## git用的基本都是linux命令

## git 的提交

两个步骤

git add 文件名.后缀

然后是

git commit -m “这里跟的是这次的说明”

## 查看修改的文件

git diff readme.txt

文件修改了

就直接

git diff 文件名.后缀

查看文件区别之前是不能add的

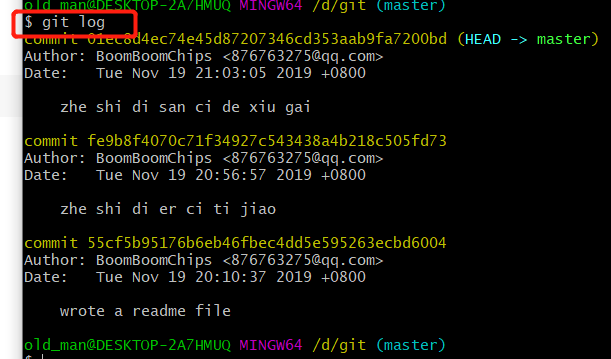
要是add了的话就要在缓存中查看区别

git diff --cached file\_name

这样就能在add之后看到区别

## 查看提交的历史纪录

git log



### 查看提交的

### 加上--pretty=oneline参数：

这样出来的就是每次修改的对应id，然后后面跟的是注释

### 版本的回退：

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1094adb...（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。

git re**set** --hard HEAD^

然后查看版本

git log 就没有第三次的提交记录了，

就只有两次的提交记录了

### 回退回退之前的版本

最新的那个版本append GPL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append GPL的commit id是1094adb...，于是就可以指定回到未来的某个版本：

$ git re**set** --hard 1094a

HEAD **is** now **at** 83b0afe append GPL

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。

查看之前的id ,然后他的头是5位数

## 查看会退回退之前的版本

git reflog

## 撤销修改

$ git checkout *-- readme.txt*

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。

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。

## 撤销缓存区的修改重新放回工作区

git reset HEAD <file>

这样历史记录就不会有东西

## 删除操作

直接删除

$ rm test.txt

注意：直接删除git status也是能看到删除信息的

Git删除

$ git rm test.txt



要想找回来可以直接

$ git checkout -- test.txt



所以是不能直接删除的

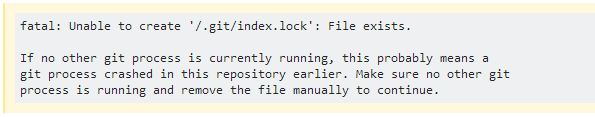
# Error:

## 提交错误：

### Index.lock错误

git提交代码出现错误fatal: Unable to create '项目路径/.git/index.lock': File exists.

具体出错代码如下：



具体原因不详，在stackoverflow上查找了很久，也不清楚原因，但是stackoverflow给出了解决的方法。

解决方法：

在当前项目下打开git bash，运行如下命令：

rm -f ./.Git/index.lock

## 添加远程仓库并提交本地数据

#### 添加远程库

阅读: 106816389

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。

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：



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：

$ git remote add origin git@github.com:michaelliao/learngit.git

请千万注意，把上面的michaelliao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：

$ git push -u origin master

Counting objects: 20, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (15/15), done.

Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.

Total 20 (delta 5), reused 0 (delta 0)

remote: Resolving deltas: 100% (5/5), done.

To github.com:michaelliao/learngit.git

\* [new branch] master -> master

Branch 'master' **set** up **to** track remote branch 'master' **from** 'origin'.

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！

### 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：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

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。

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the **list** of known hosts.

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入yes前可以对照[GitHub的RSA Key的指纹信息](https://help.github.com/articles/what-are-github-s-ssh-key-fingerprints/" \t "https://www.liaoxuefeng.com/wiki/896043488029600/_blank)是否与SSH连接给出的一致。

### 小结

要关联一个远程库，使用命令git remote add origin git@server-name: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；

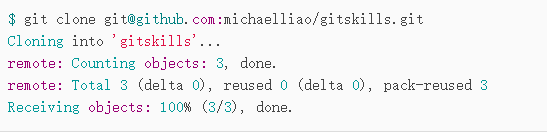
此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，**再把本地提交推送一下就完成了同步，真是太方便了！**

## 克隆仓库：

先进入到master

git clone 仓库里的地址



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。

# Git分支

注意创建的分支是在当前文件夹下面的

## 创建分支并切换

创建dev分支，然后切换到dev分支：

$ git checkout -b devSwitched to a new branch '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 devSwitched to branch 'dev'

## 分支切换

我们注意到切换分支使用git checkout <branch>，而前面讲过的撤销修改则是git checkout -- <file>，同一个命令，有两种作用，确实有点令人迷惑。

实际上，切换分支这个动作，用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支：

创建并切换到新的dev分支，可以使用：

$ git switch -c dev

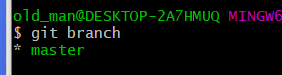
直接切换到已有的master分支，可以使用：

$ git switch master

使用新的git switch命令，比git checkout要更容易理解。

## 查看当前分支

git branch



## 将分支合并到主线上

git merge dev

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。

## 删除分支

git branch -d dev

## 分支使用小结

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>或者git switch <name>

创建+切换分支：git checkout -b <name>或者git switch -c <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

一个分支改一个文件的时候

Add,

然后，另外一个分支master修改后，

Add，然后提交commit

之后就会出现问题

就是我们所说的冲突

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

用git log --graph命令可以看到分支合并图。

## 分支合并

Git分支十分强大，在团队开发中应该充分应用。

git merge

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。

## 分支储藏

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：

$ git stashSaved working directory **and** index state WIP on dev: f52c633 add merge

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。

如何修复好分支

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 masterSwitched to branch 'master'Your branch is ahead of 'origin/master' by 6 commits.

(use "git push" to publish your local commits)

$ git checkout -b issue-101Switched to a new branch 'issue-101'

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

$ git add readme.txt $ git commit -m "fix bug 101"

[issue-101 4c805e2] fix bug 101

1 file changed, 1 insertion(+), 1 deletion(-)

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

查看工作现场：**git stash list**

然后要回到原来的分支，就是

回到dev分支干活了！

$ git checkout devSwitched to branch 'dev'

$ git statusOn branch dev

nothing to commit, working tree clean

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：

$ git stash list

stash@{0}: WIP on dev: f52c633 add merge

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：

## bug修复总结

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；

在master分支上修复的bug，想要合并到当前dev分支，可以用git cherry-pick <commit>命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

## Feature

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。

EG:

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：

$ git checkout -b feature-vulcanSwitched to a new branch 'feature-vulcan'

5分钟后，开发完毕：

$ git add vulcan.py

$ git status

切回dev，准备合并：

$ git checkout dev

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，但是这个包含机密资料的分支还是必须就地销毁：

$ git branch -d feature-vulcanerror: The branch 'feature-vulcan' is **not** fully merged.If you are sure you want to delete it, run 'git branch -D 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，需要使用大写的-D参数。。

现在我们强行删除：

$ git branch -D feature-vulcanDeleted branch feature-vulcan (was 287773e).

终于删除成功！

**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

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dev**，就改成：

$ git push origin dev

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；

dev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。

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。

## Rebase

# 标签

$ git branch

\* dev

master$ git checkout masterSwitched to branch 'master'

然后，敲命令git tag <name>就可以打一个新标签：

$ git tag v1.0

可以用命令git tag查看所有标签：

$ git tag

v1.0



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

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

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

命令git tag可以查看所有标签。

## 删除标签

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。