## 一．Git介绍

Git是目前世界上最先进的分布式版本控制器。

版本控制器：就是用来追溯自己书写的代码的记录信息。好处：可以非常方便的记录何时何地何人操作了哪些代码。

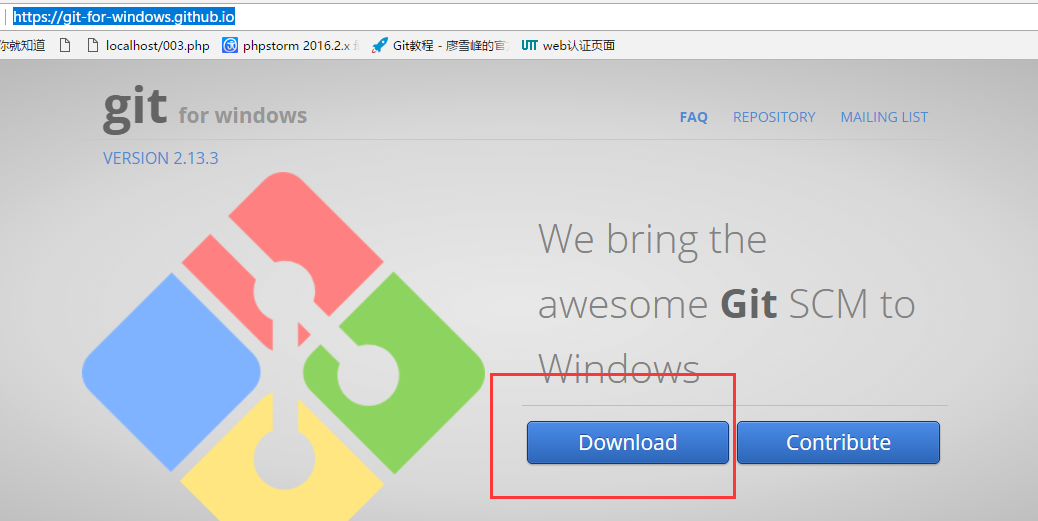
什么是分布式版本控制器？

集中式：对于集中式的版本控制器，需要搭建一个中央服务器，然后在这个中央服务器里面作为代码的仓库。

分布式：就是每个用户的电脑都是一个独立的仓库，可以记录代码的变化，即使不联网，完全也可以自己独立开发。

## 二．Git安装

<https://git-for-windows.github.io/>



安装完后还需要最后一步设置，在命令行输入：告诉你是谁

$ git config --global user.name "Your Name"

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

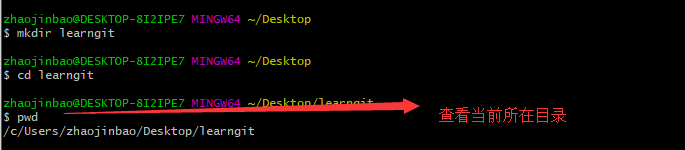
因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。

**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。**

## 三．Git操作

### 1.创建仓库

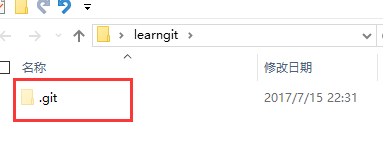
**1).创建仓库目录**



**2).通过git init命令把这个目录变成Git可以管理的仓库**



**注意：Git仓库创建好后，会在该文件存在一个.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。**



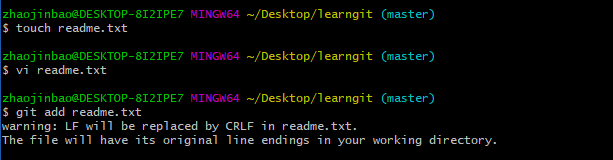
**注意：如果你没有看到.git目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看见。**

**小结：git init 初始化仓库**

### 2.添加文件到版本库

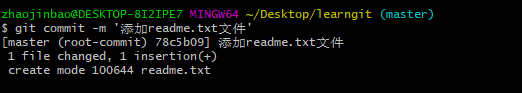
把文件添加到版本分两步，首先将文件添加到暂存区，然后再提交到版本库

**1).创建文件并把文件添加到暂存区（git add）**



**git add . 表示将当前目录下的所有文件都添加到暂存区**

**2).将文件从暂存区提交到版本库**



**-m 表示本次提交的描述**

**小结：**

**git add xxx 添加单个文件到暂存区**

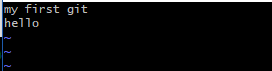
**git add . 将当前文件夹下的所有文件都添加到暂存区**

**git commit -m ‘xxx’ 将文件提交到版本库 -m 表示描述**

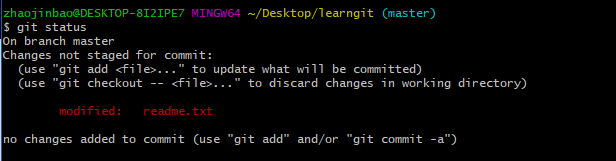
### 3.修改文件查看状态与不同

此时我们将文件内容修改



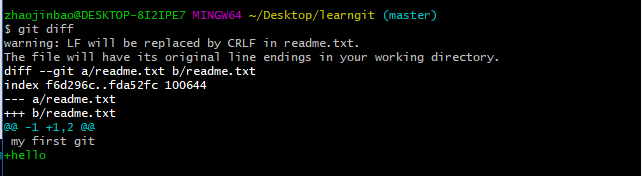


**1).查看状态**



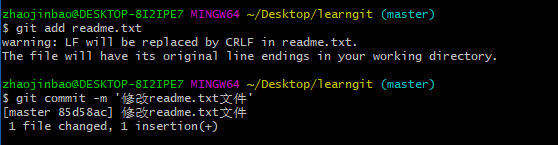
git status 命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

**2).查看此次修改的文件与之前的不同**



git diff顾名思义就是查看difference，可以从上面的命令输出看到，我们添加了一行hello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，提前命令与之前相同。



**小结：**

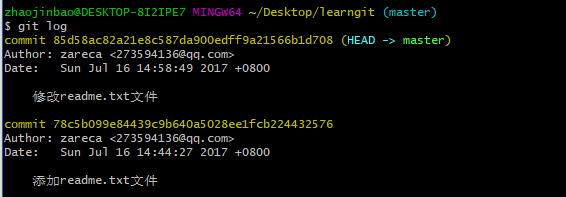
**git status 查看仓库状态**

**git diff 查看修改后文件与之前的不同。**

### 4.版本回退

如果此时我们因为某种原因想回退到之前的某个版本该怎么办？

**1).首先通过git log 查看之前的提交日志**



如果嫌这样看着太乱的话可以试试加上—pretty=oneline参数



其中85d58ac82a21e8c587da900edff9a21566b1d708和78c5b099e84439c9b640a5028ee1fcb224432576 代表的是commit id（版本号），

它是通过SHA1计算出来的一个非常大的数字，用十六进制表示，而且你的commit id 和我的肯定不一样，实际以你自己的为准。

**2).回退到之前某个版本**

知道了之前的提交日志那如何回退到之前的版本呢？

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85d58ac82a21e8c587da900edff9a21566b1d708（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。

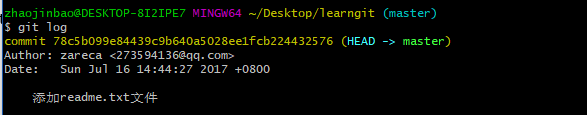
我们现在回退到上一版本使用命令：git reset –hard HEAD^



查看此时的文件：



通过查看文件发现确实是回到了上一个版本，那么我们再通过git log查看此时版本库的状态。

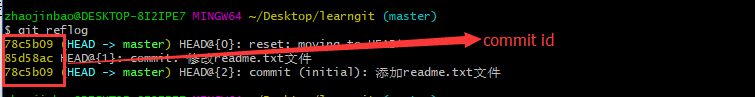


最新的“修改readme.txt文件”这个版本已经不见了，就相当于我们坐了时光机一样，穿越到了之前的版本，

**3).回退到未来某个版本**

现在肯定又有人想我穿越到了过去，那我再怎么穿越回来呢？

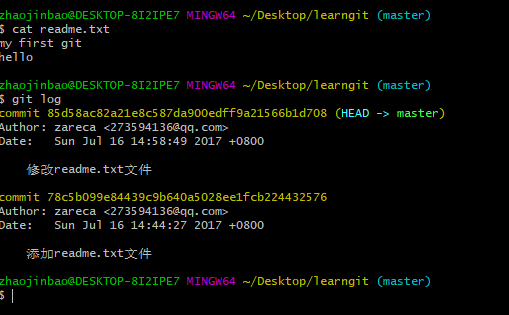
在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到”添加readme.txt文件”版本时，再想恢复到”修改readme.txt文件”，就必须找到”修改readme.txt文件”的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：



然后通过git reset –hard 85d58ac回退到”修改readme.txt文件”版本。



我们再通过查看文件内容，和历史提交记录发现又回到了”修改readme.txt文件”的版本



**小结：**

**git log 查看提交历史记录，以便确认回退到历史哪个版本。**

**git reflog 查看命令历史，以便确定要回到未来哪个版本**

**git reset - - hard commit\_id 版本之间的穿梭回退**

### 5.工作区与暂存区

**1).工作区：**

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



**2).暂存区：**

是逻辑上的一个概念，是看不到的。

**3).版本库：**

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。



分支和HEAD的概念我们以后再讲。

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：

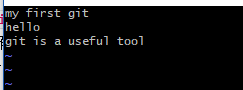
第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

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。

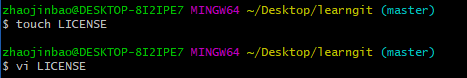
因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

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。

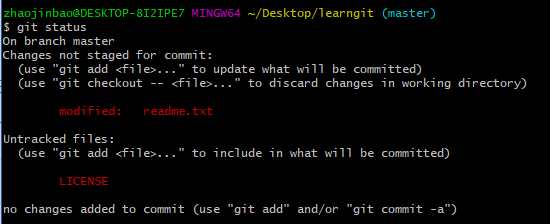
俗话说，实践出真知。现在，我们再练习一遍，先对readme.txt做个修改，比如加上一行内容：



然后，在工作区新增一个LICENSE文本文件（内容随便写）。

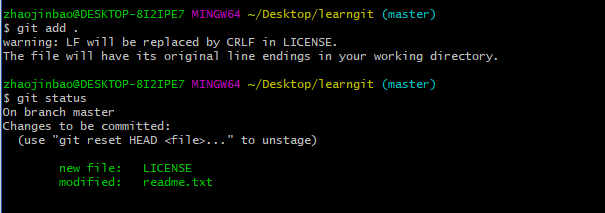


先用git status查看一下状态：



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，所以它的状态是Untracked。

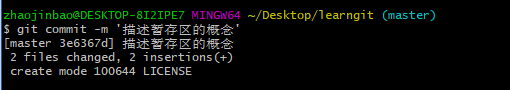
现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：



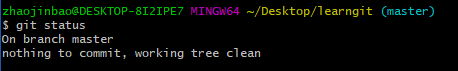
现在，暂存区的状态就变成这样了：



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：



**小结：**

**工作区是指我们本地目录，暂存区是逻辑上的定义，我们想提交某个文件时，需要将工作区的文件放入到暂存区（git add），然后再提交。**

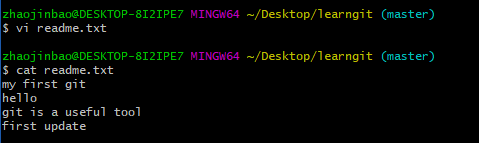
### 6.管理修改

场景：比如说你修改readme.txt文件，并且git add 了，随后你又修改了readme.txt文件，但是此时你没有git add,直接git commit的了，这种情况下，你的第二次修改将不会被提交。

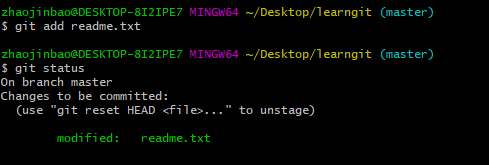
原因是你第二次修改没有放入暂存区，而git commit 是将暂存区的内容提交到版本库的。

下面实际操作一下：

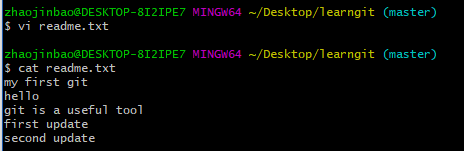
1)第一次修改readme.txt文件：添加一行first update



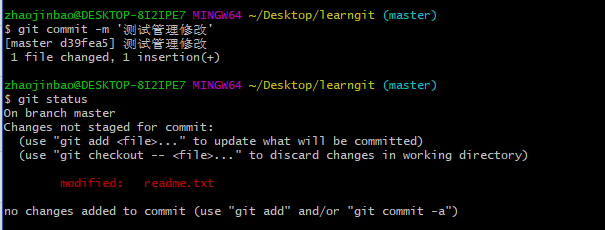
2)git add 此次的修改



3)第二次修改readme.txt文件，添加一行second update

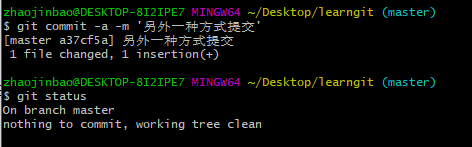


4)直接git commit 提交



发现第二次的修改没有被提交。验证了上面的结论，只有在暂存区的数据才能被提交。所以要想被提交必须先将文件放入到暂存区，然后再提交

我们在修改了某些文件时，也可以直接使用git commit -a -m ‘xxxx’ 提交。-a就相当于git add .操作,将当前目录下所有修改了的文件放入暂存区。



**小结：**

**在暂存区的数据才能被提交到版本库。**

**修改了某些文件时，也可以直接使用git commit -a -m ‘xxxx’ 提交**

### 7.撤销修改

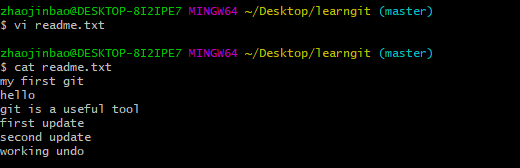
场景1：修改了工作区的某个文件内容如何丢弃修改？

场景2：不但修改了工作区的某个文件内容，而且还添加到了暂存区，如何丢弃修改？

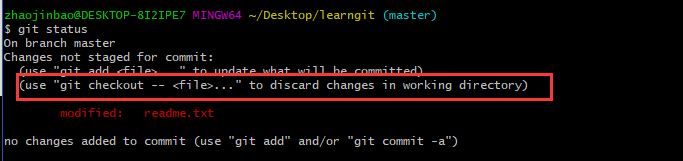
场景3：不但添加到了暂存区而且还提交到了版本库，如何丢弃修改？

**场景1：修改了工作区的某个文件内容如何丢弃修改？**

1.修改readme.txt文件 添加一行working undo

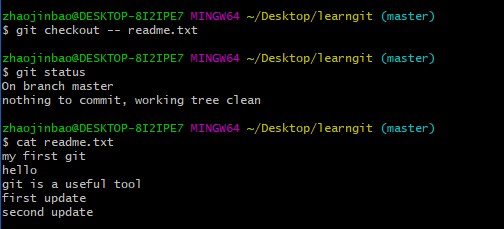


2.git status 查看状态



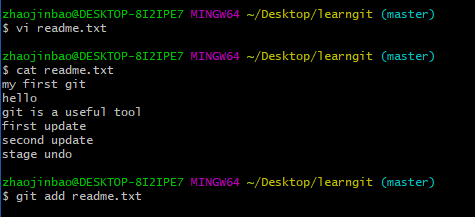
你可以发现git会提示你，git checkout – file 可以丢弃工作区的修改

3.git checkout – file 丢弃修改

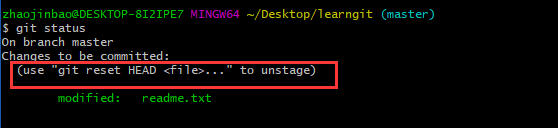


**场景2：不但修改了工作区的某个文件内容，而且还添加到了暂存区，如何丢弃修改？**

1.修改readme.txt文件 添加一行stage undo

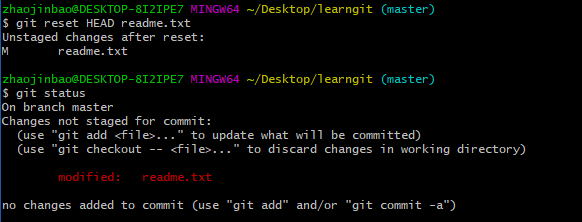


2.git status查看状态



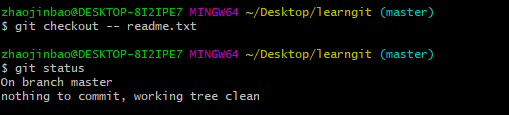
Git同样告诉我们，用命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区.

3.git reset HEAD file 撤销暂存区的修改，重放回工作区。



此时修改的文件已经被放回到工作区。

4.从工作区中撤销本次修改

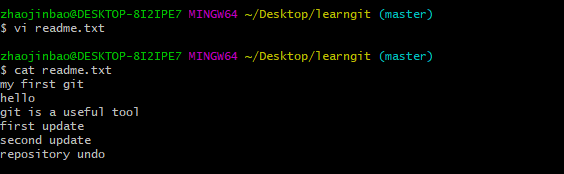


此时修改内容已被丢弃。

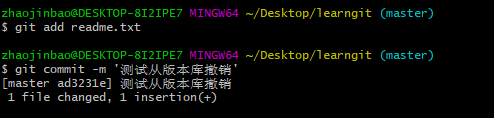
**场景3：不但添加到了暂存区而且还提交到了版本库，如何丢弃修改？**

这种情况就需要参考**版本回退**这一节了，回退到上一个版本

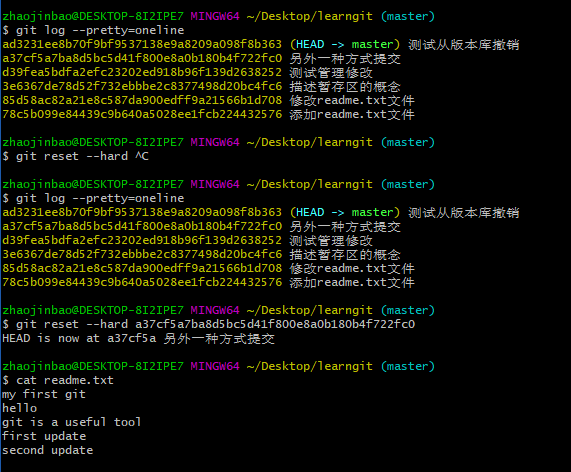
1. 修改readme.txt文件，添加一行repository undo



1. 提交到版本库



1. 回退到上一个版本



**小结：**

**1.对于只在工作区修改了文件内容，撤销此次修改使用 git checkout – file撤销**

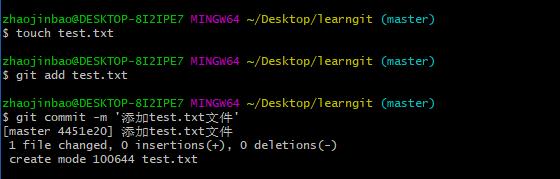
**2.对于提交到了暂存区的文件内容修改，撤销此次修改需要分两步，第一步:git reset HEAD file 重放回工作区，第二步：git checkout – file 从工作区中撤销。**

**3.对于已经提交到版本库的文件内容修改，撤销需要回退到上一个版本，使用git reset –hard commit\_id.**

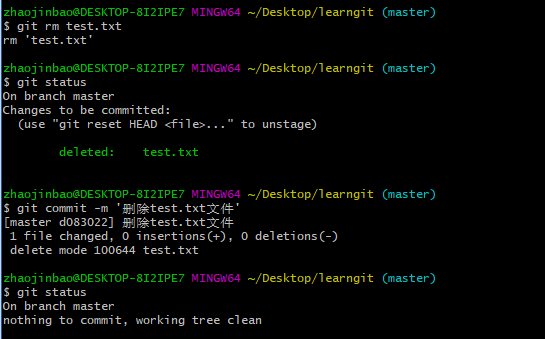
### 8.删除文件

如何将已经提交到版本库的文件删除？

1. 添加文件到版本库



1. 从版本库中删除



**小结：**

**将文件从版本库删除也是分为两步：**

**第一步：git rm file 从版本库删除**

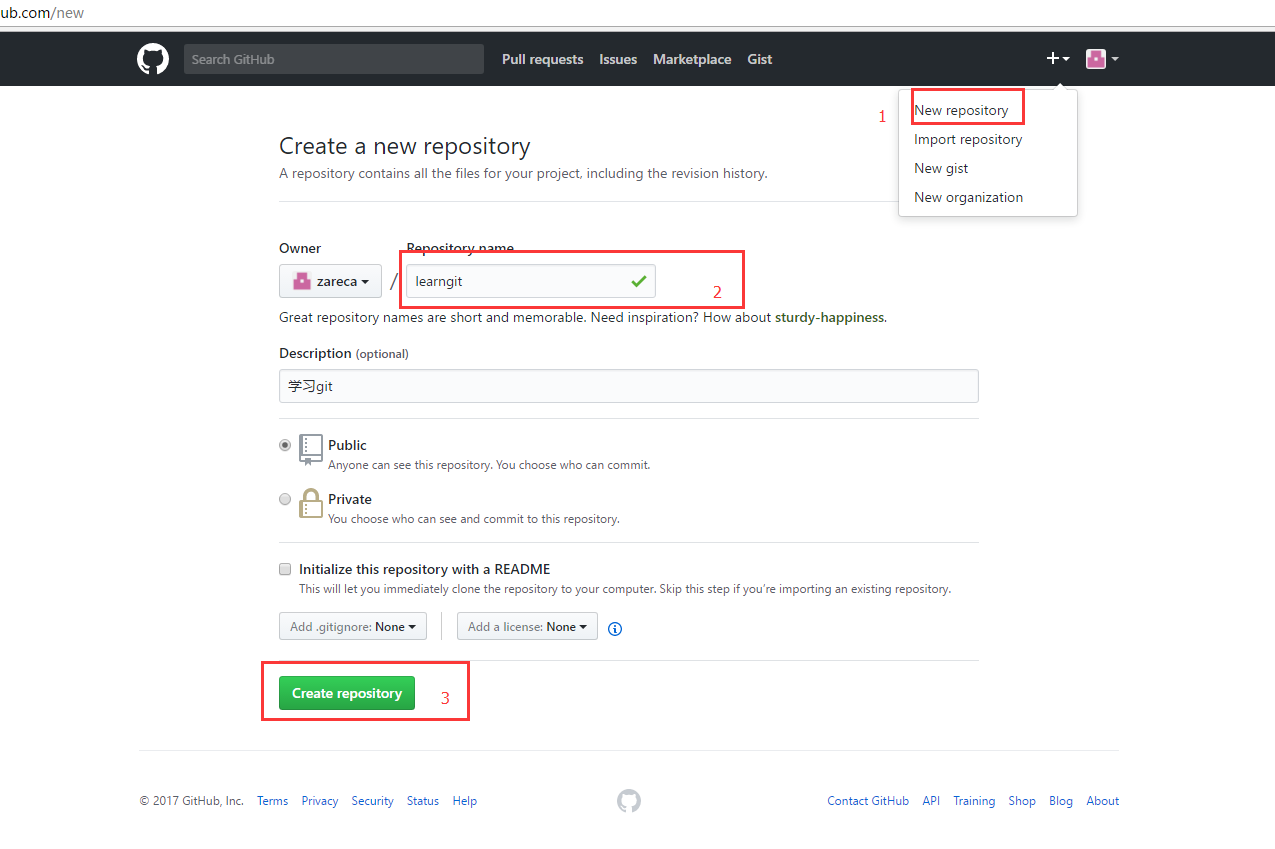
**第二步：git commit -m ‘xxx’ 提交本次的操作**

## 四．远程仓库

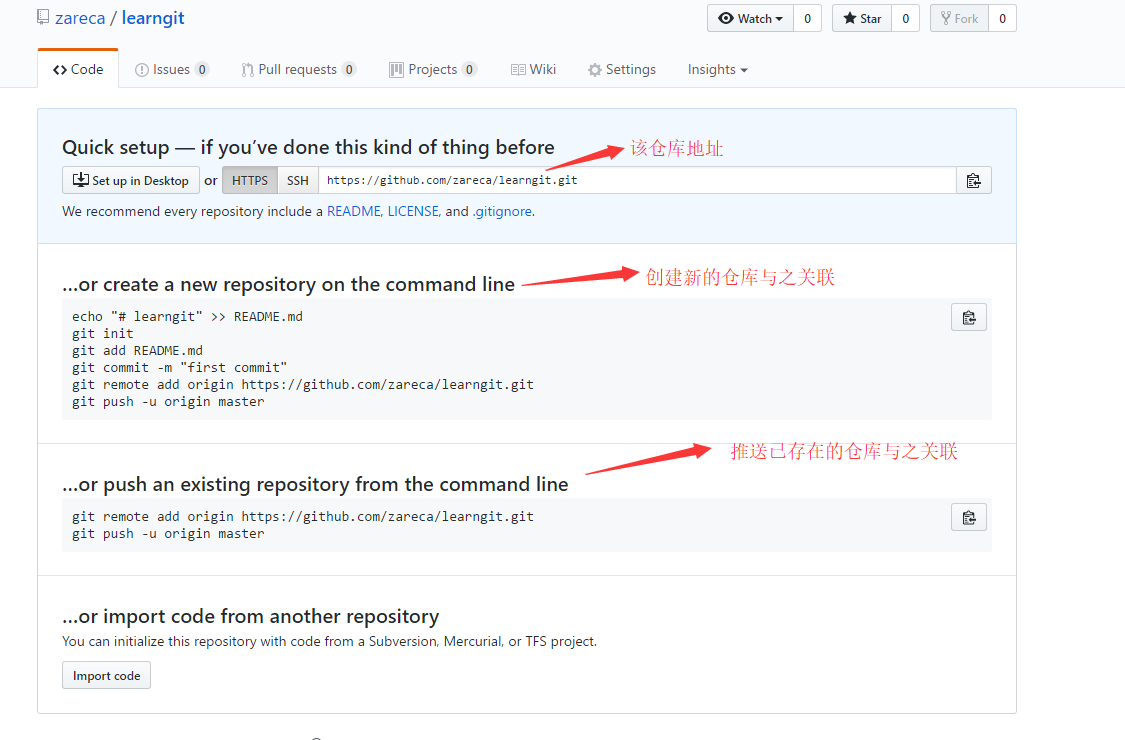
### 1.添加远程仓库

以Github为例

1).在github上创建仓库



创建完成之后



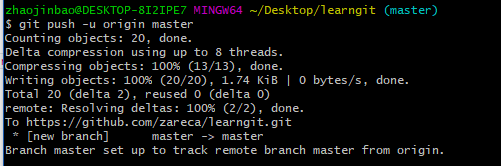
仓库创建完成之后，github会提示我们从这个仓库里克隆出新的仓库，或者创建新的仓库与之个管理，或者推送已存在的仓库与之关联。

2).把本地的仓库与github仓库关联。



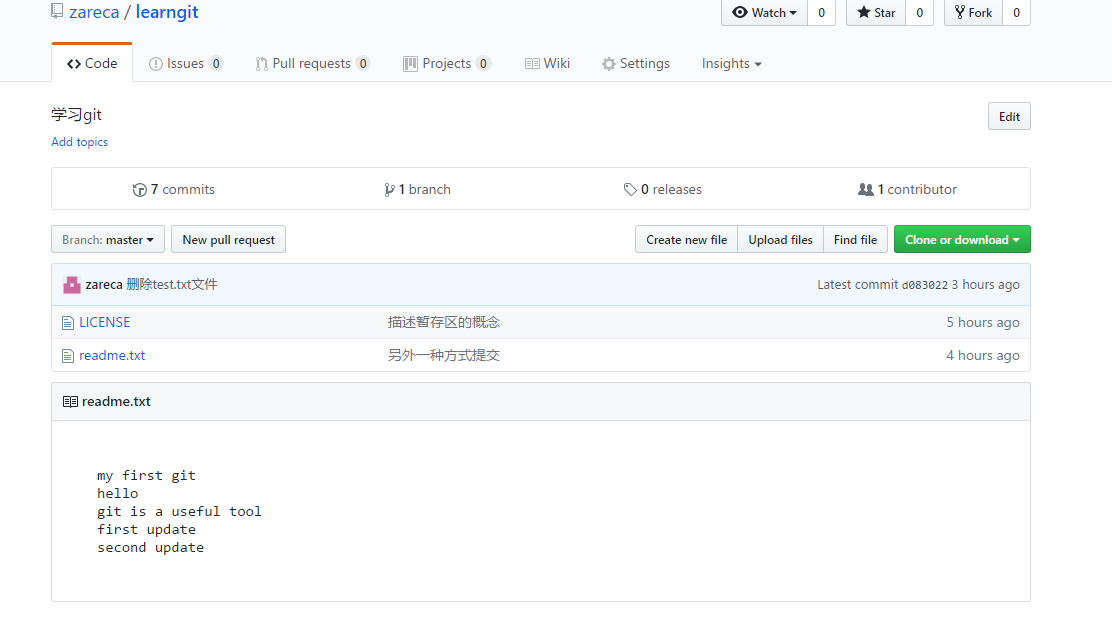
origin 是远程库的名字，github的默认叫法

3).把本地仓库的内容推送到github仓库



**。**

推送完成后，github上就有了你提交的内容



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上。

**小结：**

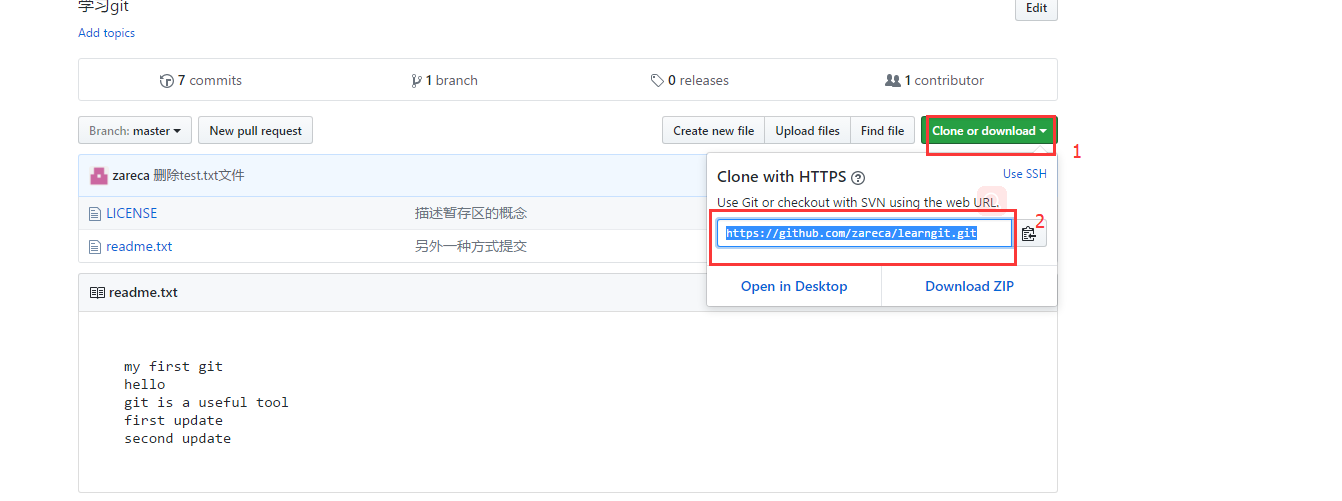
**git remote add origin xxx关联一个远程库**

**git push -u origin master 第一次将master分支的所有内容推送到远程库**

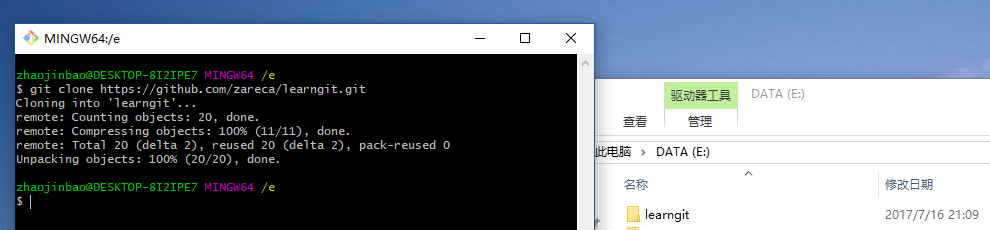
**git push origin master 上面命令第一次推送之后，以后的每次本地提交，使用不带 -u 参数的命令即可。**

### 2克隆远程仓库

上面讲了如何将本地仓库与远程仓库关联，还有一种情况就是将远程仓库克隆到本地



此时我将仓库克隆到了E盘



这样你的E盘就会出现你克隆的那个仓库。

**小结：**

**要克隆一个仓库，首先你必须要知道仓库的地址，然后使用git clone命令克隆**

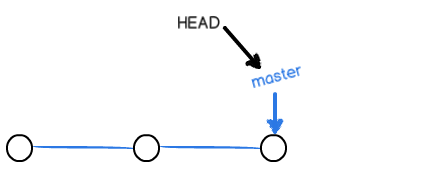
## 五．分支管理

分支：是指在开发主线上又开辟了一条线，这样能在不影响主线的同时继续工作

### 创建与合并分支

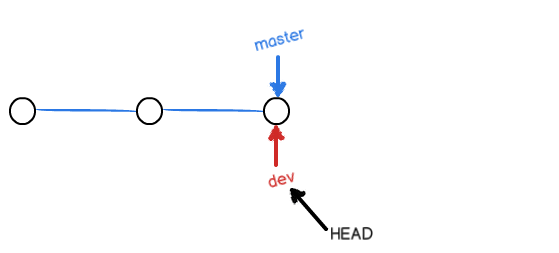
在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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：



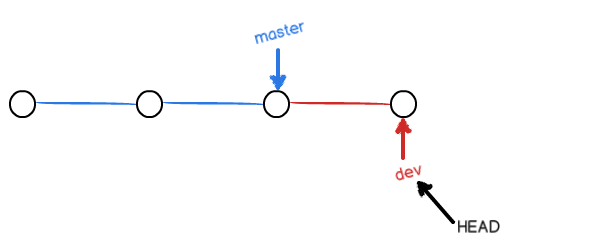
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：

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：

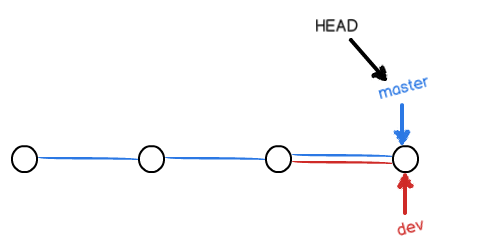


你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

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：

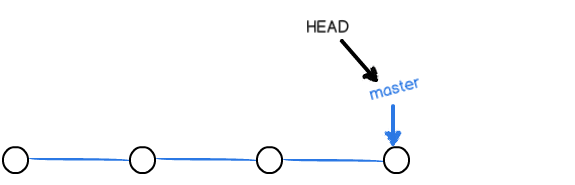


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



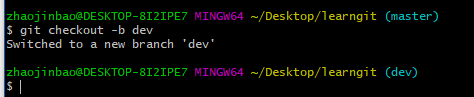
所以Git合并分支也很快！就改改指针，工作区内容也不变！

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：



实战：

1).创建并切换分支

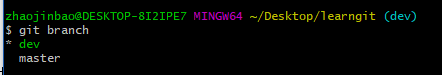


git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

git branch dev

git checkout dev

2).查看当前分支

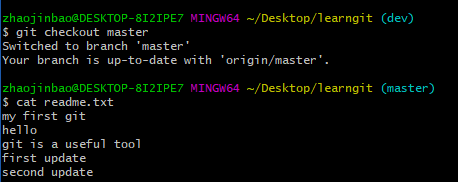


git branch命令会列出所有分支，当前分支前面会标一个\*号。

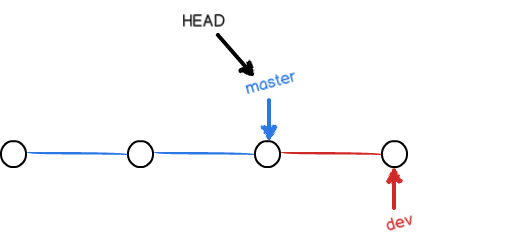
3).修改文件内容提交



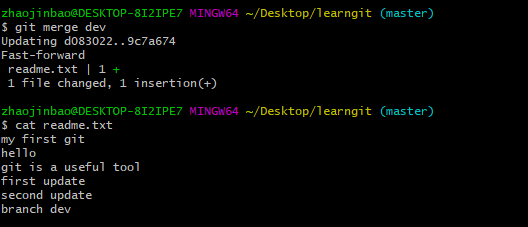
4).切换回主分支



切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！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：



5).将dev分支合并到master分支



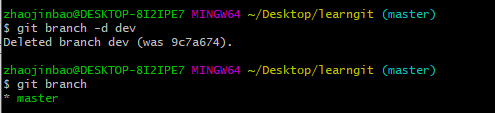
git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并完成后，就可以放心地删除dev分支了：

6).删除dev分支



删除后，再看branch，就只剩下master分支了。

**小结：**

**查看分支：git branch**

**创建分支：git branch <name>**

**切换分支：git checkout <name>**

**创建+切换分支：git checkout -b <name>**

**合并某分支到当前分支：git merge <name>**

**删除分支：git branch -d <name>**

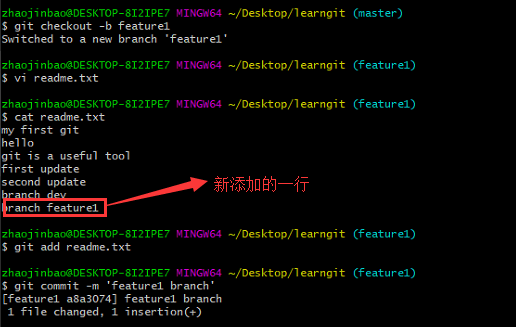
### 解决冲突

场景：你新建并切换到了分支feature1，然后修改了readme.txt文件并提交。随后你又切换到了master分支，同样修改了readme.txt文件并提交

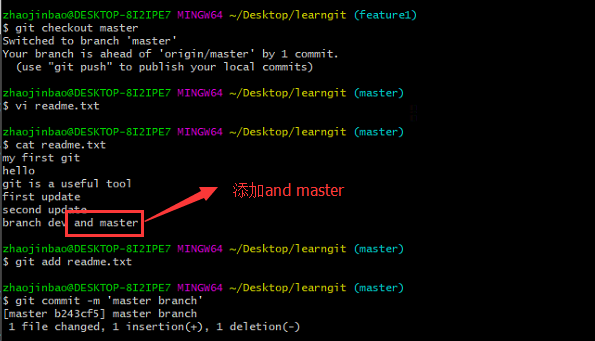
这时如果你要合并feature1分支时，即会产生冲突。

模拟并解决：

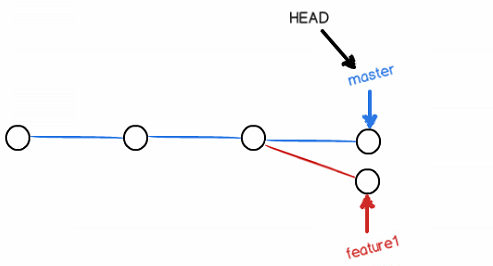
1).创建并切换分支feature1，修改readme.txt文件，最后提交



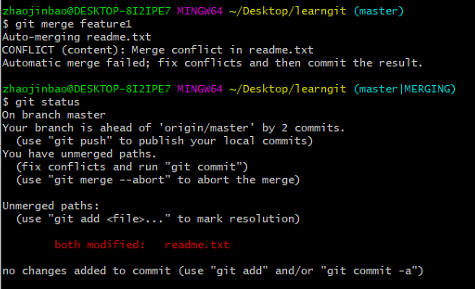
2).切换到主分支，修改readme.txt文件，并提交



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



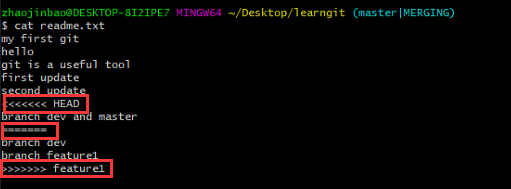
3).合并分支



这是git告诉我们 readme.txt文件产生冲突，必须手动解决冲突后再提交

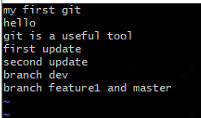
git status 也告诉我们冲突的文件。

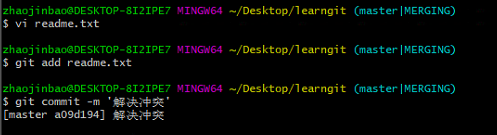
4).查看冲突文件



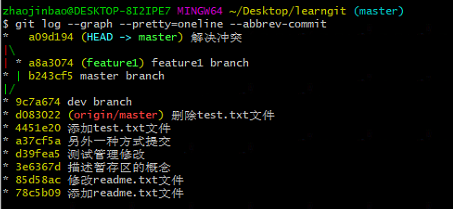
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存

5).修改文件，手动解决冲突





6).查看分支合并情况



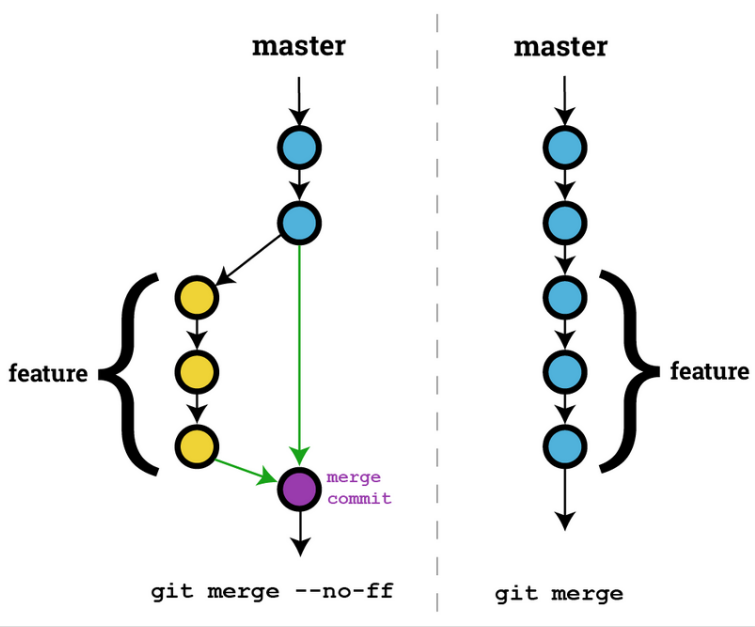
7).删除分支



扩展：

git merge xxx 默认是使用fast-forward，fast-forward方式就是当条件允许的时候，git直接把HEAD指针指向合并分支的头，完成合并。属于“快进方式”，不过这种情况如果删除分支，则会丢失分支信息。因为在这个过程中没有创建commit。

git merge --no-ff xxx 其中—no-ff指的是强行关闭fast-forward方式，使得合并分支后能够保留分支的commit的历史记录。



**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。**

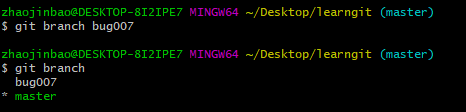
### Bug分支

bug在软件开发中就像家常便饭一样，有了bug就需要修复，而git又特别提倡使用分支，所以每个bug都可以通过创建一个临时分支来修复，修复后，合并分支，然后再删除临时分支。

场景：当你接到线上环境代号为007的bug时，很自然的会去创建一个bug分支，分支名为bug007，但是，此时你正在dev分支上进行工作还没有提交，并不是你不想提交，而是你的工作还没完成还没法提交，但是现在这个bug又非常紧急，要求你尽快解决，这时我们该怎么办？

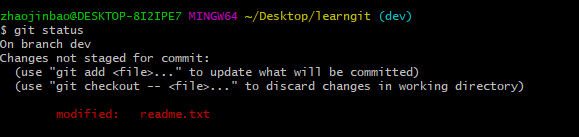
模拟并解决：

1).创建bug007分支



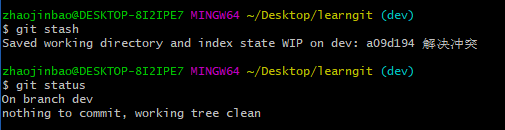
2).模拟在dev上正在工作还没有提交





3).git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

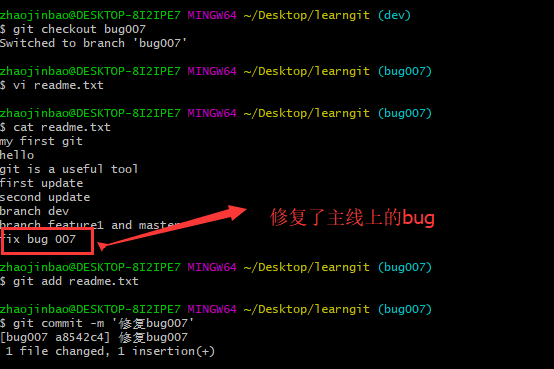
使用git stash将当前dev正在开发的工作储藏起来



再通过git status查看发现工作区现在是干净的了。

此时我们就可以切换到bug分支进行bug修复

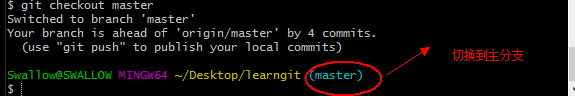
4).切换到bug007分支修复bug

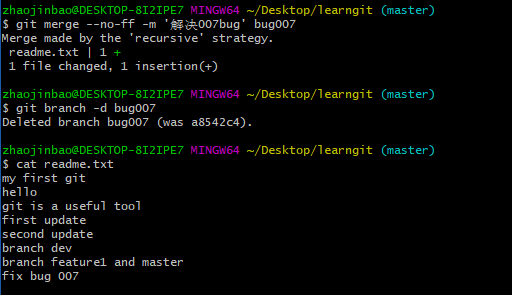


此时bug已经修改，那么我们就可以很放心的合并到主分支并将bug007分支删掉。

因为bug007是在主分支上建立的，所以此时需要切换到主分支上删除

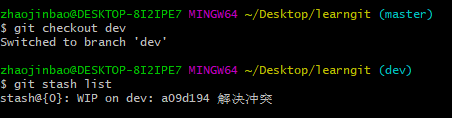
5).合并并删除bug007分支





6).之前为了修改bug，我们将dev的内容储藏起来了，那么现在我们怎么恢复呢？首先我们应该切换到dev分支上面，

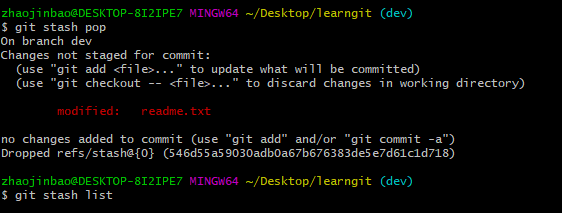
再通过git stash list 查看



我们发现之前的工作现场还在。

那我们可以通过git stash pop来恢复现场

7).恢复工作现场



再git stash list 发现已经没有内容了。

**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。**

### 4．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。

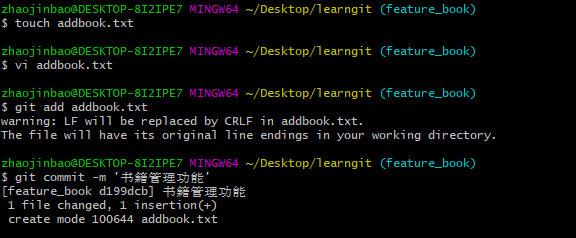
添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在你收到了一个新需求，要求你完成一个书籍管理的功能，

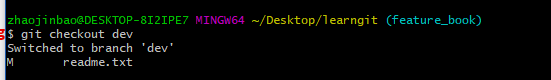
于是准备开发创建feature分支



半天后你开发完毕：



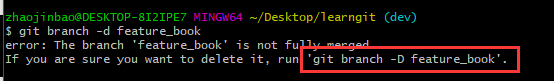
切换回dev分支准备合并



如果一切顺利的话，合并删除分支。

但是就在这时，接到上级命令说这个功能必须取消。

于是我们使用git branch -d 命令删除这个feature分支



这时git会提示我们，feature\_book没有被合并，如果真的要强行删除使用git branck -D 命令来删除



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。**