# Git教程

#### Git简介

##### 1.1 Git是什么？

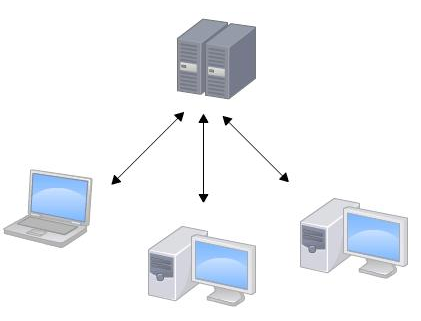
Git是目前世界上最先进的分布式版本控制系统（没有之一）。

Linus在1991年创建了开源的Linux，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！在2002年Linus选择了BitMover公司授权Linux社区免费使用一个商业的版本控制系统BitKeeper，但是，在2005年，BitMover公司因发现了有人试图破解BitKeeper的协议，要收回Linux社区的免费使用权，于是Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

##### 1.2 集中式和分布式版本控制系统有什么区别？

**集中式**：

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。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。



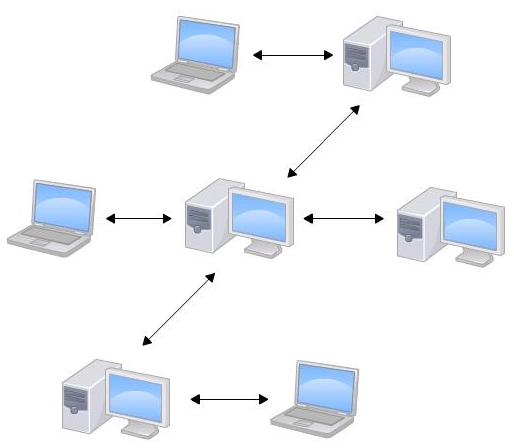
**缺点:**

1.必须联网才能工作

2.安全度不高，重要的东西都放在一个中央服务器中,如果被黑或者挂了，就呵呵了

**分布式**：

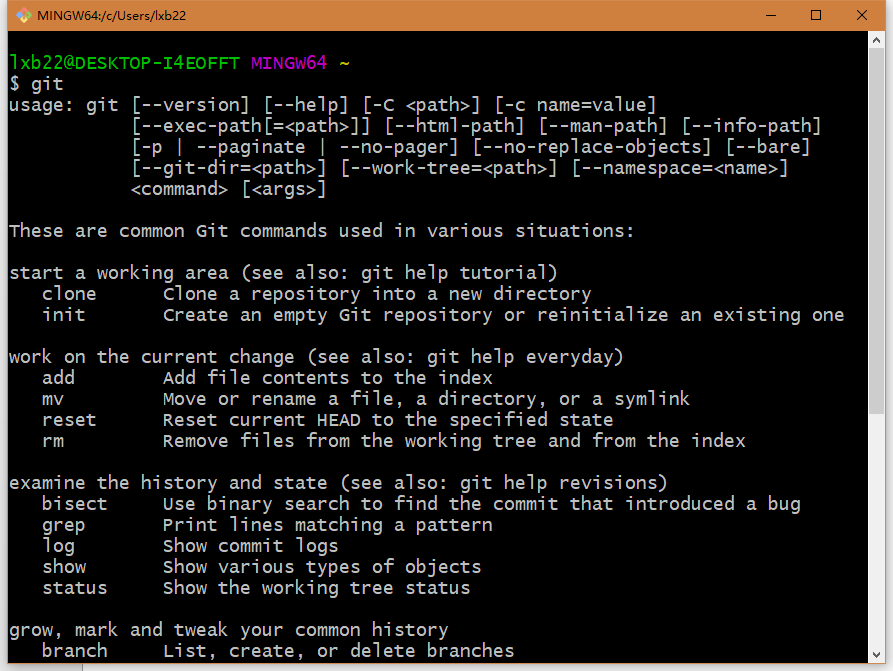
首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。和集中式版本控制系统相比，分布式版本控制系统的安全性高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。当然，Git的优势不单是不必联网这么简单，后面我们还会看到Git极其强大的分支管理。



#### Git的安装

##### 2.1 Git的安装（Windows）

在Windows上使用Git，可以从Git官网直接[下载安装程序](https://git-scm.com/downloads)，（网速慢的请移步[国内镜像](https://pan.baidu.com/s/1kU5OCOB#list/path=%2Fpub%2Fgit)），然后按默认选项安装即可。安装完成后，在开始菜单里或右键找到“Git”->“Git Bash”，输入”git”,会出现一大堆命令之类的东西，获取”git --version”可以查看git版本号，就说明Git安装成功！





Tip:顺便提一句：git清理控制台和windows有点不一样，

Windows的控制台清理是: **cls**

Git的控制台清理时：

**reset**(这个命令将会刷新屏幕，本质上只是让终端显示页向后翻了一页，如果向上滚动屏幕还可以看到之前的操作信息。一般都会用这个命令。)

**clear**(这个命令将完全刷新终端屏幕，之前的终端输入操作信息将都会被清空，这样虽然比较清爽，但整个命令过程速度有点慢，使用较少。)

##### 2.2 Git的配置

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，当然也可以对某个仓库指定不同的用户名和Email地址。

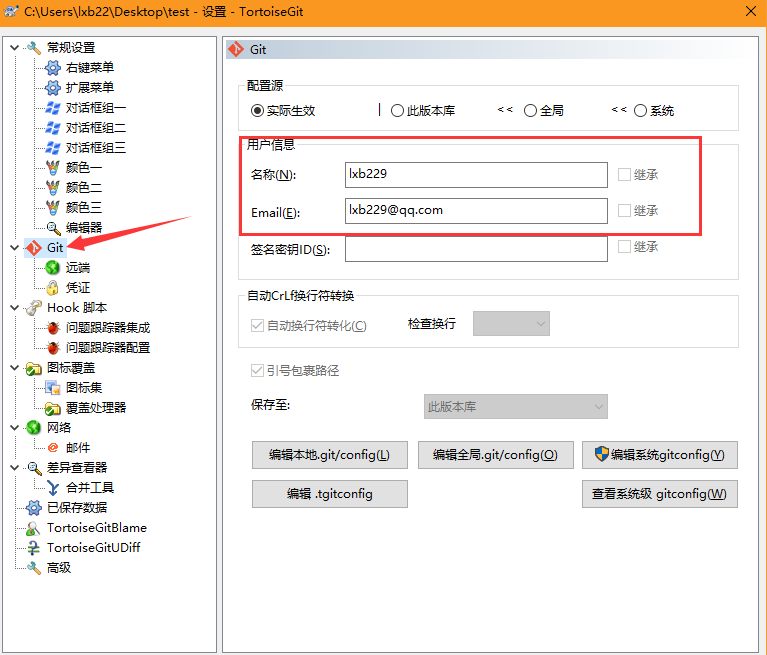
##### 2.3 Git的图形化界面TortoiseGit安装

TortoiseGit 简称 tgit, 中文名海龟Git. 海龟Git只支持神器 Windows 系统, 有一个前辈海龟SVN, TortoiseSVN和TortoiseGit都是非常优秀的开源的版本库客户端. 分为32位版与64位版.并且支持各种语言,包括简体中文(Chinese, simplified; zh\_CN). [TortoiseGit下载页面](https://tortoisegit.org/download/)。

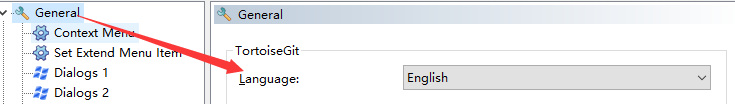
先安装程序包,然后安装语言包(LanguagePack). 因为TortoiseGit 只是一个程序壳,必须依赖一个 Git Core。前面的几步都是默认下一步，在选择SSH客户端的时候. 可以选择 TortoiseGitPlink(位于TortoiseGit安装目录/bin 下), 也可以选择 Git 默认的SSH客户端,位于 Git安装目录/bin/ssh.exe(如果配置了 Path,那直接是 ssh.exe)，后面的都默认(如果需要的话，可以选择自己的安装目录)



在Git设置中，可以看到我们之前设置好的用户名和email，其他的配置都可以先不用配置。



不要问我为什么是中文的，先下载的语言包，安装了在 git设置—常规设置—语言-中文



#### 创建版本库

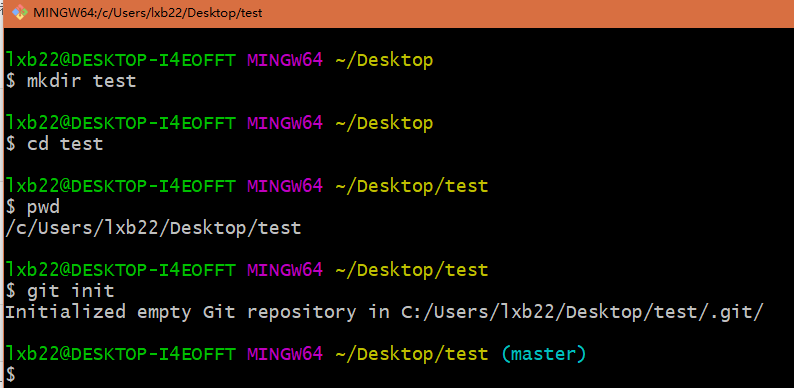
什么是版本库呢？版本库又名仓库，英文名**repository**，你可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：



pwd命令用于显示当前目录。在我的电脑上个仓库位于/Desktop/test(如Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。)

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

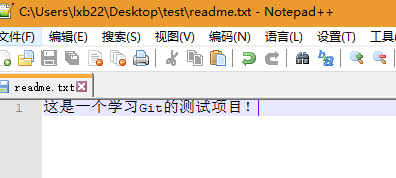
瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。如果你没有看到.git目录，那是因为这个目录默认是隐藏的，用ls -ah命令就可以看见。

使用Windows的要特别注意：不要使用Windows自带的**记事本**编辑任何文本文件, 原因是记事本是用UTF-8编码来保存文件的，建议你下载[Notepad++](http://notepad-plus-plus.org/)代替记事本，把Notepad++的默认编码设置为UTF-8 without BOM即可。

#### 四．Git命令操作

##### 4.1 Git的添加与提交

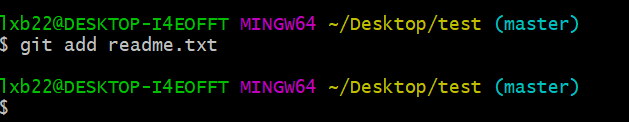
现在我们编写一个readme.txt文件，内容如下：



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：

$ git add readme.txt



执行上面的命令，没有任何显示，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：

$ git commit -m "添加readme.txt文件"

$ git commit –a –m "提交的描述信息"

$ git add .

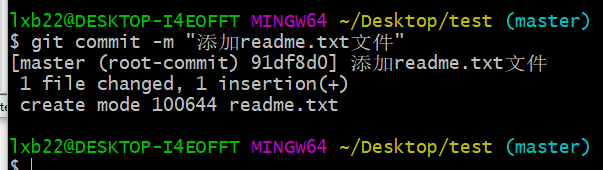
$ git commit –amend

git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容。

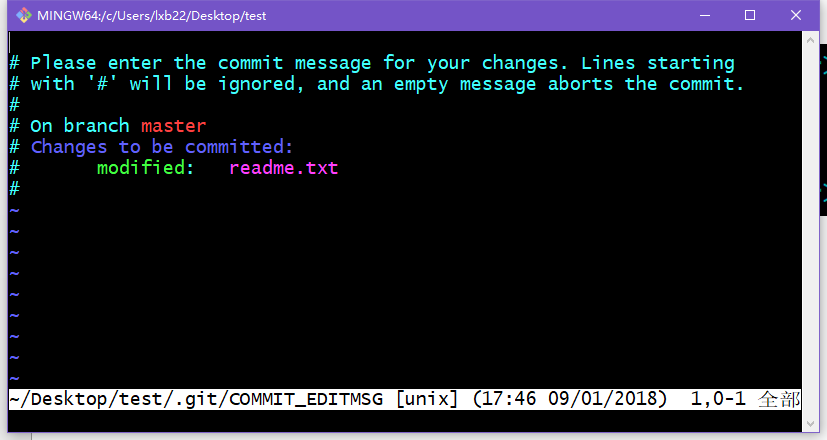
git commit 命令的-a 选项可只将所有被修改或者已删除的且已经被git管理的文档提交倒仓库中。如果只是修改或者删除了已被Git管理的文档，是没必要使用git add 命令的。

git add .命令除了能够判断出当前目录（包括其子目录）所有被修改或者已删除的文档，还能判断用户所添加的新文档，并将其信息追加到索引中。

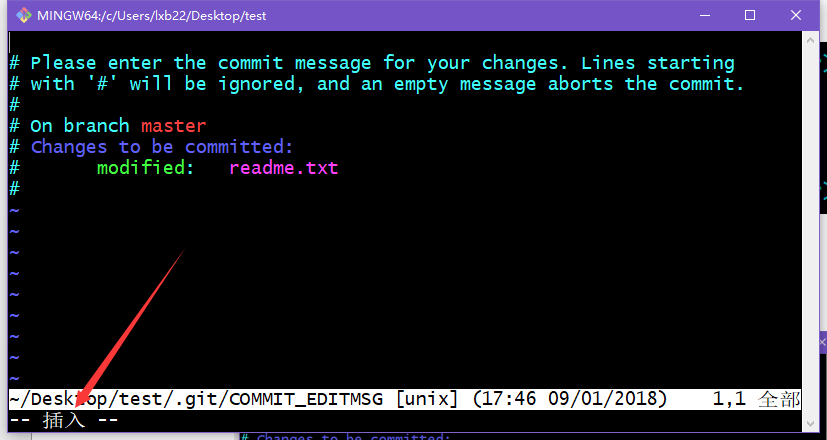
git commit --amend 对于已经修改提交过的注释，如果需要修改，可以借助 git commit --amend 来进行。



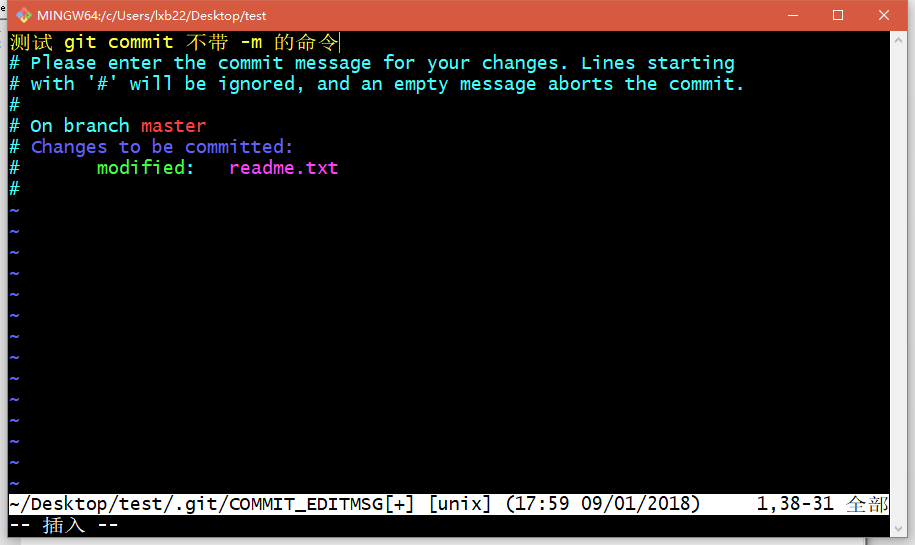
附：如果我们这里不用-m参数的话，git将调到一个文本编译器（通常是vim）来让你输入提交的描述信息，如果进入vim的话，



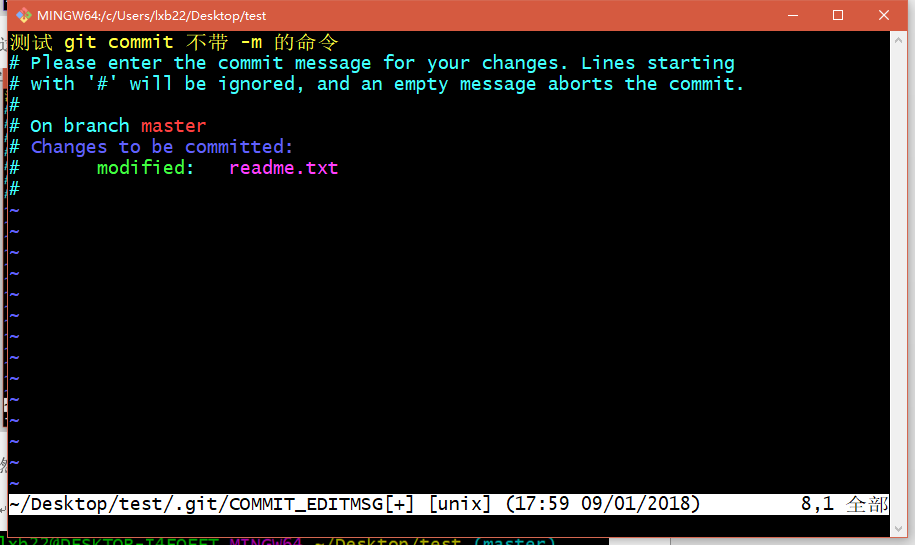
先输入**i**，下方就会变成插入

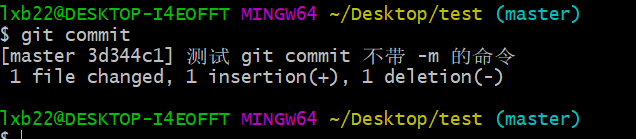


这下就可以输入该次提交的内容描述了



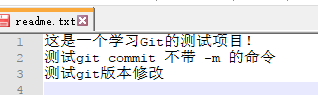
然后按ESC，输入 **:wq** (必须是英文的)就可以退出到命令控制台了





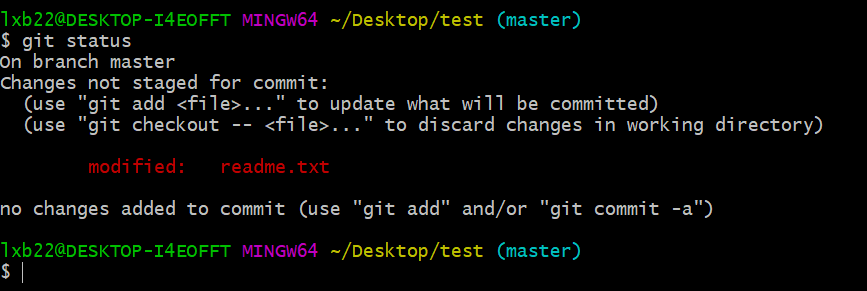
##### 4.2 Git版本修改

我们已经成功地添加并提交了一个readme.txt文件，现在，是时候继续工作了，于是，我们继续修改readme.txt文件，改成如下内容：



现在，运行git status命令看看结果：

$ git status



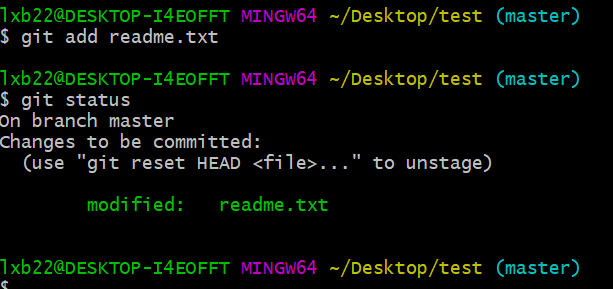
git status命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

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：

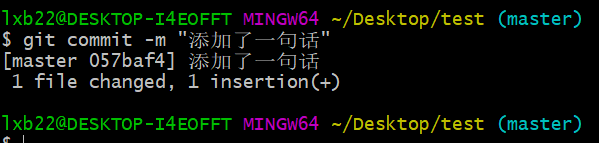
$ git diff readme.txt



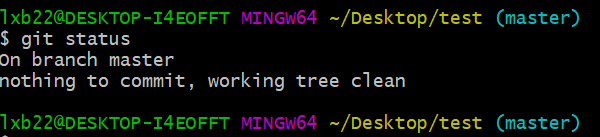
git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在最后一行添加了” 测试git版本回退”的一句话, 知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add，同样没有任何输出。在执行第二步git commit之前，我们再运行git status看看当前仓库的状态：



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：



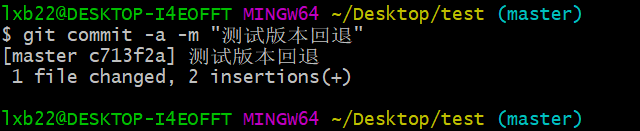
提交后，我们再用git status命令看看仓库的当前状态：



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（working directory clean）的。

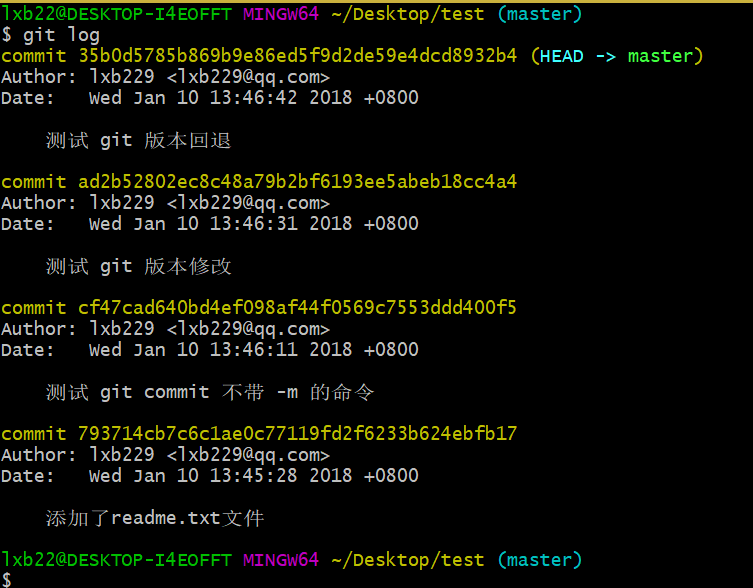
##### 4.3 Git版本回退

现在，你已经学会了修改文件，然后把修改提交到Git版本库，现在，再练习一次，修改readme.txt文件。然后提交：



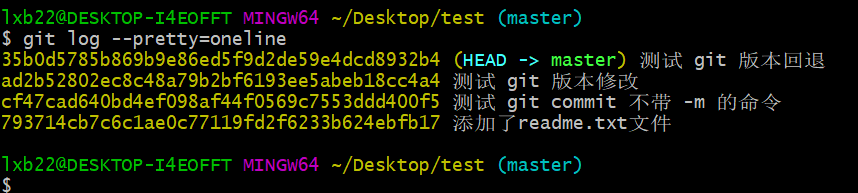
像这样，你不断对文件进行修改，然后不断提交修改到版本库里，每当文件修改的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。在Git中，我们用git log命令查看历史提交记录：

$ git log



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，看得眼花缭乱的，可以试试加上--pretty=oneline参数：

$ git log --pretty=oneline



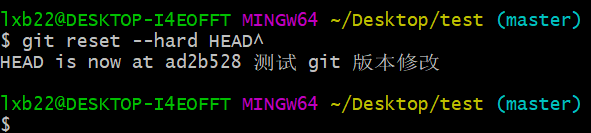
需要友情提示的是，你看到的一大串类似35b0d5...8932b4的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

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35b0d5...8932b4（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。

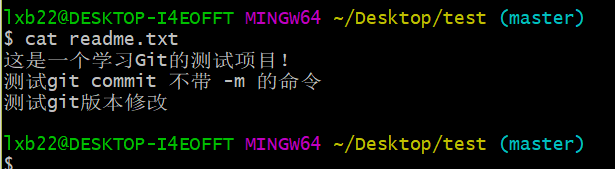
$ git re**set** --hard HEAD~1

现在，我们要把当前版本回退到上一个版本，就可以使用git reset命令：

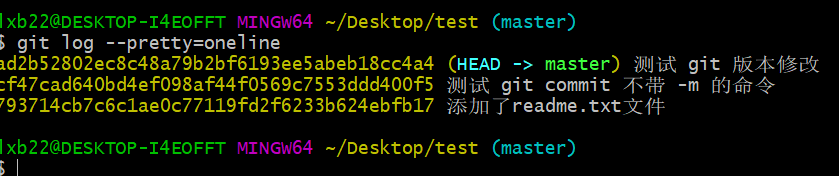
$ git re**set** --hard HEAD**^**



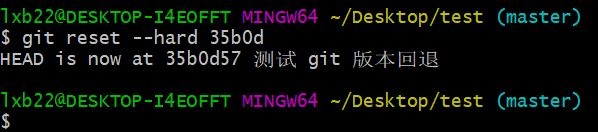
看看readme.txt的内容是不是回退的版本



然后我们用git log再看看现在版本库的状态：



最新的那个版本已经看不到了！那如果我们需要回到最新的版本的话，就需要最新的版本的commit id的 35b0d5...8932b4, 指定回到未来的某个版本：



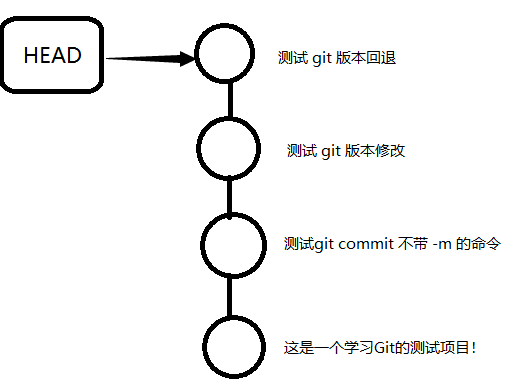
版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

如果找不到新版本的commit id怎么办？Git提供了一个命令git reflog用来记录你的每一次命令：

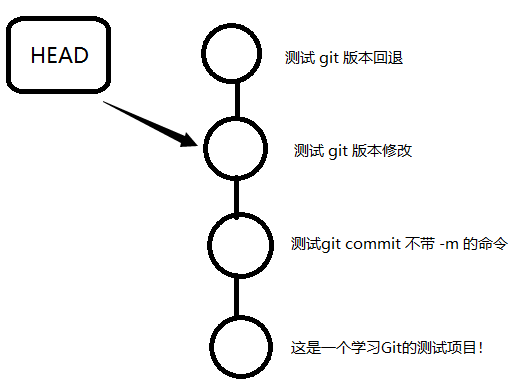


这样就可以找到之前最新的commit id

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 测试 git 版本回退，改为指向测试 git 版本修改：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。



改为：



##### 4.4 工作区和暂存区

###### 工作区（Working Directory）

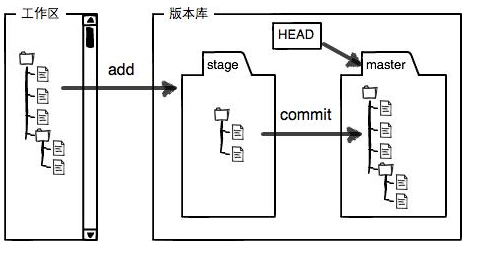
就是你在电脑里能看到的目录，比如我的test文件夹就是一个工作区：

###### 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。

###### 暂存区（Stage）

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；

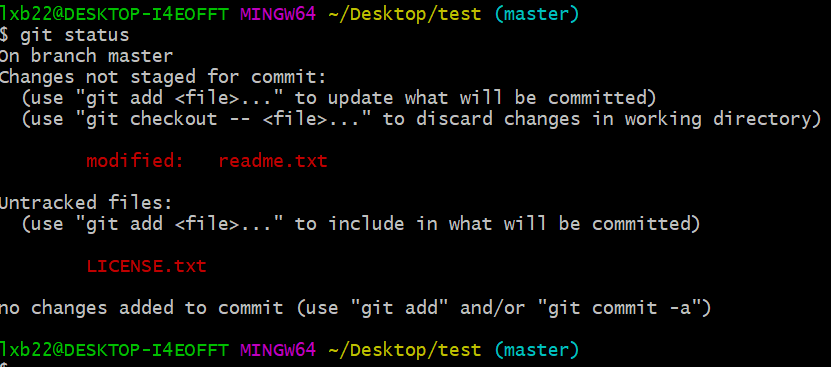
第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

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。

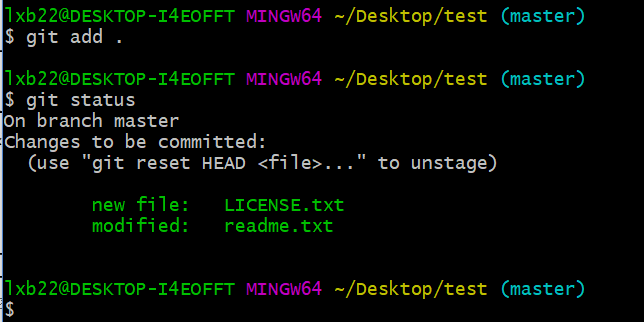
现在，我们再对readme.txt做个修改，然后，在工作区新增一个LICENSE文本文件（内容随便写）。

先用git status查看一下状态：

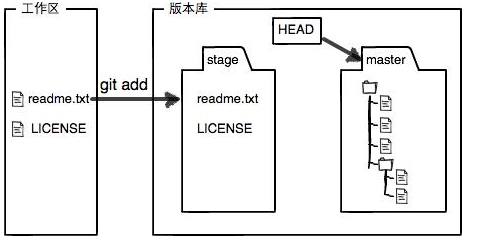


Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

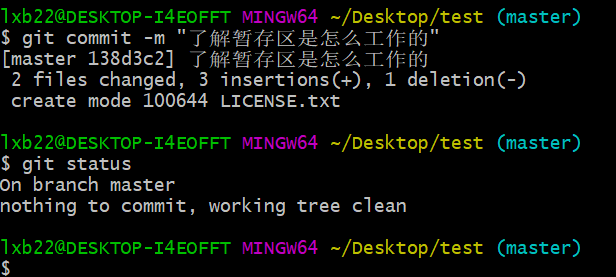
现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：



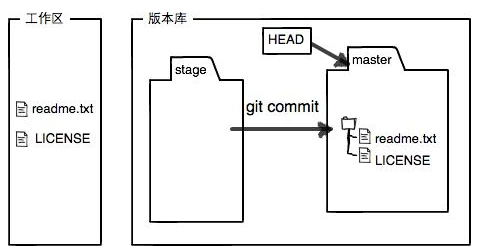
现在，暂存区的状态就变成这样了：



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：



了解了暂存区，那么我么来测试一下，对readme.txt文件进行一些操作：

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

这样的操作后，我么第二次的修改有没有被提交呢？事实是没有！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：

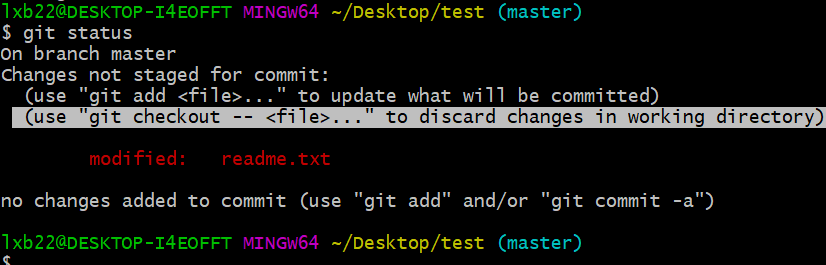
第一次修改 -> git add -> 第二次修改 -> git add -> git commit

##### 4.5 撤销修改

现在我么在readme.txt中添加了一行：



在你准备提交前，发现又不想要这句话，既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用git status查看一下：



你可以发现，Git会告诉你，git checkout -- file可以丢弃工作区的修改：

$ git checkout -- readme.txt

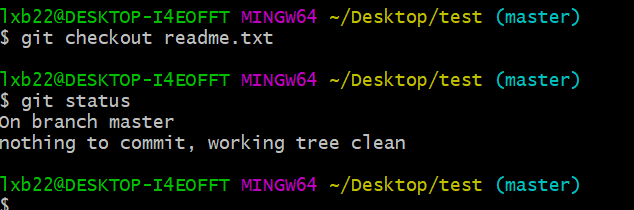
命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

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。

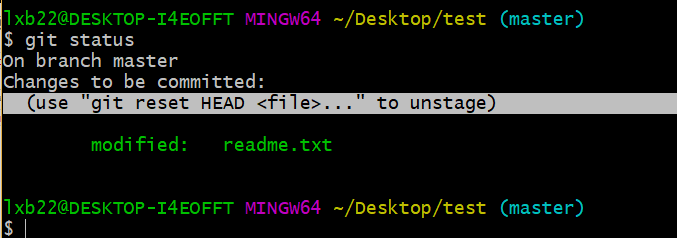
现在，看看readme.txt的文件内容：



文件内容果复原了。

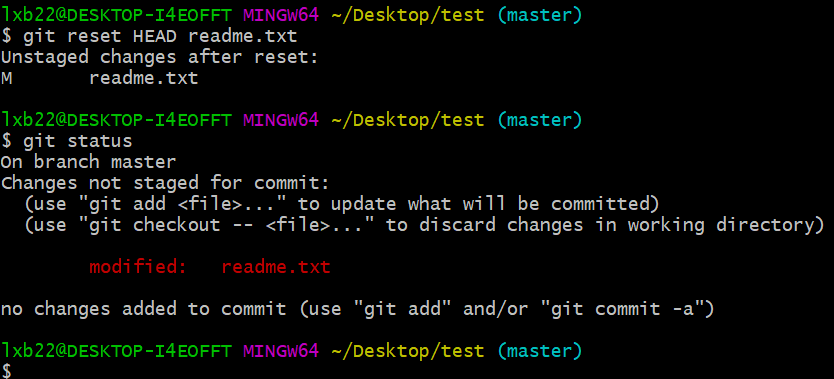
git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

如果我们把readme.txt文件提交到了暂存区，还没有提交：

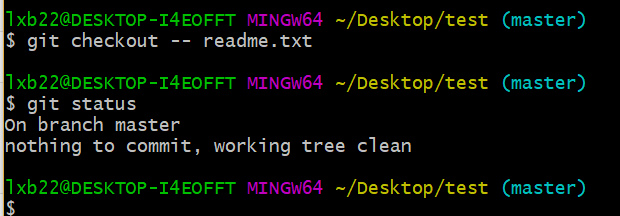


Git同样告诉我们，用命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区，git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

$ git re**set** HEAD file



现在暂存区是干净的，工作区有修改，再用git checkout –- file 对工作区撤销，就可以了！



如果是撤销所有的已经add的文件:

$ git re**set** HEAD .

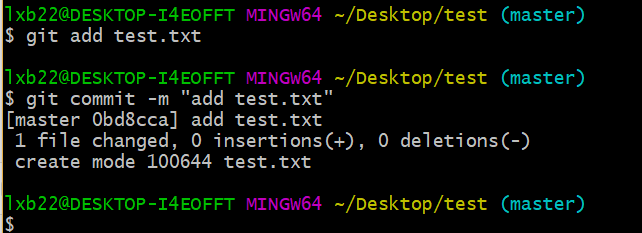
可以看到，这个是重置版本到HEAD，和 git commit add .一样，后面都有一个 . ，这就表示取消所有的add文件。

也可以使用版本回退，回到HEAD

$ git re**set** --hard HEAD

##### 4.6 删除文件

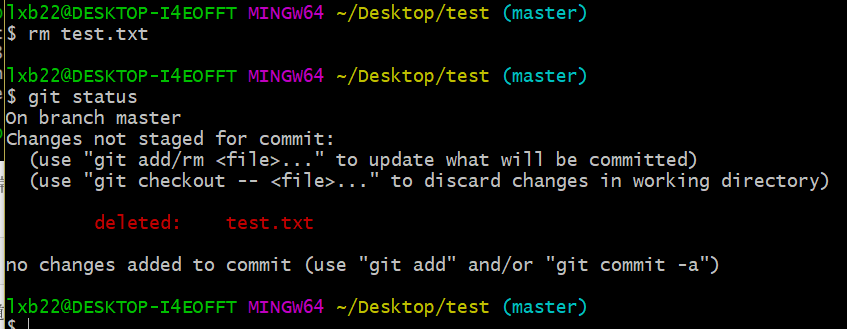
在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：



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：

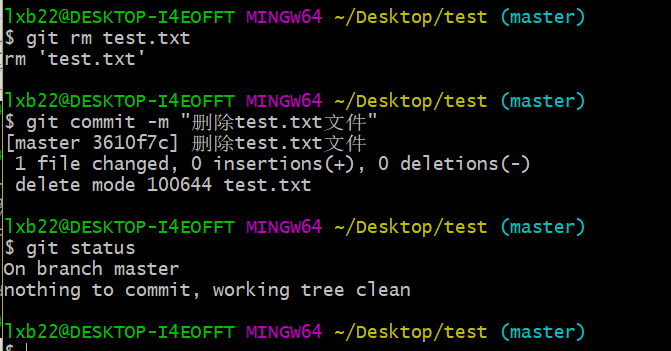
$ rm test.txt

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：

$ git rm test.txt



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

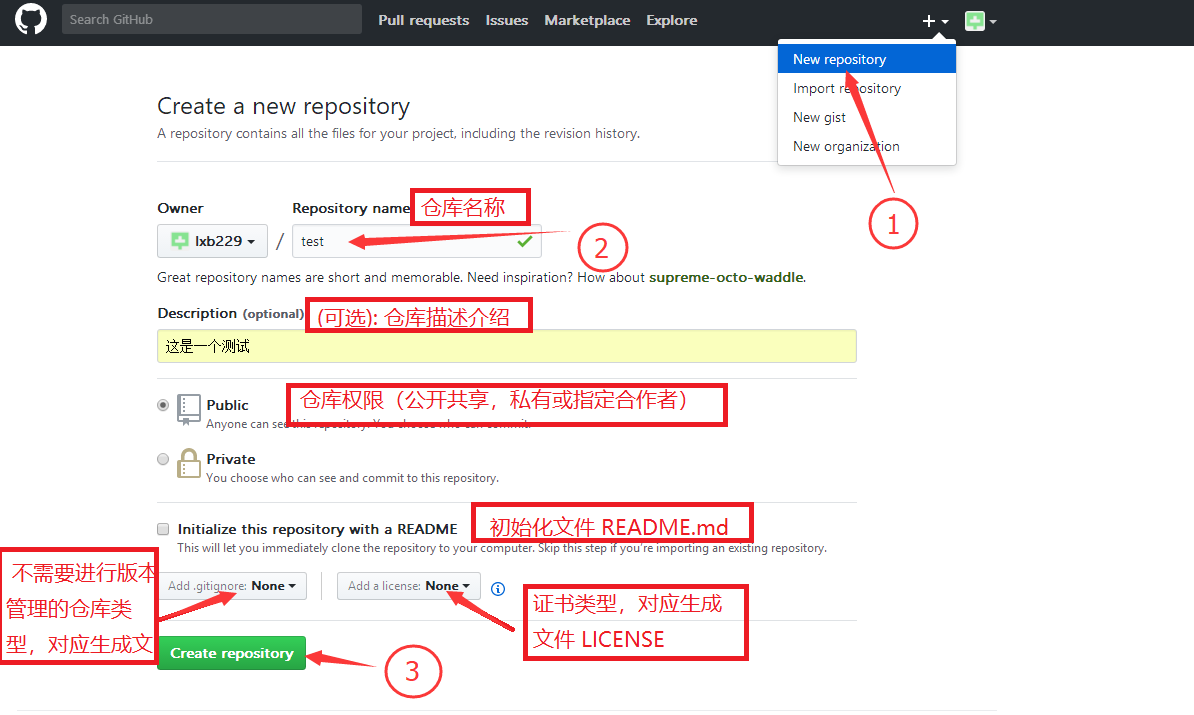
git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

#### 五．远程仓库

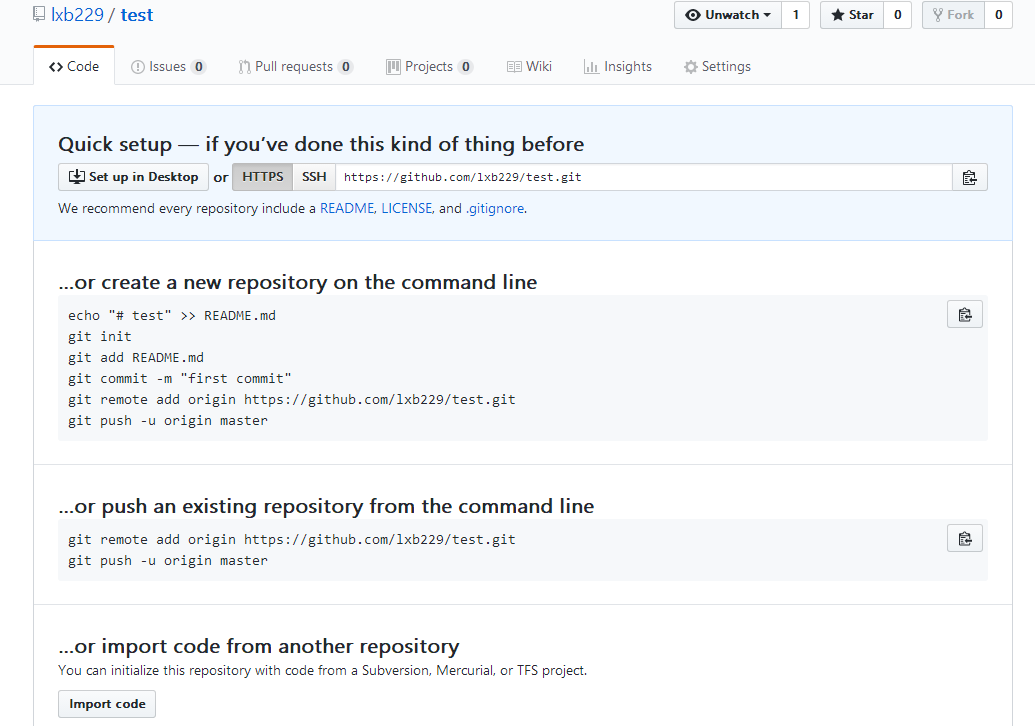
##### 5.1 添加远程库

现在我们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：

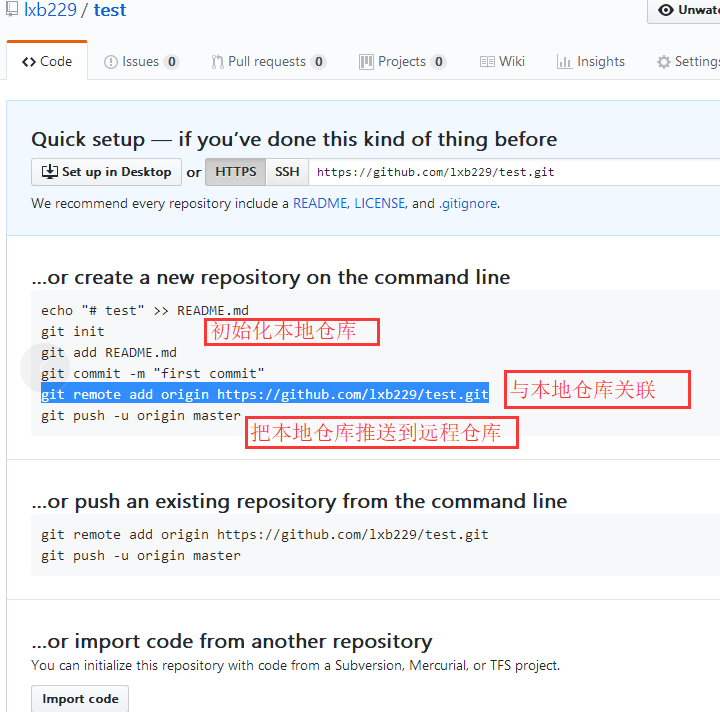


在Repository name填入test，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



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：

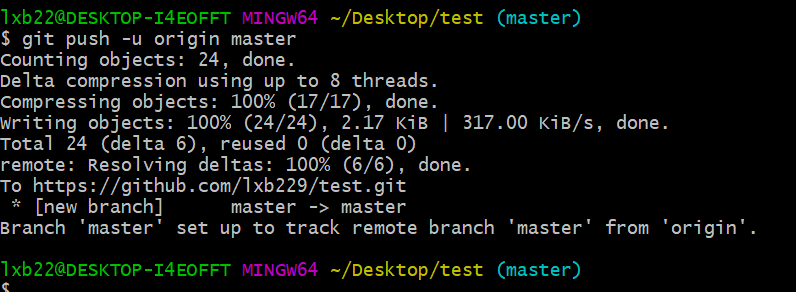


$ git remote add origin 远程仓库地址



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：



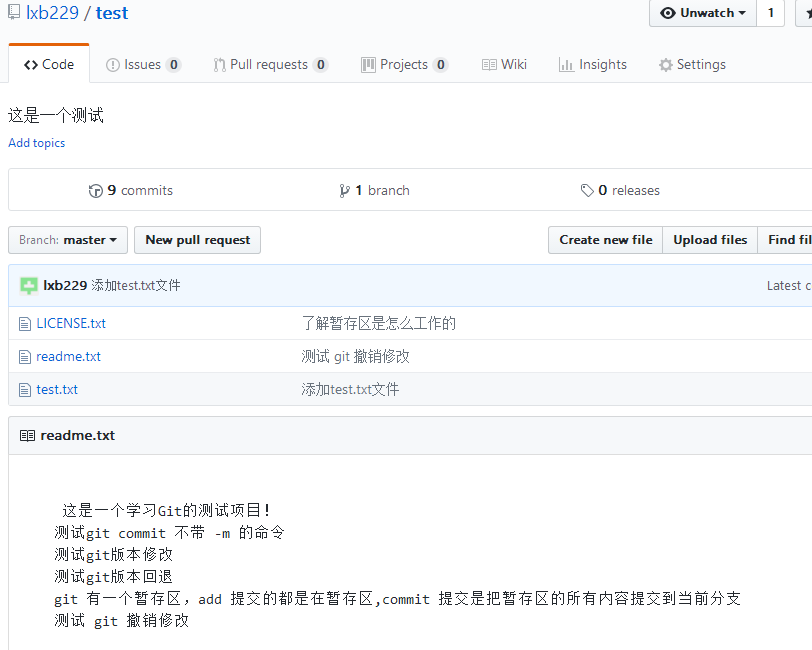
把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

$ git push -u origin master

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:

$ git push origin master

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：

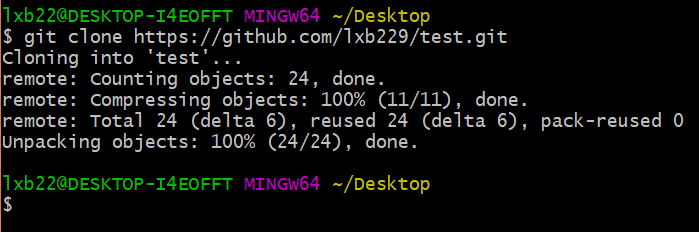


把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

##### 5.2 从远程库克隆

如果我们需要从gitHub远程库上克隆项目，就需要

$ git clone https://github.com/lxb229/test.git

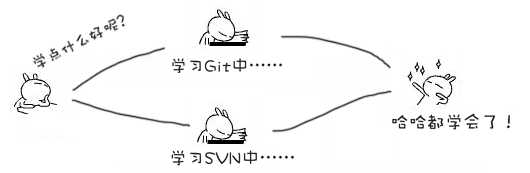


你也许还注意到，GitHub给出的地址不止一个，还可以用git@github.com:lxb229/test.git这样的地址。实际上，Git支持多种协议，默认的git://使用ssh，通过ssh支持的原生git协议速度最快。如果有兴趣的可以去看看！

#### 六． 分支管理

##### 6.1 分支介绍

分支就是项目立项之后，每个人分配到各自的模块独自开发，不和其他人交互，在某个时间点，合并到一起，组成一个完整的项目！就像下图，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN！



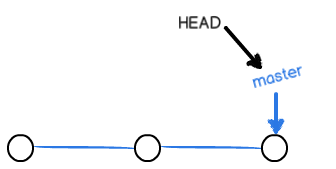
分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

##### 6.2 创建与合并分支

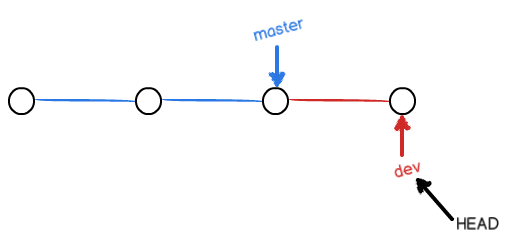
在版本回退里，我们已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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：

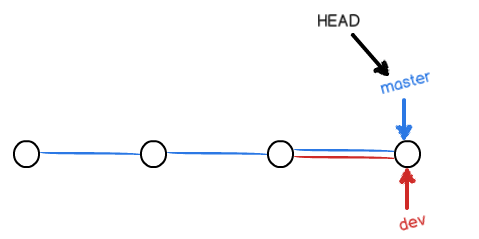


每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：

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：

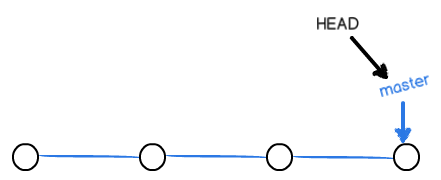


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



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：



下面开始实战。

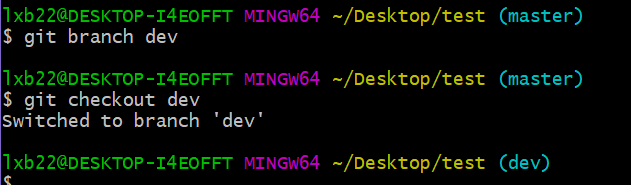
首先，我们创建dev分支，然后切换到dev分支：

$ git checkout –b dev



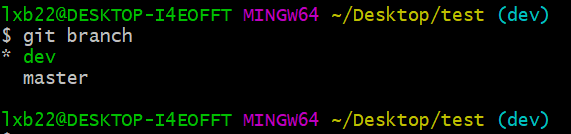
注意:如果你出现fatal: Not a git repository (or any of the parent directories): .git的问题，这说明你当前的路径并不是一个git库，必须进入到库对应的文件夹内，如果是同一个文件夹，那么你需要 git init 来初始化git 的本地仓库。

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

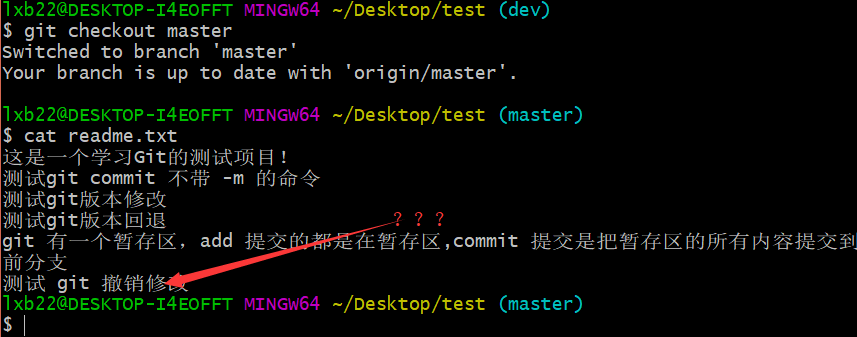


git branch命令会列出所有分支，当前分支前面会标一个\*号。

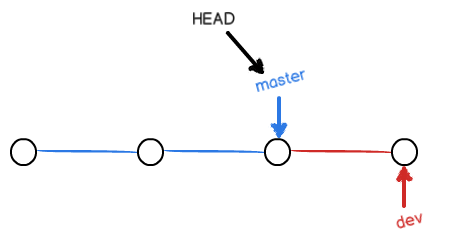
然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行，然后提交：



现在，dev分支的工作完成，我们就可以切换回master分支：

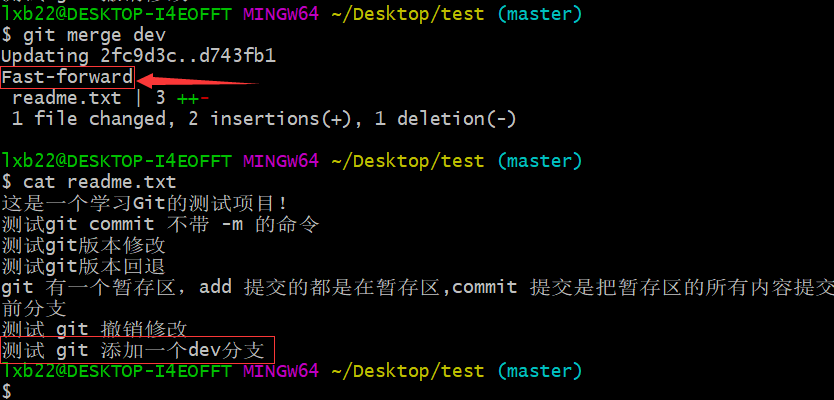


切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev



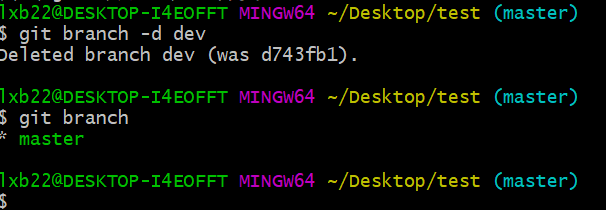
git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

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, 删除后，查看branch，就只剩下master分支了：

$ git branch -d dev



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。

##### 6.3 解决冲突

合并分支往往也不是一帆风顺的，也可能会出现问题。

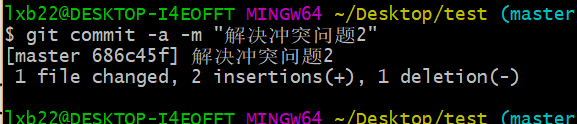
准备新的feature1分支，继续我们的新分支开发：

在readme.txt最后一行添加一句话，在feature1分支上提交，并切换到master分支：

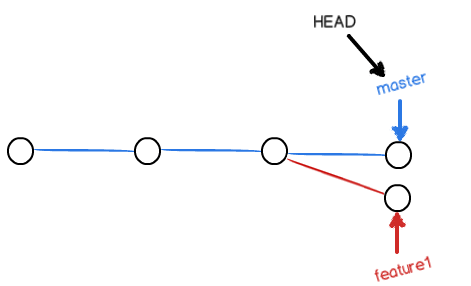


Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。

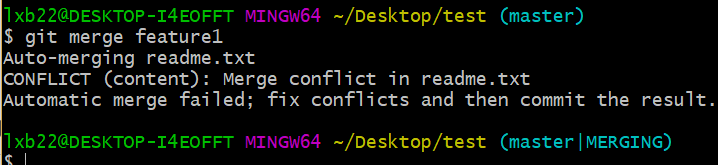
在master分支上把readme.txt文件的最后一行再修改一下提交。



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



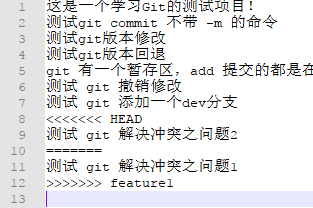
这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突：



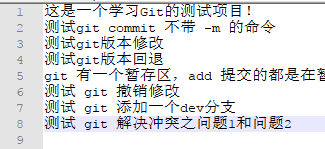
Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：



我们可以直接查看readme.txt的内容：



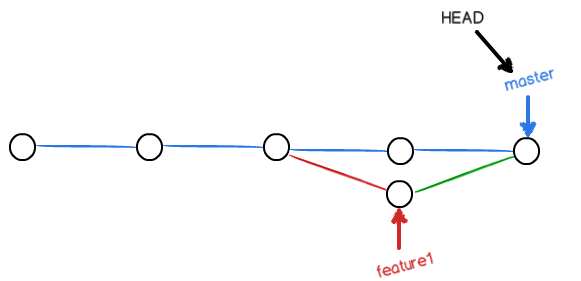
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：



再提交：



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：

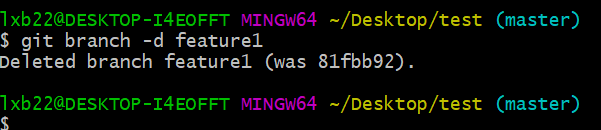
$ git log –graph

或

$ git log --graph --pretty=oneline --abbrev-commit



最后，删除feature1分支：



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。

##### 6.4 分支管理策略

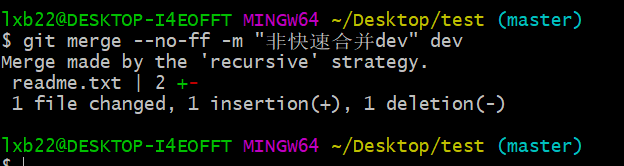
通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

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readme.txt文件，并提交一个新的commit，然后切换回master，准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

$ git merge --no-ff -m "merge with no-ff" dev

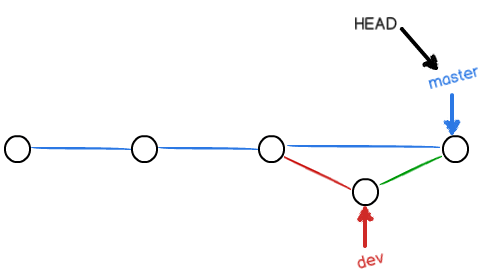


因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史：



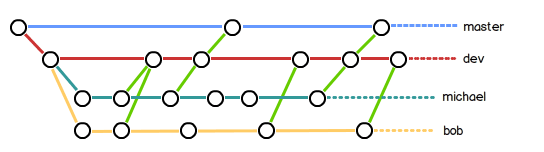
合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。可以看到，不使用Fast forward模式，merge后就像这样：



**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：

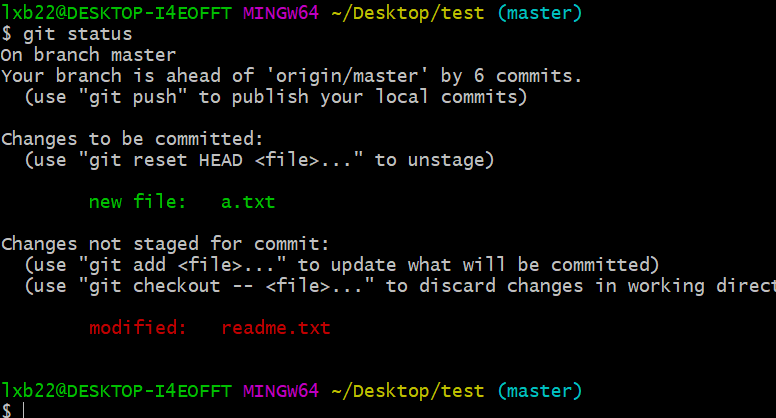
首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；

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；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：

##### 6.5 Bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

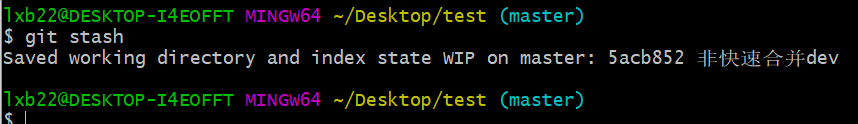
当你接到一个修复一个代号001的bug的任务时，很自然地，你想创建一个分支bug001来修复它，但是，等等，当前正在dev上进行的工作还没有提交：



并不是你不想提交，而是工作只进行到一半，还没法提交, 怎么办？

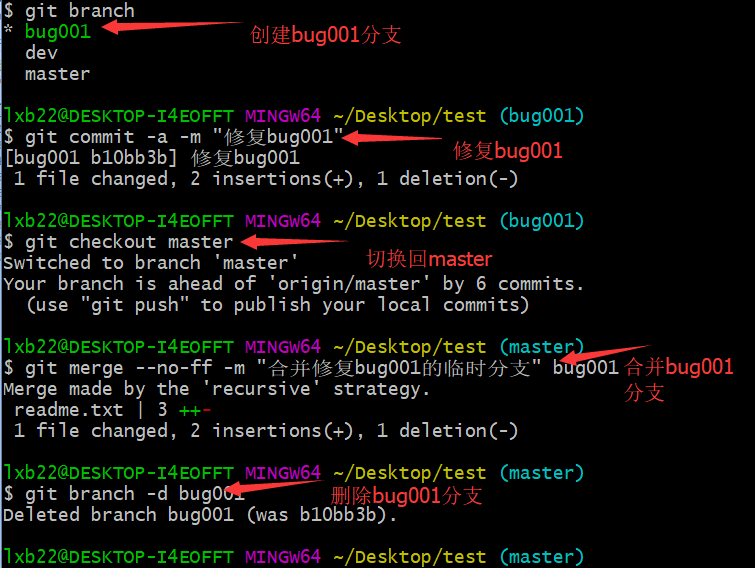
幸好，Git还提供了一个stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

$ git stash



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：



bug修复只花了5分钟！现在，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：

$ git stash list

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；

$ git stash apply

$ git stash drop

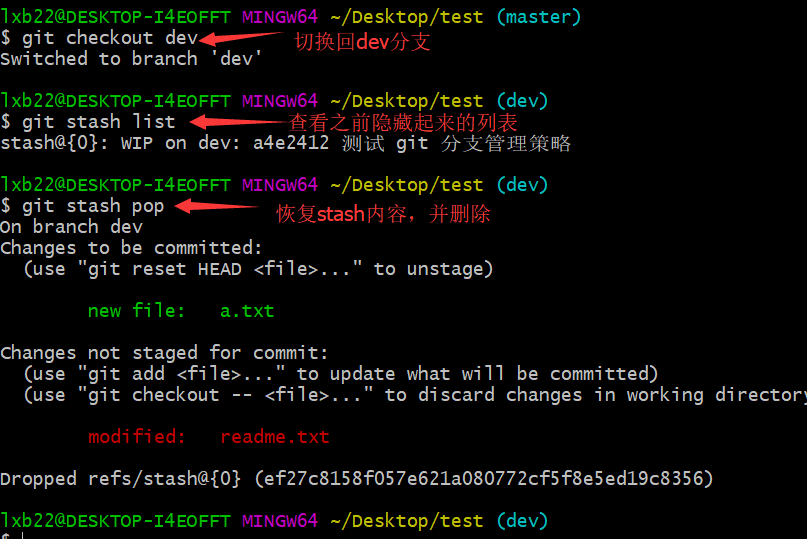
另一种方式是用git stash pop，恢复的同时把stash内容也删了,

再用git stash list查看，就看不到任何stash内容了,

$ git stash pop

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：

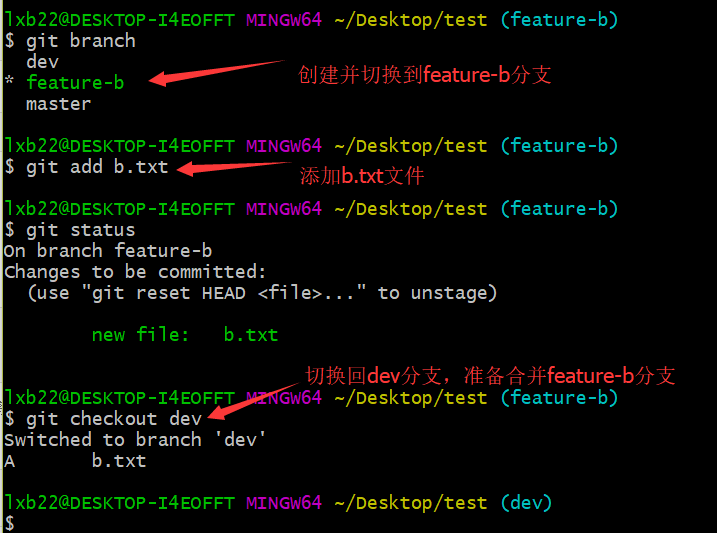
$ git stash apply stash@{0}

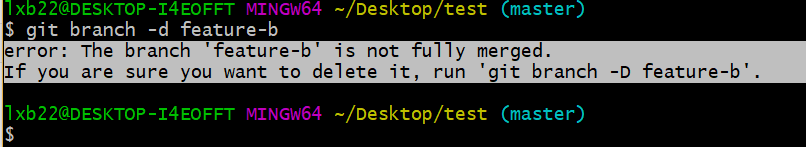


##### 6.6 Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。

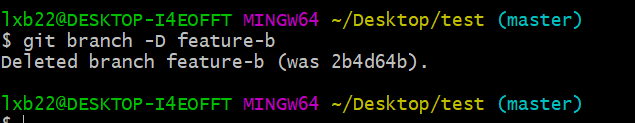
添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。现在，我们需要添加一个b.txt的文件，开发完毕后，切回dev，准备合并，切顺利的话，feature分支和bug分支是类似的，合并，然后删除。但是，就在此时，b.txt不需要了，必须取消！虽然白干了，但是这个分支还是必须就地销毁：





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，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：



##### 6.7 多人协作

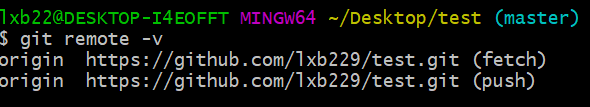
当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：

$ git remote

或者，用git remote -v显示更详细的信息：

$ git remote -v



上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

**推送分支**

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。

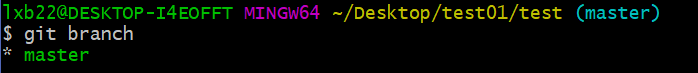
总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

**抓取分支**

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

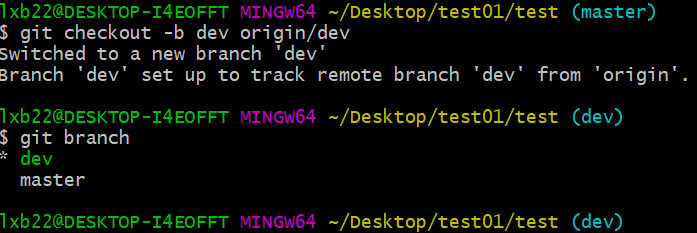
当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：



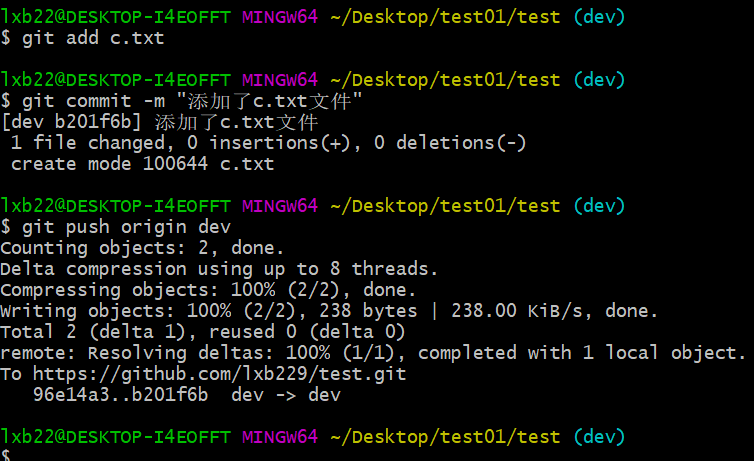
现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

注意：要在分支dev 上开发，就必须先有人进行了dev的分支推送git push origin dev



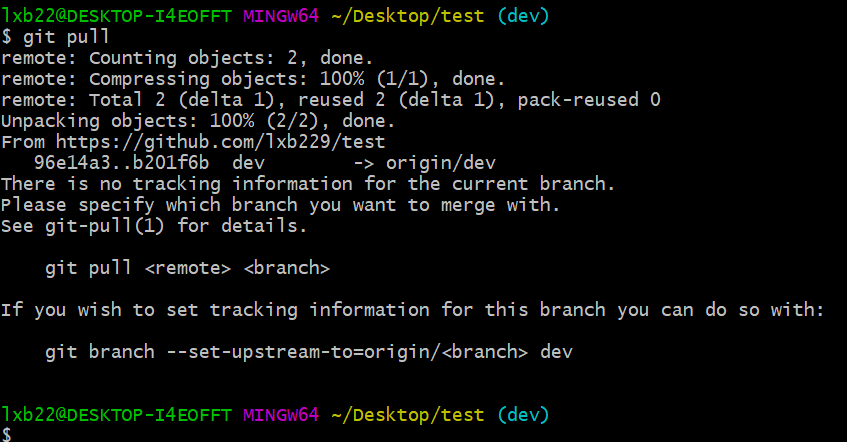
现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：



你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：

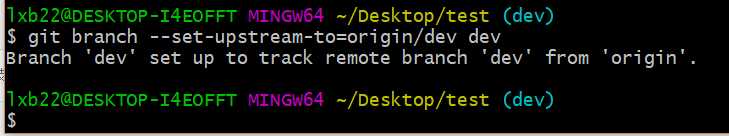


推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：

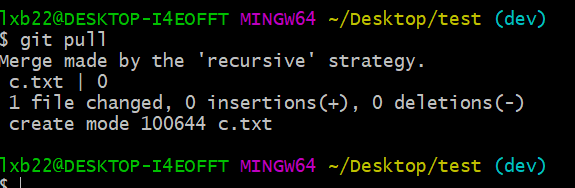


git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

$ git branch –set-upstream-to=origin/dev dev



再pull：



这回git pull成功，如果有时候合并有冲突，那就需要手动解决，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：

因此，多人协作的工作模式通常是这样：

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。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

#### 七． 标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

##### 7.1 **创建标签**

在Git中打标签非常简单，首先，切换到需要打标签的分支上，然后，敲命令git tag <name>就可以打一个新标签，可以用命令git tag查看所有标签：

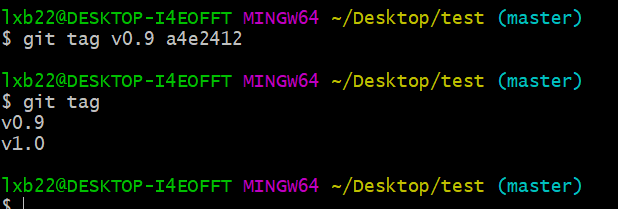
$ git tag v1.0



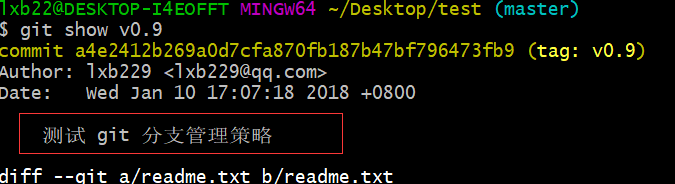
默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了，比方说要对测试 git 分支管理策略这次提交打标签，它对应的commit id是a4e2412，敲入命令：

$ git tag v0.9 a4e2412



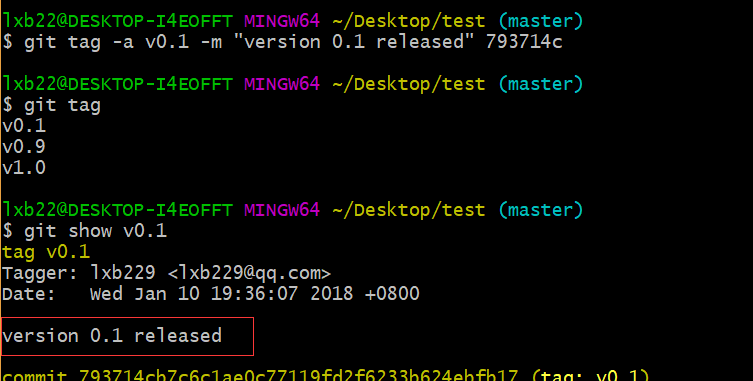
注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：



可以看到，v0.9确实打在测试 git 分支管理策略这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v0.1 -m "version 0.1 released" 3628164



##### 7.2 **操作标签**

如果标签打错了，也可以删除：

$ git tag -d v0.1

不会自动推送到远程。所以，打错的标签可以在本地安全删除。如果要推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

或者，一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

$ git tag -d v0.9

然后，从远程删除。删除命令也是push，但是格式如下：

$ git push origin :refs/tags/v0.9

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

#### 八． 自定义Git

在安装Git一节中，我们已经配置了user.name和user.email，实际上，Git还有很多可配置项。比如，让Git显示颜色，会让命令输出看起来更醒目：

$ git config --global color.ui **true**

**配置别名**

比如git status我们只需要敲一行命令，告诉Git，以后st就表示status：

$ git config --global **alias**.st status

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

$ git config --global **alias**.co checkout

$ git config --global **alias**.ci commit

$ git config --global **alias**.br branch

以后提交就可以简写成：

$ git ci -m "bala bala bala..."

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在撤销修改一节中命令git reset HEAD file可以把暂存区的修改撤销掉（unstage），重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

$ git config --global **alias**.unstage 'reset HEAD'

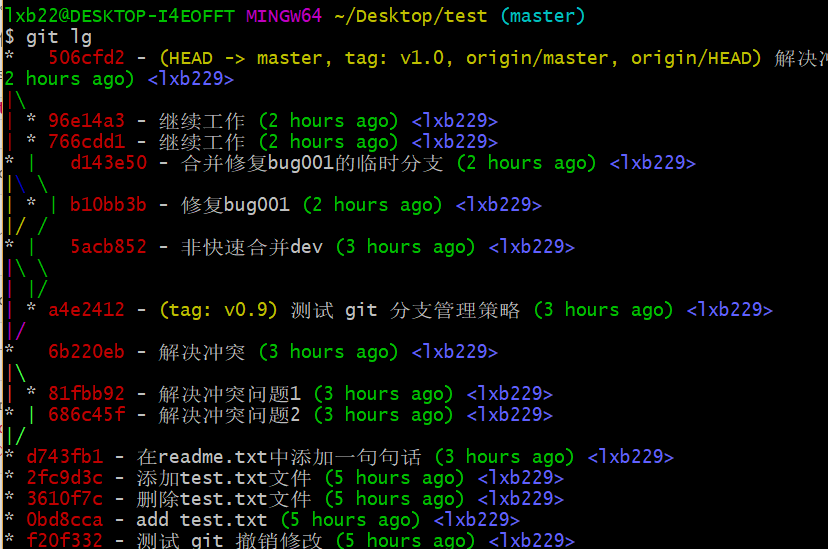
配置一个git last，让其显示最后一次提交信息：

$ git config --global **alias**.last 'log -1'

最后一个非常棒的配置，日志的命令配置lg

git config --global **alias**.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

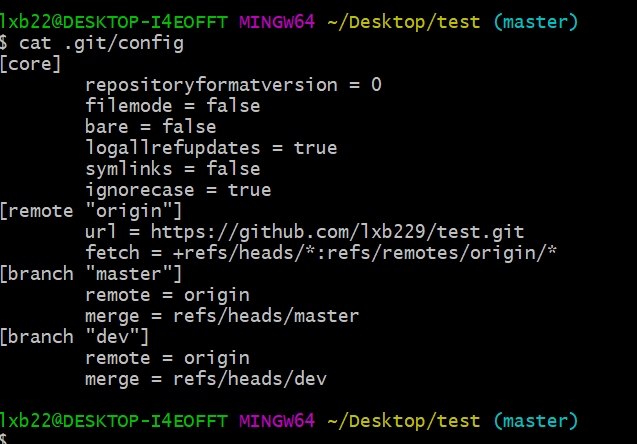
来看看git lg的效果：



**配置文件**

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中：



config 配置指令

$ git config --global

config 配置有system级别 global（用户级别） 和local（当前仓库）三个 设置先从system-》global-》local  底层配置会覆盖顶层配置 分别使用--system/global/local 可以定位到配置文件

查看系统config

$ git config --system --list

查看当前用户（global）配置

$ git config --global --list

查看当前仓库配置信息

$ git config –local –list