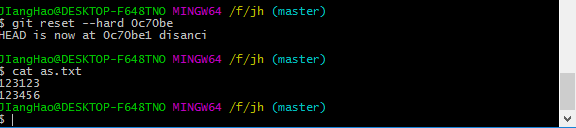
现在看看.txt的内容是不是上一个版本 用cat命令



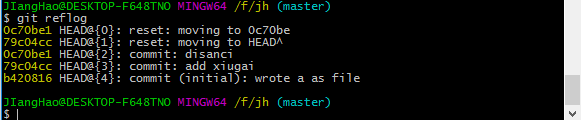
现在我们用git log再看看现在版本库的状态



现在最新的版本已经看不到了，如果想回去，只要上面的命令行窗口还没有被关掉，找到那个版本的commit id（黄色的行 版本号没必要写全，前几位就可以了）再查看as.txt最新版本内容就可以了



现在，你回退到了某个版本，关掉了电脑，如果想恢复，找不到最新的commit id，Git提供了一个命令，git reflog用来记录每一次命令



显示了每次操作的commit id，可以通过版本号id找回来

# 四．工作区和暂存区

工作区（working directory）: 就是你在电脑里能看到的目录，比如新建的文件夹就是一个工作区：

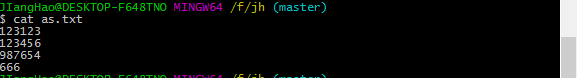
版本库（repository）：工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

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。

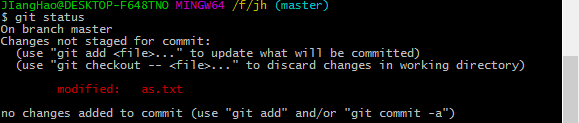
# 五．管理修改（Git跟踪并管理的是修改，而非文件）

# 六．撤销修改

现在我们在文件夹里加一个内容，然后发现错误，需要改正：用cat查看当前版本内容

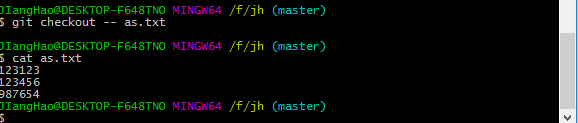


要改正这个错误，删掉最后一行，手动把文件恢复到上一个版本的状态，用git status查看当前状态：



图片内容里有一行显示的是，git checkout -- file可以放弃工作区的修改：（--很重要，没有—就变成了“切换到另一个分支”的命令）

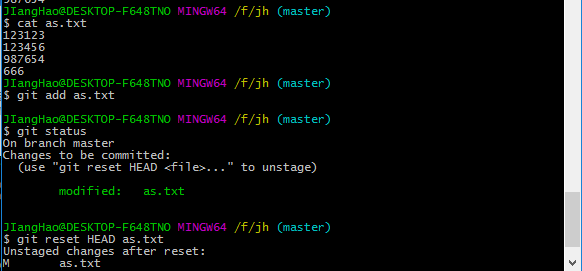
修改后再用cat命令查看当前的.txt文件内容



现在改回去，以前的错误不但写出来，还git add到暂存区了：

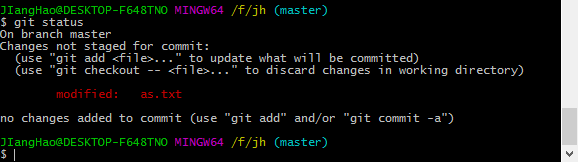
庆幸的是，在commit之前，你发现了这个问题，用git status查看，修改只是添加到了暂存区，还没有提交：

Git（图片）告诉我们，用git reset HEAD file可以把暂存区修改撤销掉（unstage）,重新放回工作区：

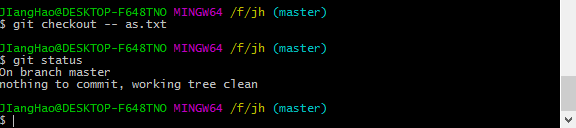


git reset 命令即可回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

再用git status查看一下，现在暂存区是干净的，工作区有修改：

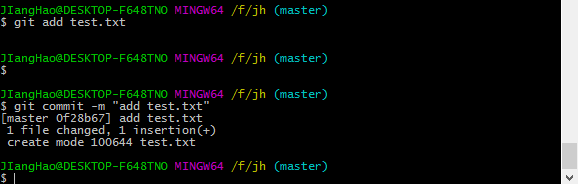


然后丢弃工作区的修改：工作区干净，已改好



# 七．删除文件

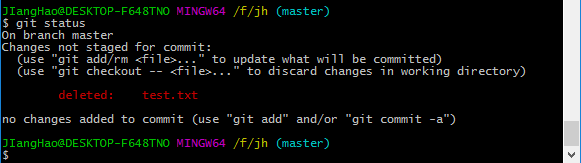
在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：



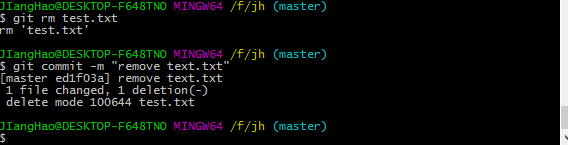
一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：



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：



现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

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”。

# 八．远程仓库

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

$ ssh-keygen -t rsa -C "youremail@example.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，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。

第二步：

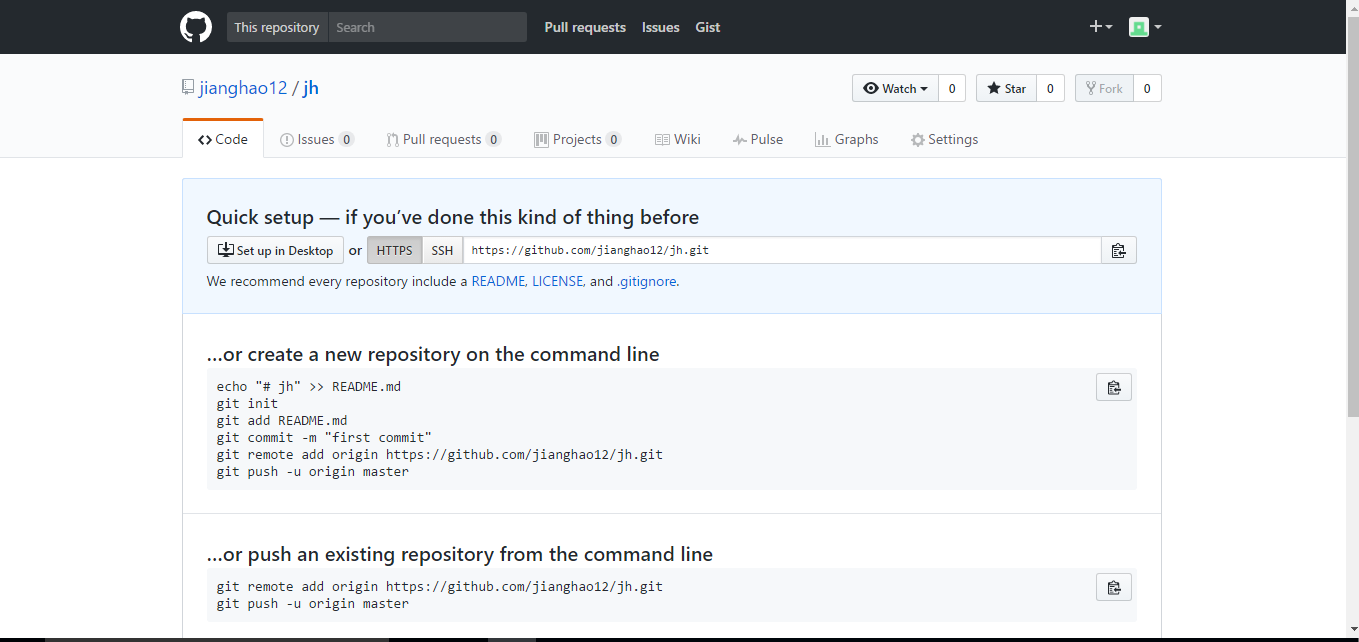
第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：https://github.com/settings

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：

# 九．添加远程库

已经在本地创建了一个git仓库，又想在GitHub创建一个git仓库，并且让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。

首先登陆GitHub，创建一个新的仓库：<https://github.com/new>



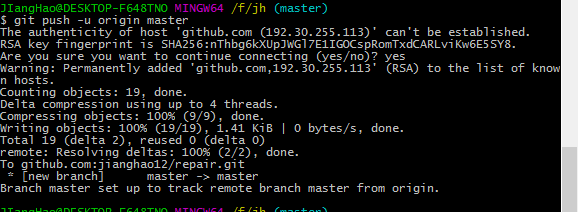
目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

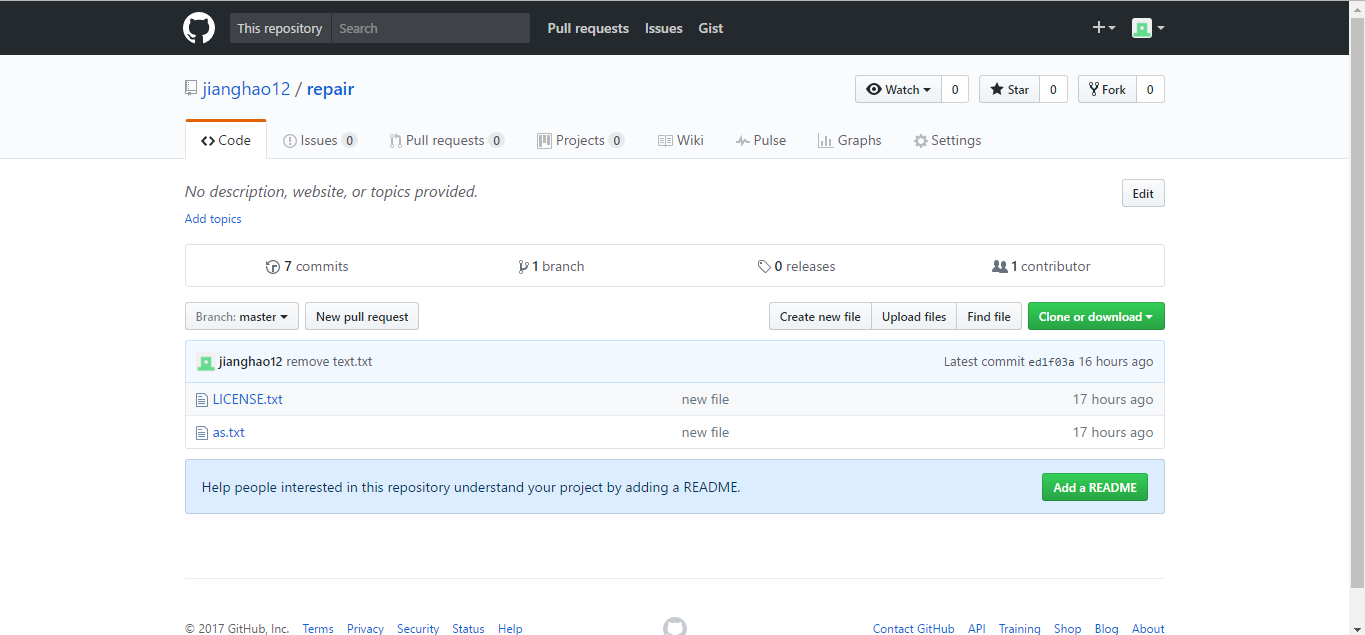
$ git remote add origin [git@github.com:jianghao12/learngit.git](mailto:git@github.com:jianghao12/learngit.git)

把上面的jianghao12替换成自己的GitHub账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

下一步，就可以把本地库的所有内容推送到远程库上：



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：



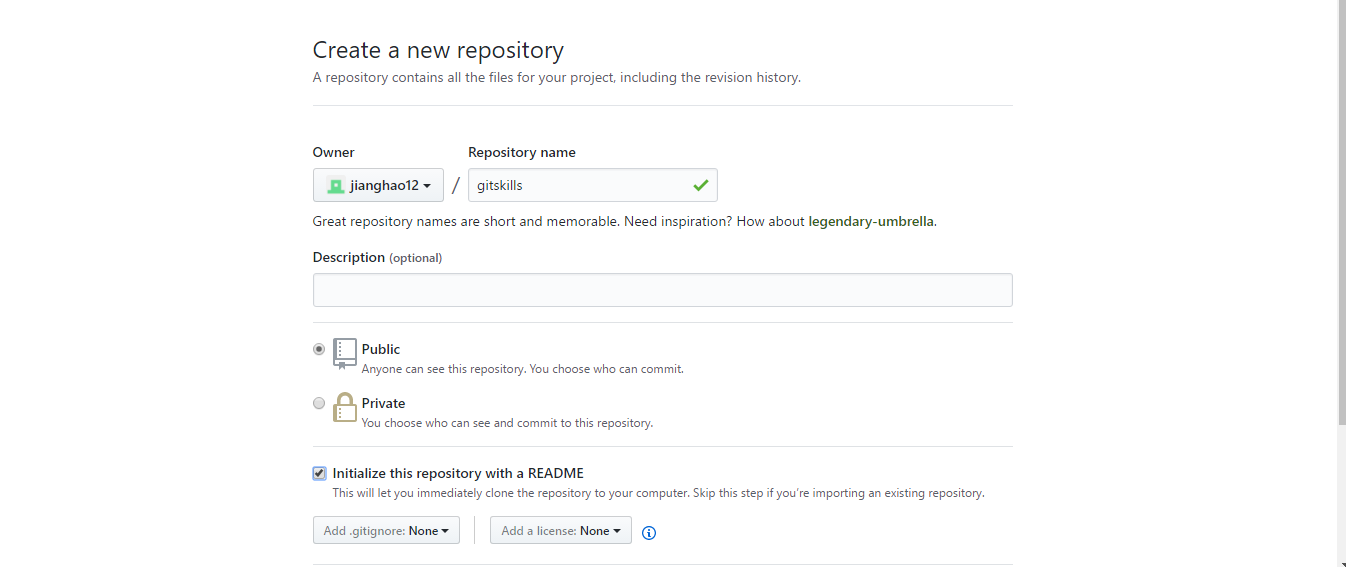
从现在起，只要本地作了提交，就可以通过命令：

$ git push origin master

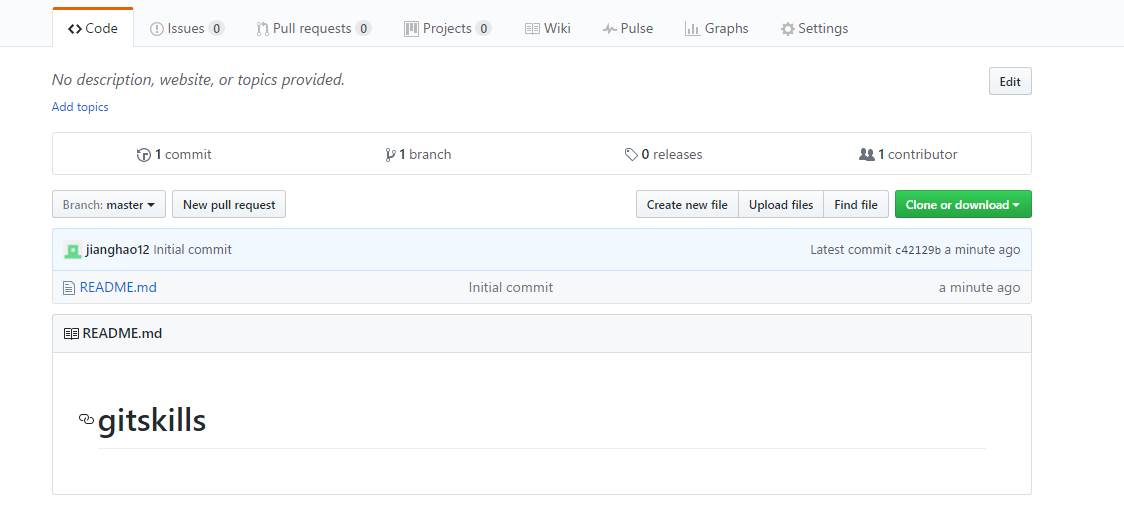
把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

# 十．从远程库克隆

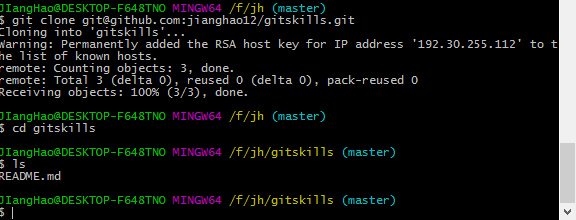
登陆GitHub，创建一个新的仓库，名字叫gitskills:



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：



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：



# 十一．创建与合并分支

在[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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：



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：

#### 创建与合并分支

阅读: 805016

在[版本回退](http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000/0013744142037508cf42e51debf49668810645e02887691000)里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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：



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：

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：



创建Dev分支，然后切换到Dev分支



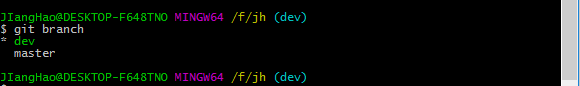
git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

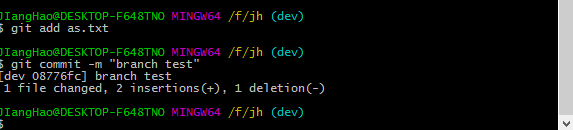
Switched to branch 'dev'

然后用git branch命令查看当前分支：（会列出所有分支，当前分支前面会标一个\*号）



然后，我们就可以在dev分支上正常提交，比如对.txt做个修改，加上一行：

然后提交：



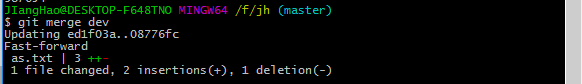
现在，dev分支的工作完成，我们就可以切换回master分支：

切换回master分之后，再查看一个as.txt文件，刚才添加内容没有，因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到mater分支上：

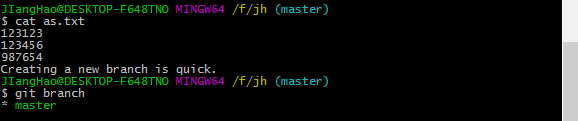
git merge命令用于合并指定分支到当前分支。合并后，再查看as.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。（这次合并是“快进模式”，以后还有其他方式）



合并完成后，就可以放心的删除dev分支了：



删除后，查看branch，就只剩下master分支了：



因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

# 十二.解决冲突

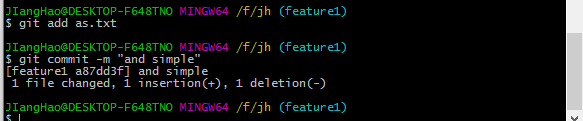
合并分支有时候会出错

准备新的feature1分支，继续我们的新分支开发：

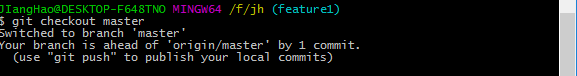


修改as.txt最后一行（随便改）

在feature1分支上提交：



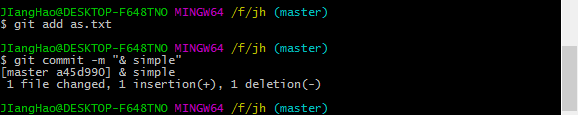
现在切换到master分支：



Git提示我们当前master分支比远程master分支要超前一个提交

在master分支上把as.txt文件的最后修改

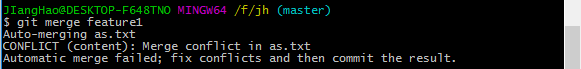
提交：



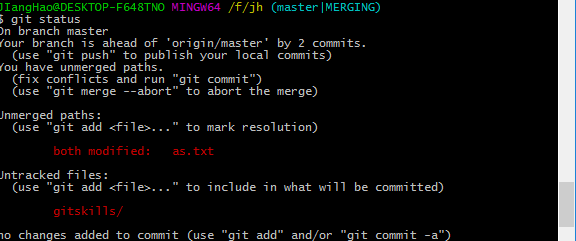
现在，master分支和feature1分支各自都分别有新的提交，变成了这样：



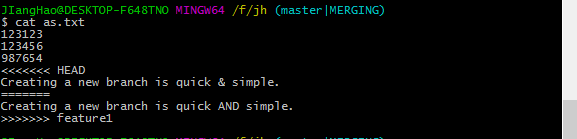
这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：



果然冲突了！Git告诉我们，as.txt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：



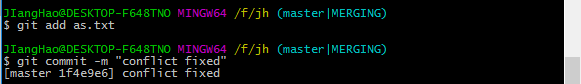
查看as.txt



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

Creating a **new** branch is quick **and** simple.

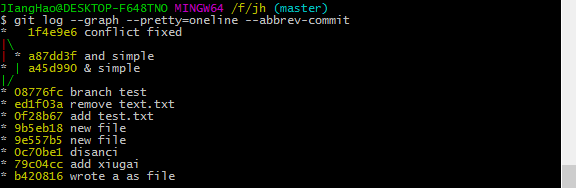
再提交：



现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：



最后，删除feature1分支：



### 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。

# 十三.分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

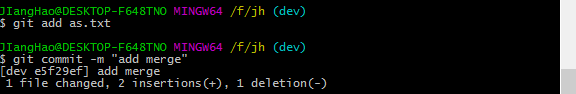
如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

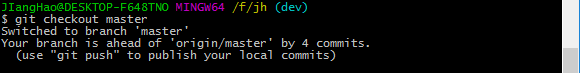
首先，仍然创建并切换dev分支：



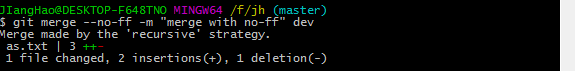
修改as.txt文件，并提交一个新的commit：



切回master：

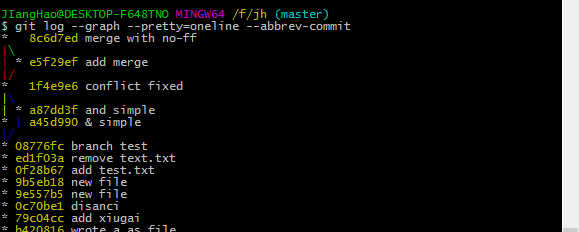


准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：



因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史：



可以看到，不使用Fast forward模式，merge后就像这样：



分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：

