

# Git简介

## Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

## Git有什么特点？

简单来说就是：高端大气上档次！

## 那什么是版本控制系统？

指对软件开发过程中各种程序代码、配置文件及说明文档等文件变更的管理。版本控制最主要的功能就是追踪文件的变更。

# 集中式vs分布式

## 集中式的版本控制系统：CVS、SVN等

集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。集中式版本控制系统最大的毛病就是必须联网才能工作（局域网也可以）。

免费，速度慢且必须联网。有一台电脑充当“中央服务器”角色

## 分布式版本控制系统：Git、BitKeeper

分布式版本控制 系统没有“中央服务器”，每个人的电脑上都是一个完整的版本库，不需要联网，因为版本库就在自己的电脑上。

既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件 A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送 给对方，就可以互相看到对方的修改了。

免费，安全性高，不必联网，版本切换速度快，极其强大的分支管理，

# 在Windows上安装Git

从https://gitforwindows.org/下载，然后按默认选项安装 即可。



安装完成后，在开始菜单里找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！

安装完成后，还需要最后一步设置，在命令行输入：

$ 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。

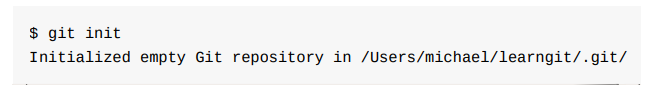
# 创建版本库

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”。

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：

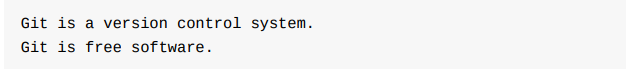


瞬间Git就把仓库建好了，而且告诉你是一个空的仓库。

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

# 知识点1：把文件添加到版本库

现在我们编写一个 readme.txt 文件，内容如下：



一定要放到 GitHome 目录下（子目录也行），因为这是一个Git仓库，放到其他 地方Git再厉害也找不到这个文件。

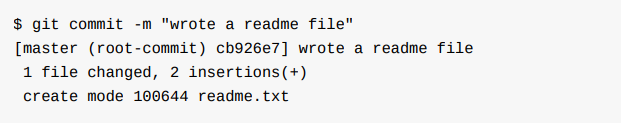
和把大象放到冰箱需要3步相比，把一个文件放到Git仓库只需要两步。

第一步，用命令 git add 告诉Git，把文件添加到仓库：



执行上面的命令，没有任何显示，这就对了，Unix的哲学是“没有消息就是好消 息”，说明添加成功。

第二步，用命令 git commit 告诉Git，把文件提交到仓库：

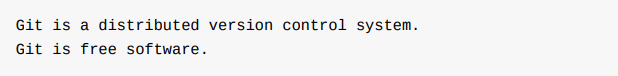


-m 后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

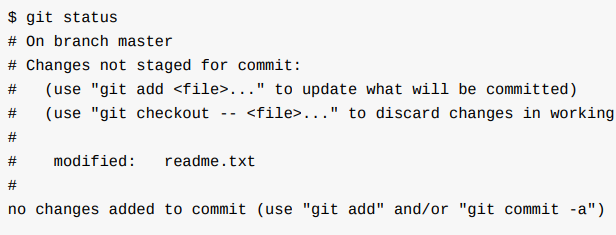
执行成功后会告诉你，1个文件被改动（我们新添加的 readme.txt文件），插入了两行内容（readme.txt有两行内容）。

# 知识点2：时光机穿梭(版本回退)

续修改readme.txt文件，改成如下内容：

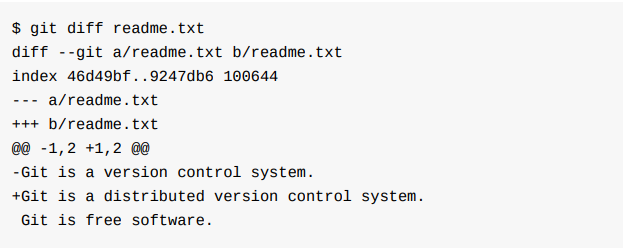


运行 git status 命令看看结果：



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。

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。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的 readme.txt，所以，需要用 git diff 这个命令看看：

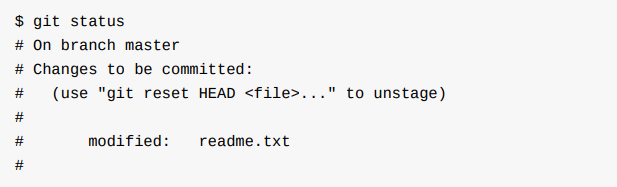


git diff 顾名思义就是查看difference。

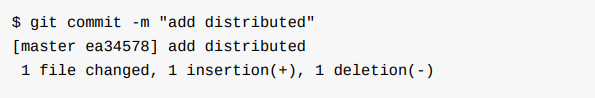
知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是 git add ：



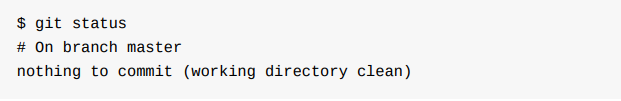
同样没有任何输出。在执行第二步 git commit 之前，我们再运行 git status 看看当前仓库的状态：



git status 告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心 地提交了：



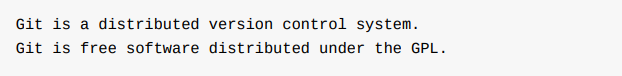
提交后，我们再用 git status 命令看看仓库的当前状态：



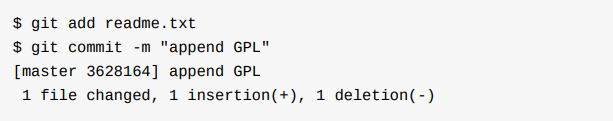
Git告诉我们当前没有需要提交的修改，而且，工作目录是干净的。

## 版本回退

修改readme.txt文件如下：

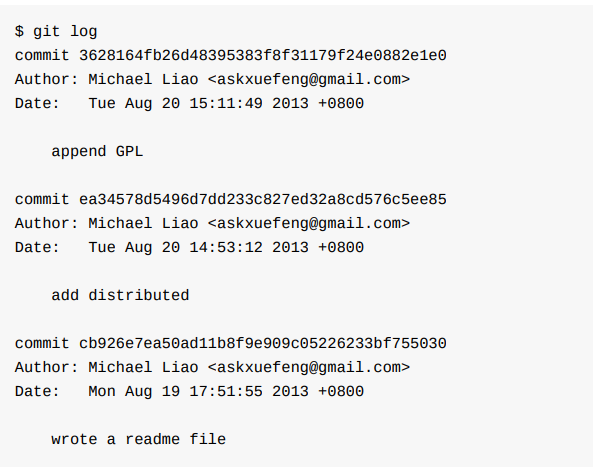


然后尝试提交：



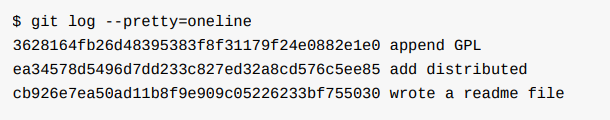
每当你觉得文件修改到一定程度 的时候，就可以“保存一个快照”，这个快照在Git中被称为 commit 。一旦你把文件 改乱了，或者误删了文件，还可以从最近的一个 commit 恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

版本控制系统肯定有某个命令可以告诉我们 历史记录，在Git中，我们用 git log 命令查看：



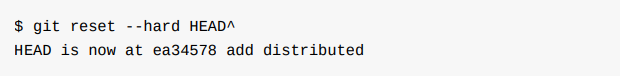
git log 命令显示从最近到最远的提交日志，我们可以看到3次提交。

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：



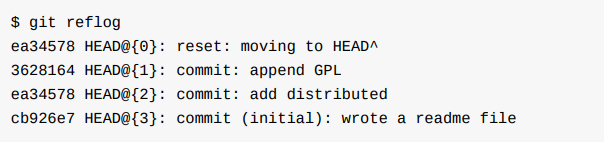
你看到的一大串类似 3628164...882e1e0 的是 commit id （版本号）。

用 HEAD 表示当前版本，上一 个版本就是 HEAD^ ，上上一个版本就是 HEAD^^ 。回退到上一个版本“add distributed”，就可 以使用 git reset 命令：



也可以使用 git reset –hard 版本号 来回退指定版本

Git提供了一个命令 git reflog 用来记录你的每一次命令，查看版本号：

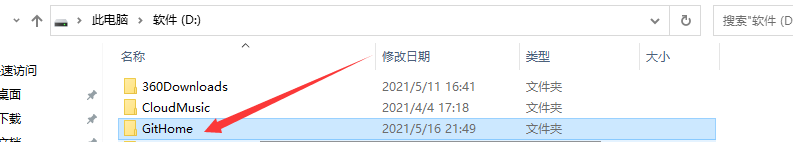


# 知识点3：工作区和暂存区

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。

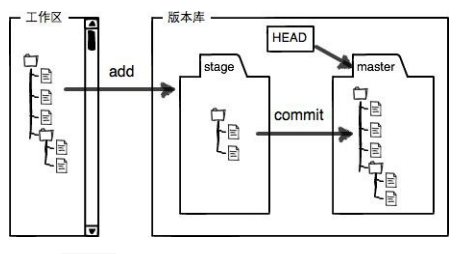
## 工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的 GitHome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 。



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；

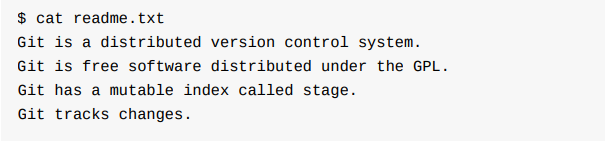
第二步是用 git commit 提交更改，实际上就是把暂存区的所有内容提交到当前分支。

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。

# 知识点4：管理修改

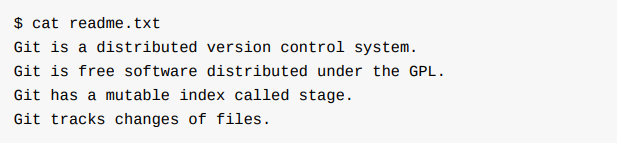
第一步，对 readme.txt做一个修改，比如加一行内容：



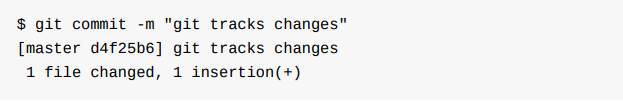
然后，添加：



然后，再修改readme.txt：



提交：



提交后，再看看状态：

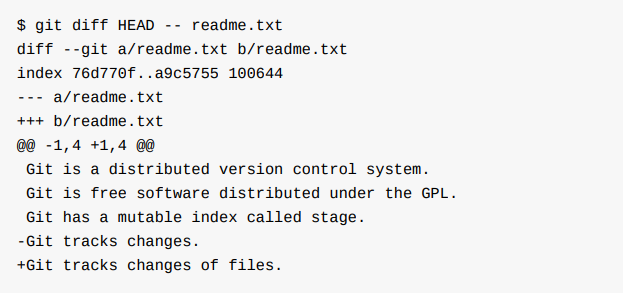


咦，怎么第二次的修改没有被提交？

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。

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：

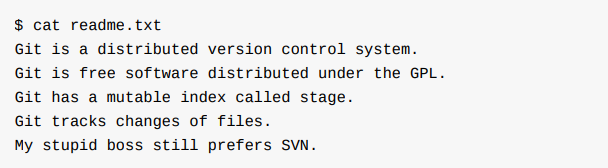


可见，第二次修改确实没有被提交。

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

# 知识点5：撤销修改

在 readme.txt 中添加一行：



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：

用命令 git reset HEAD file 可以把暂存区的修改撤销掉，重新放回工作区。

小结：

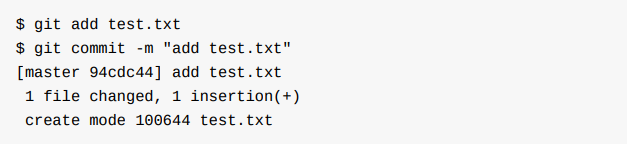
场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命 令 git checkout -- file 。

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，不过前提是没有推送到远程库。

# 知识点6：删除文件

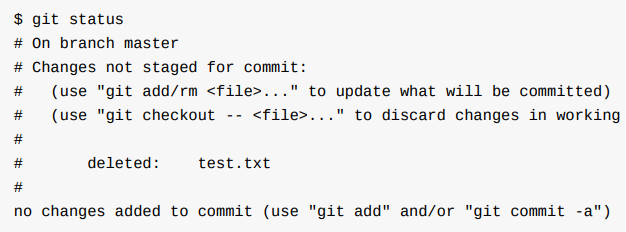
在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git 并且提交：



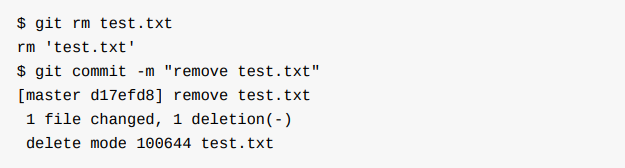
直接在文件管理器中把没用的文件删了，或者用 rm 命令删 了：



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 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。

# 知识点7：远程仓库

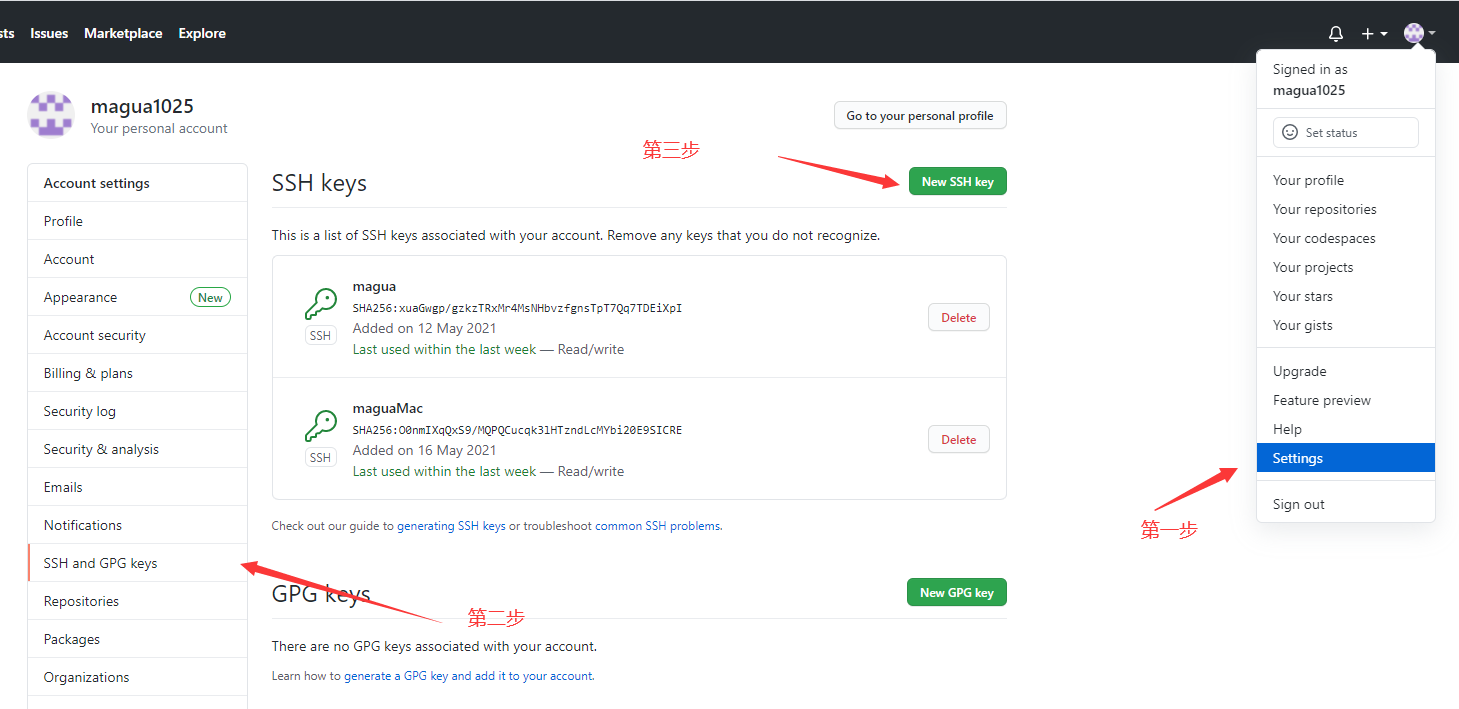
去 <https://github.com/> 注册账号

由于你的本地Git仓库和GitHub 仓库之间的传输是通过SSH加密的，所以，需要一点设置：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：



第2步：登陆GitHub，打开“Settings”，“SSH and GPG key”页面： 然后，点“New SSH key”，填上任意Title，在Key文本框里粘贴 id\_rsa.pub 文件 的内容：

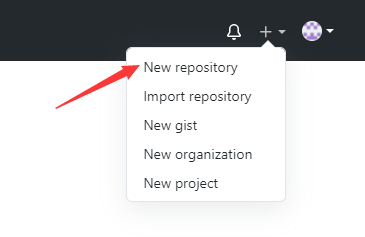


点“Add Key”，你就应该看到已经添加的Key

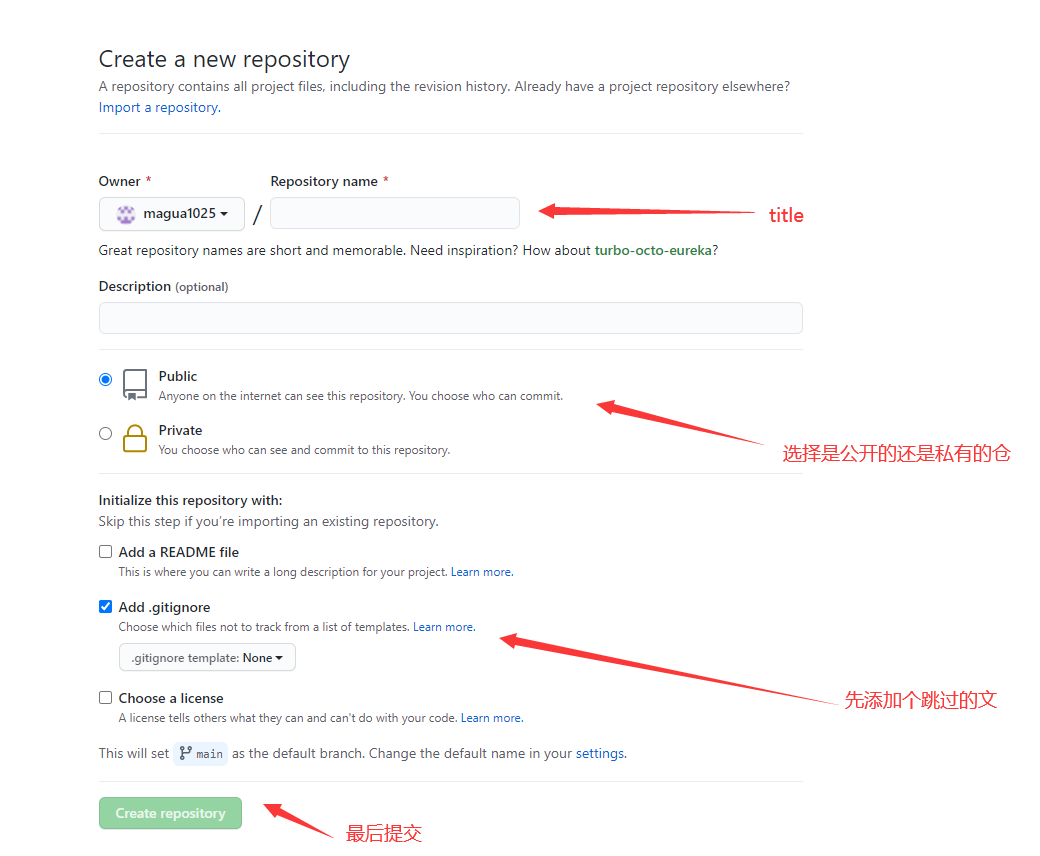
# 知识点8：添加远程库

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。

首先，登陆GitHub，然后，在右上角找到“New repository”按钮，创建一个新的仓库：



在Repository name填入 learngit ，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：



在Repository name填入 learngit ，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：



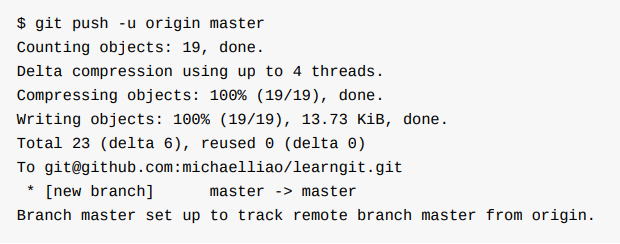
目前，在GitHub上的这个 learngit 仓库还是空的，GitHub告诉我们，可以从这 个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓 库的内容推送到GitHub仓库。

现在，我们根据GitHub的提示，在本地的 learngit 仓库下运行命令：



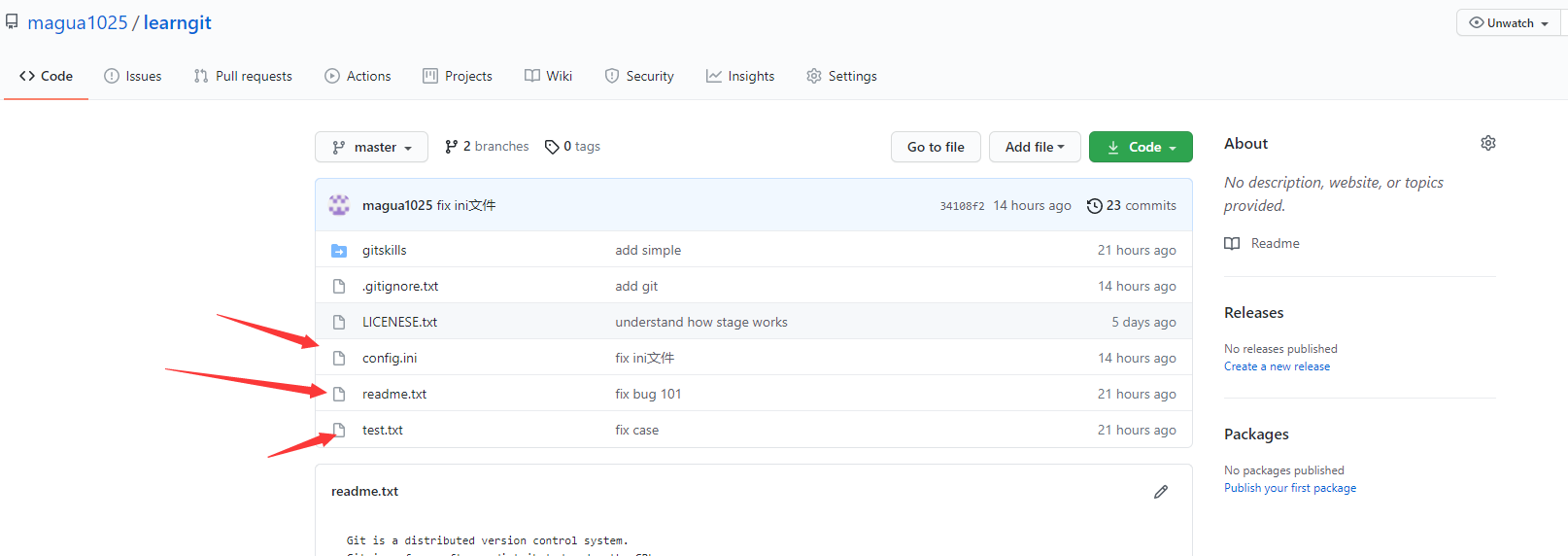
请千万注意，把上面的红色部分替换成你自己的GitHub账户名，否则，你在 本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

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。 下一步，就可以把本地库的所有内容推送到远程库上：



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：

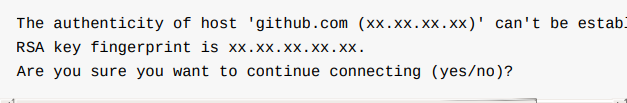


从现在起，只要本地作了提交，就可以通过命令：



## SSH警告

当你第一次使用Git的 clone 或者 push 命令连接GitHub时，会得到一个警告：



这是因为Git使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要 你确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入 yes 回车即可。

**小结：**

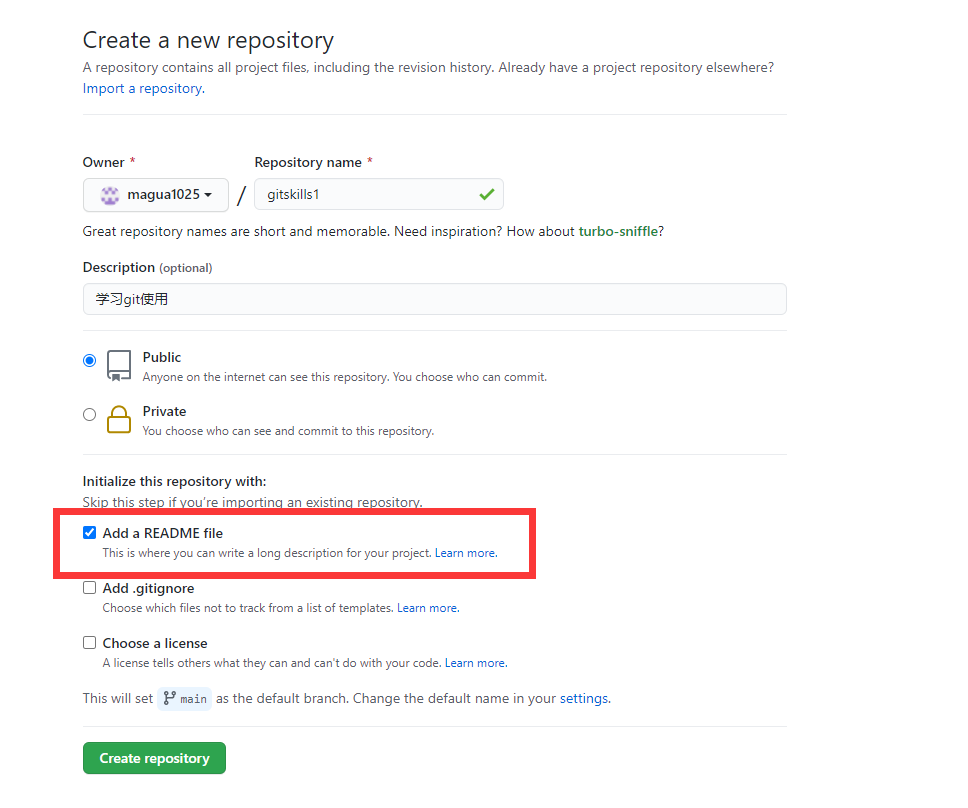
要关联一个远程库，使用命令 git remote add origin git@servername:path/repo-name.git ；

关联后，使用命令 git push -u origin master 第一次推送master分支的所有 内容；

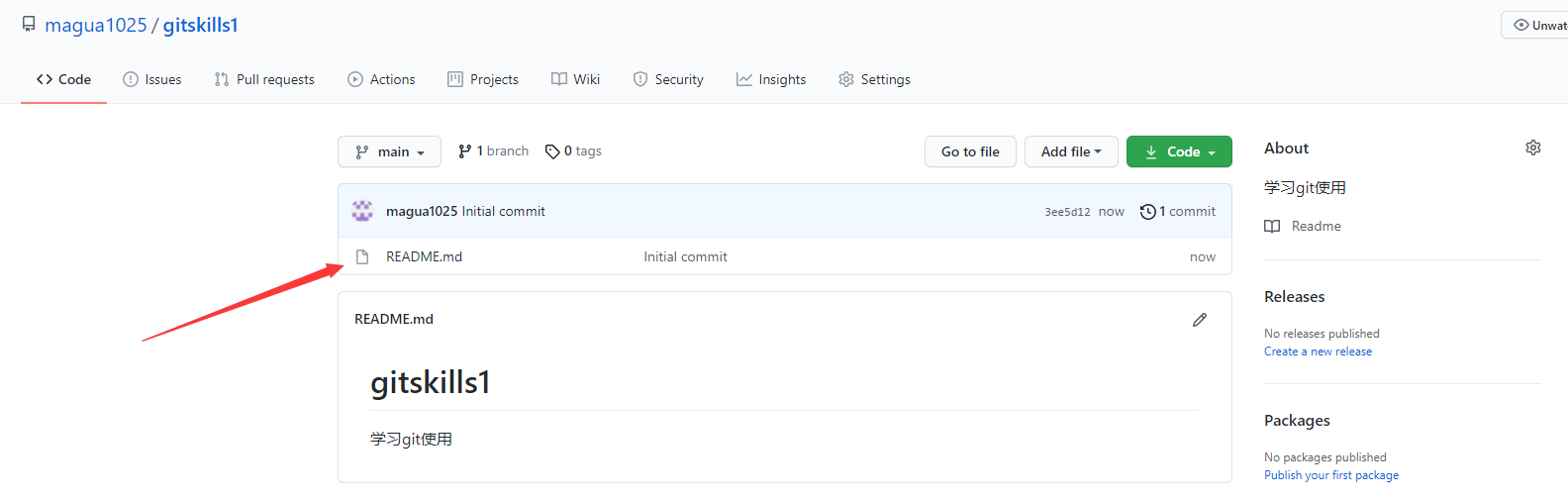
此后，每次本地提交后，只要有必要，就可以使用命令 git push origin master 推送最新修改；

# 知识点9：从远程库克隆

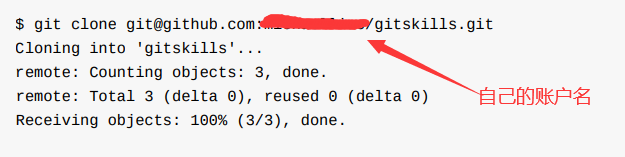
首先，登陆GitHub，创建一个新的仓库，名字叫 gitskills ：



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：



注意把Git库的地址换成你自己的，然后进入 gitskills 目录看看，已经 有 README.md 文件了。

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

Tips：直接克隆仓库的url也可以

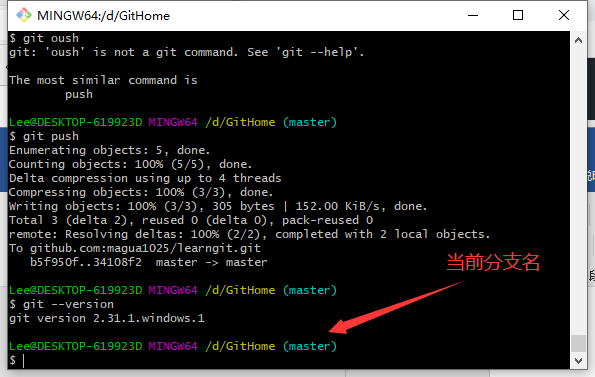
小结：

要克隆一个仓库，首先必须知道仓库的地址，然后使用 git clone 命令克隆。 Git支持多种协议，包括 https ，但通过 ssh 支持的原生 git 协议速度最快。

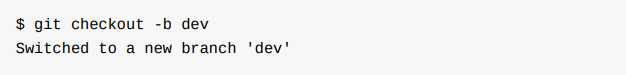
# 知识点10：分支管理

## 创建与合并分支

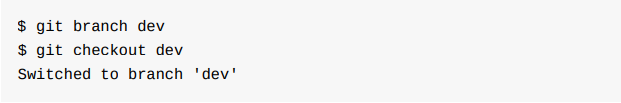
在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线 就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支， 即 master 分支。



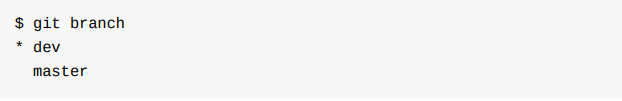
首先，我们创建 dev 分支，然后切换到 dev 分支：



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：



然后，用 git branch 命令查看当前分支：

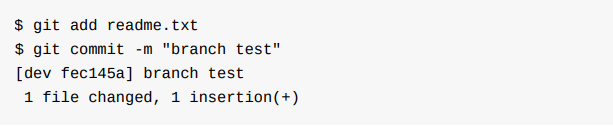


git branch 命令会列出所有分支，当前分支前面会标一个 \* 号。

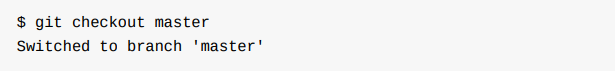
然后，我们就可以在 dev 分支上正常提交，比如对readme.txt做个修改，加上一 行：



然后提交：

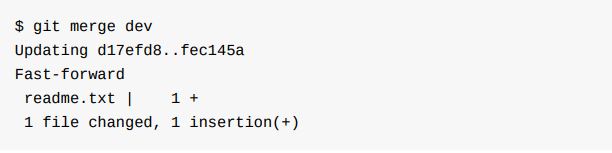


现在， dev 分支的工作完成，我们就可以切换回 master 分支：



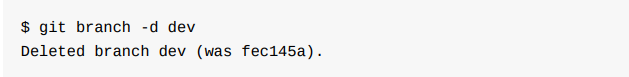
切换回 master 分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因 为那个提交是在 dev 分支上，而 master 分支此刻的提交点并没有变

现在，我们把 dev 分支的工作成果合并到 master 分支上：

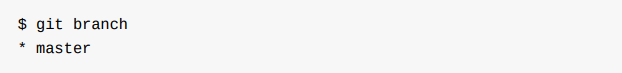


git merge 命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内 容，就可以看到，和 dev 分支的最新提交是完全一样的。

合并完成后，就可以放心地删除 dev 分支了：



删除后，查看 branch ，就只剩下 master 分支了：



小结：

Git鼓励大量使用分支：

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>

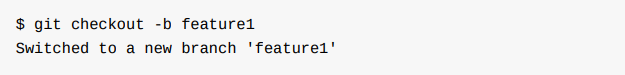
合并某分支到当前分支： git merge <name>

删除分支： git branch -d <name>

# 知识点11：解决冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的 feature1 分支，继续我们的新分支开发：



修改readme.txt最后一行，改为：



在 feature1 分支上提交：



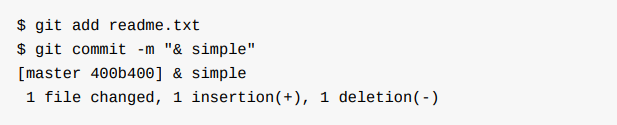
切换到 master 分支：



Git还会自动提示我们当前 master 分支比远程的 master 分支要超前1个提交。 在 master 分支上把readme.txt文件的最后一行改为：

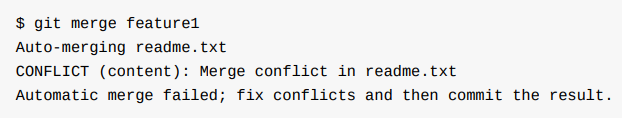


提交：

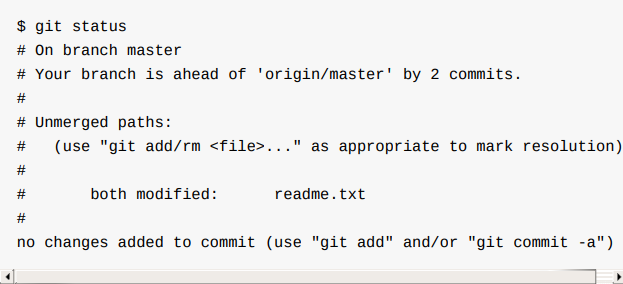


现在， master 分支和 feature1 分支各自都分别有新的提交。

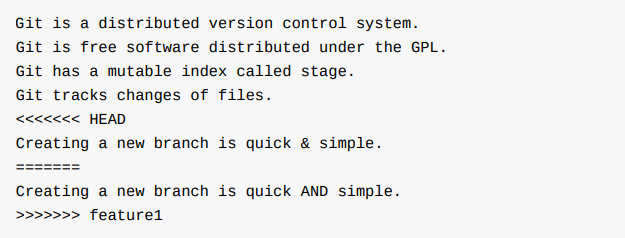
这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合 并就可能会有冲突，我们试试看：



果然冲突了！Git告诉我们，readme.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：



我们可以直接查看readme.txt的内容：



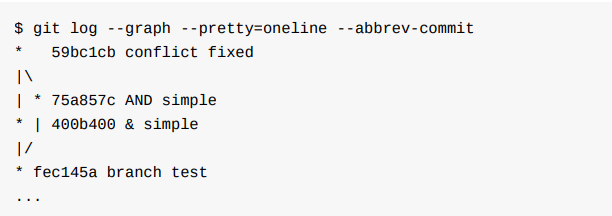
Git 用 <<<<<<< ， ======= ， >>>>>> 标记出不同分支的内容，我们修改如下后保存：



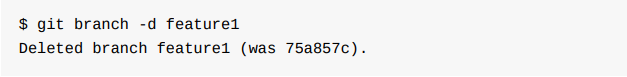
再提交：



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：

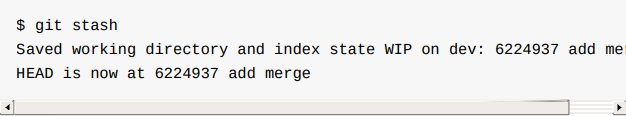
当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用 git log –graph –pretty=oneline 命令可以看到分支合并图。

# 知识点12： Bug分支

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？

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：

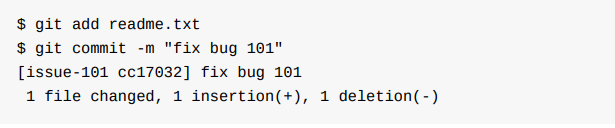


现在，用 git status 查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

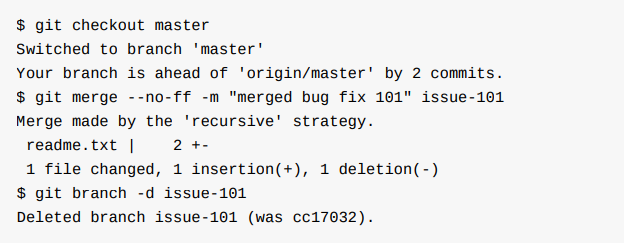
首先确定要在哪个分支上修复bug，假定需要在 master 分支上修复，就 从 master 创建临时分支：



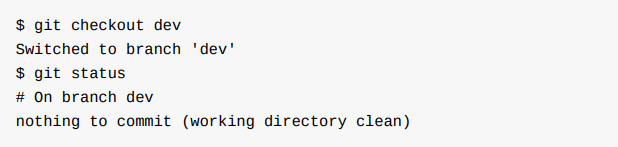
现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后 提交：



修复完成后，切换到 master 分支，并完成合并，最后删除 issue-101 分支：



现在，是时候接着回到 dev 分支干活了！



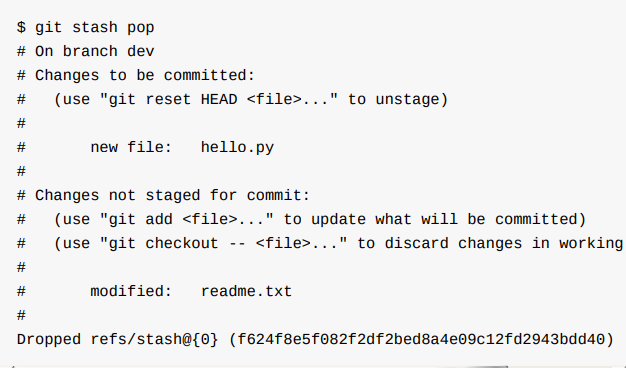
工作区是干净的，刚才的工作现场存到哪去了？用 git stash list 命令看看：



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：



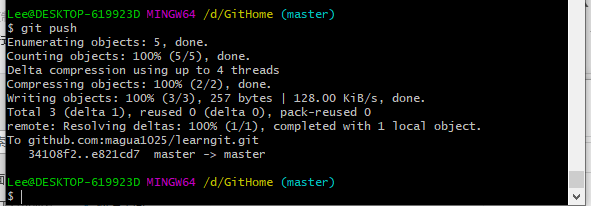
小结：

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。

# 知识点13：推送到远程库

我们可以使用命令 git push 推送到关联的远程分支：



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，需要先用 git pull 试图合并；

3. 如果合并有冲突，则解决冲突，并在本地提交；

4. 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 。