# git

## 版本控制

### 版本控制工具应该具备的功能

协同修改

多人并行不悖的修改服务器端的同一个文件。

数据备份

不仅保存目录和文件的当前状态，还能够保存每一个提交过的历史状态。

版本管理

在保存每一个版本的文件信息的时候要做到不保存重复数据，以节约存储空间，提高运行效率。这方面SVN采用的是增量式管理的方式，而Git采取了文件系统快照的方式。

权限控制

对团队中参与开发的人员进行权限控制。对团队外开发者贡献的代码进行审核——Git独有。

历史记录

查看修改人、修改时间、修改内容、日志信息。将本地文件恢复到某一个历史状态。

分支管理

允许开发团队在工作过程中多条生产线同时推进任务，进一步提高效率。

## 版本控制简介

### 版本控制

工程设计领域中使用版本控制管理工程蓝图的设计过程。在IT开发过程中也可以

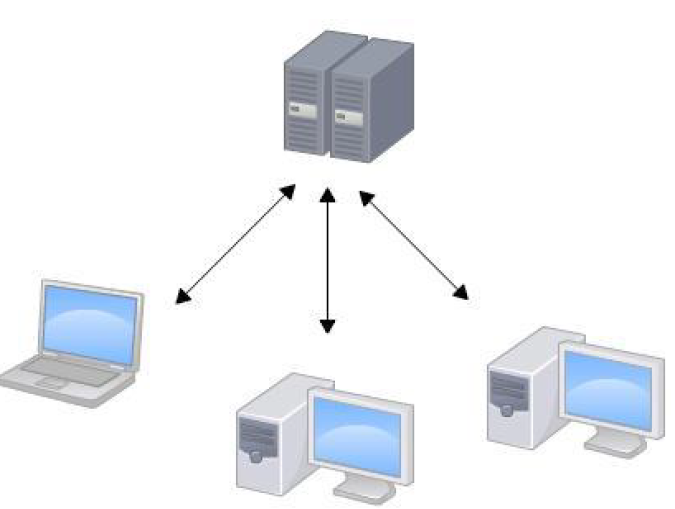
使用版本控制思想管理代码的版本迭代。

### 版本控制工具

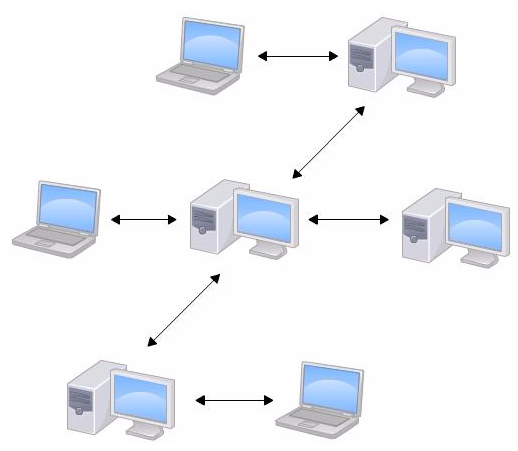
思想：版本控制

实现：版本控制工具

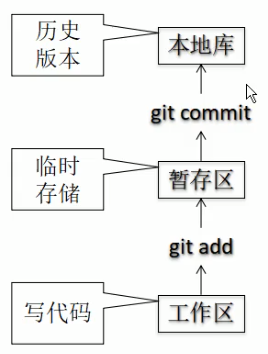
集中式版本控制工具：CVS、SVN、VSS……



分布式版本控制工具：Git、Mercurial、Bazaar、Darcs……

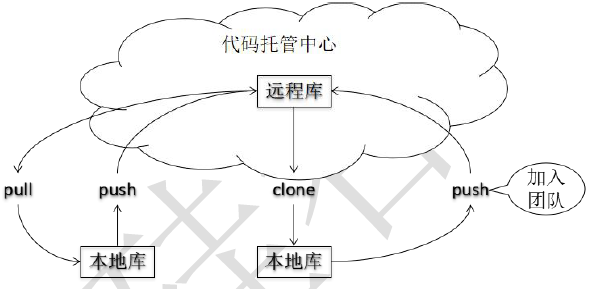


## git的结构

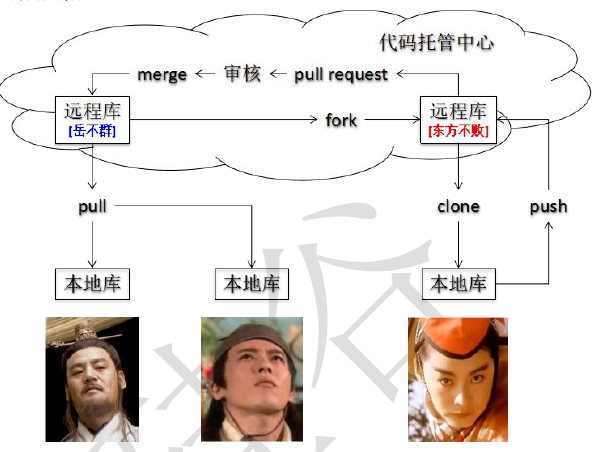


## 本地库和远程库的交互

### 团队协作

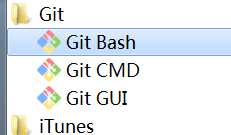


### 跨团队协作

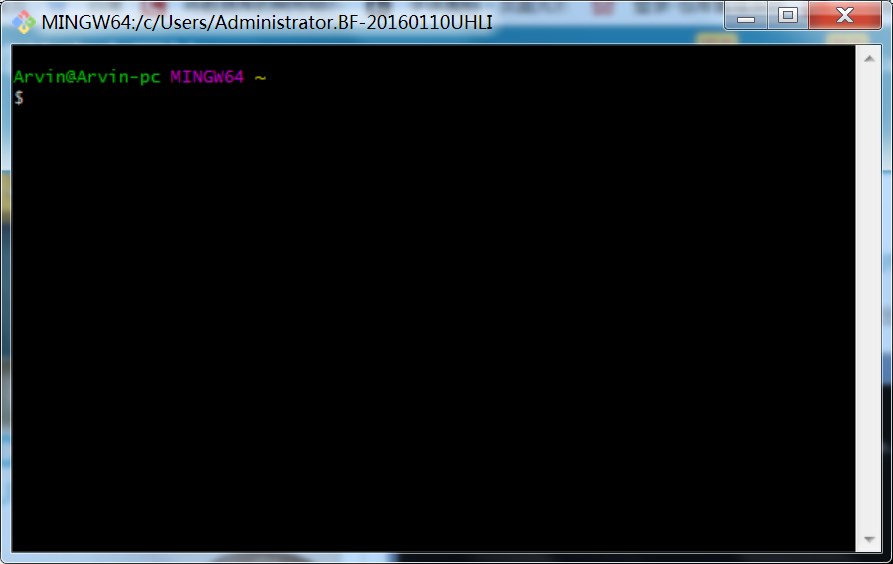


## 设置全局身份

### 打开git Bash



如果出现以下界面说明安装成功了（除非脸黑，否则是没有任何问题的）



### 设置签名

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。

形式：

作用：区分不同开发人员的身份

辨析：这里设置的签名和登录远程库(代码托管中心)的账号、密码没有任何关系。

命令

项目级别/仓库级别：仅在当前本地库范围内有效

 git config user.name tom\_pro

 git config user.email goodMorning\_pro@atguigu.com

 信息保存位置：./.git/config 文件

系统用户级别：登录当前操作系统的用户范围

 git config --global user.name tom\_glb

 git config --global goodMorning\_pro@atguigu.com

 信息保存位置：~/.gitconfig 文件（当前用户文件夹下）

 级别优先级

 就近原则：项目级别优先于系统用户级别，二者都有时采用项目级别

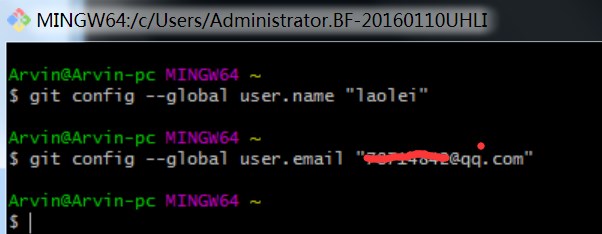
的签名。

 如果只有系统用户级别的签名，就以系统用户级别的签名为准。

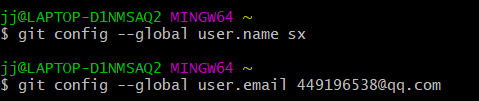
 二者都没有不允许。会报错。

设置项目级别

|  |
| --- |
| $ git config --global user.name "Your Name"  $ git config --global user.email "email@example.com" |

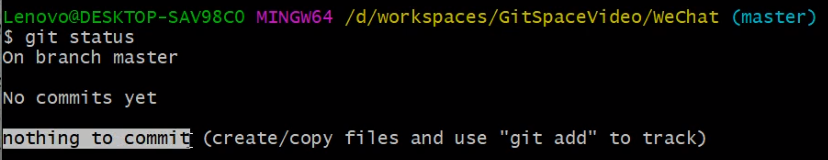


设置系统级别



## 查看状态

语法：git status



在master 分支，还没有提交。缓存区还没有提交，创建文件使用add提交到缓存区，是git能够追踪文件。

## 添加文件

现在我们编写一个readme.txt文件，内容如下：

锄禾日当午;

一定要放到D:\git\repository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。

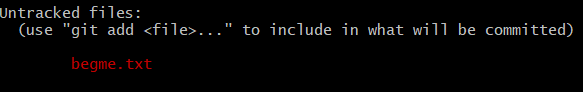
把一个文件放到Git仓库只需要两步。

第一步，用命令git add告诉Git，把文件添加到暂存区：

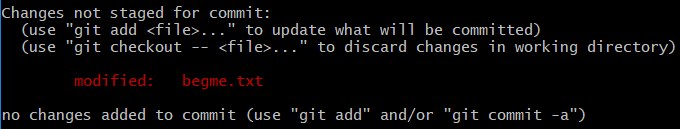
$ git add readme.txt

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。

如果是未追踪的文件：只能使用 git add 是使文件变为追踪状态

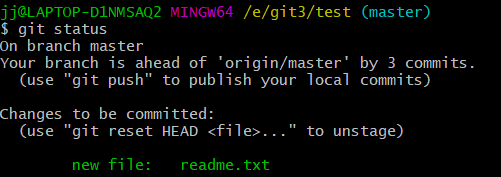


如果是已经追踪的文件：可以使用git commit 提交



第二部：使用status查看状态

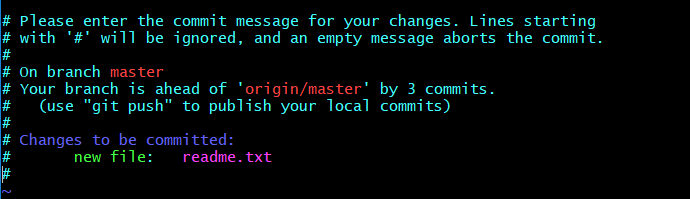
$ git status



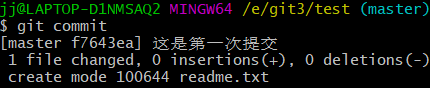
状态显示为绿色，等待被提交。

第三步：将文件提交到本地库

命令：git commit

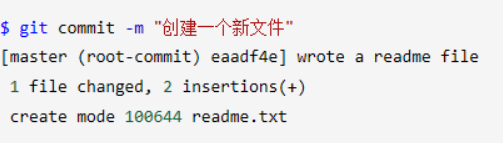


进入vim编辑器中，编辑提交的注释。



第二种提交方式：不用进入vim编辑器，直接写操作注释。

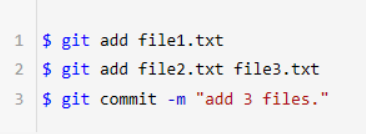
格式：git commit -m “创建一个新文件”



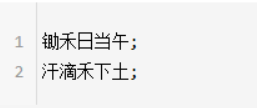
简单解释一下git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

git commit命令执行成功后会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：插入了两行内容（readme.txt有两行内容）。

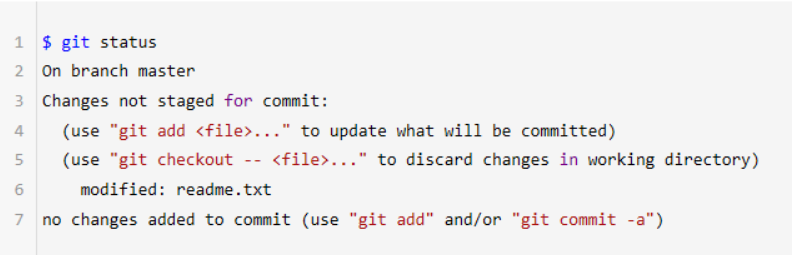
为什么Git添加文件需要add，commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：



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：

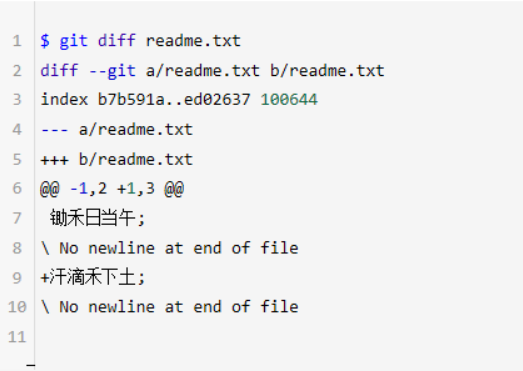


现在，运行git status命令看看结果：



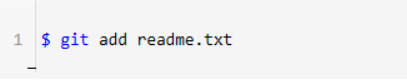
git status命令可以让我们时刻掌握仓库当前的状态，上面的命令输出告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

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gitdiff这个命令看看：

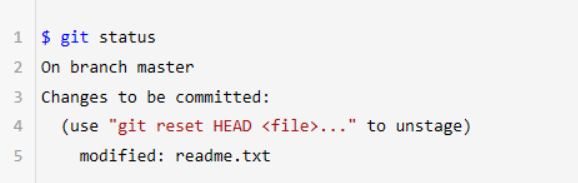


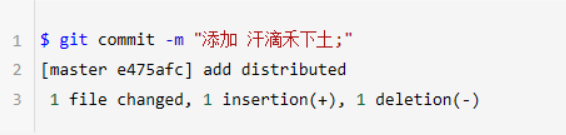
git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第二行添加了一个 "汗滴禾下土;"。

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：





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 tree clean）的。

添加文件到Git仓库，分两步：

2使用命令git add <file>，注意，可反复多次使用，添加多个文件；

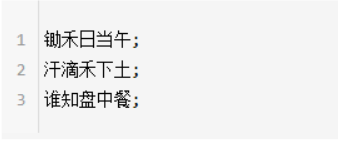
3使用命令git commit -m <message>，完成。

4要随时掌握工作区的状态，使用git status命令。

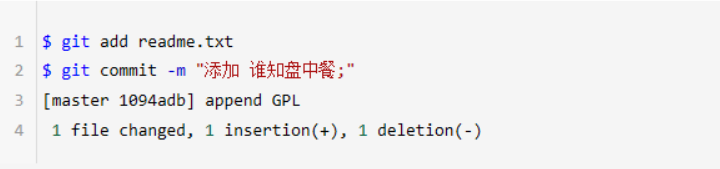
5如果git status告诉你有文件被修改过，用git diff可以查看修改内容

## 版本操作

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文件如下



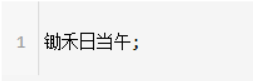
然后尝试提交：



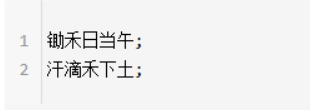
像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在，我们回顾一下readme.txt文件一共有几个版本被提交到Git仓库里了：

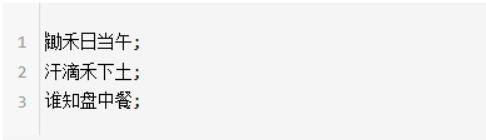
版本1：创建一个新文件



版本2：添加 汗滴禾下土;



版本3：添加 谁知盘中餐;

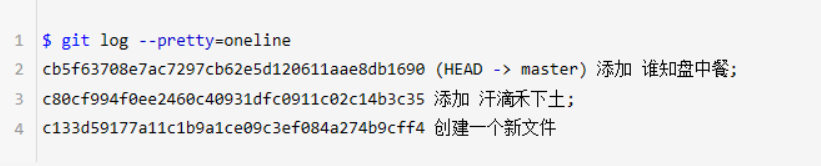


当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log命令查看：



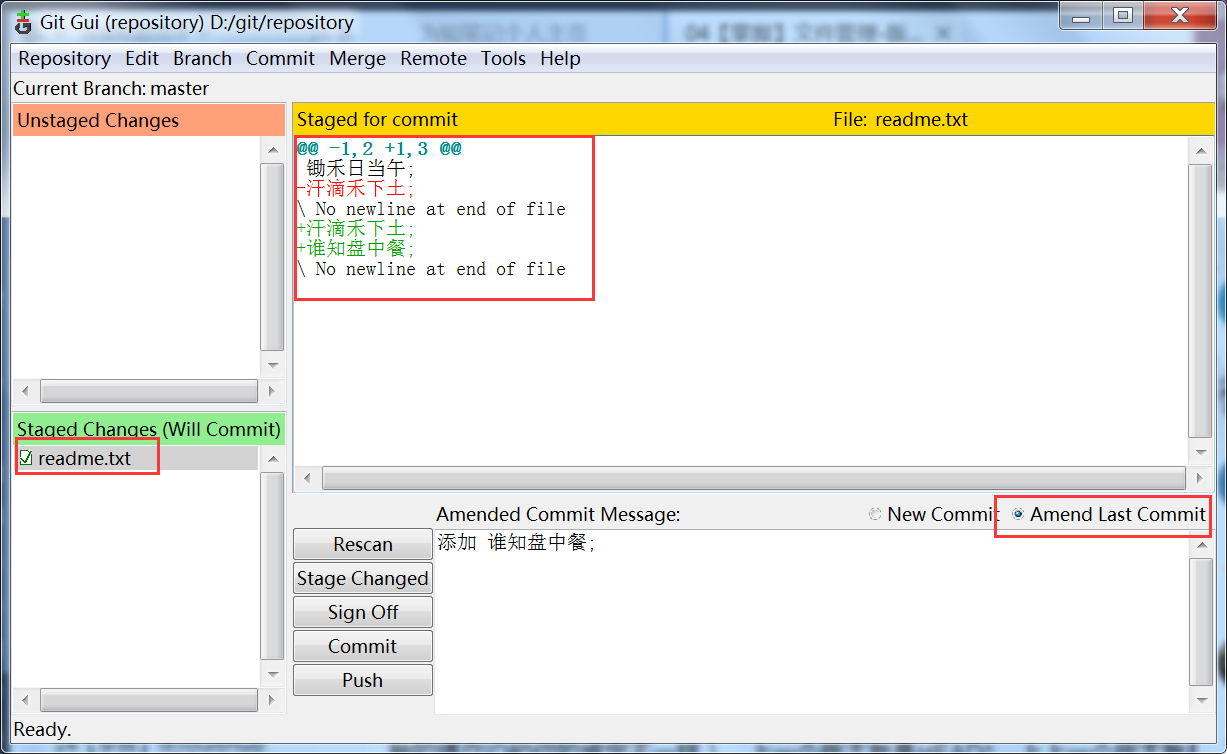
git log命令显示从最近到最远的提交日志，我们可以看到3次提交，最近的一次是append GPL，上一次是add distributed，最早的一次是 创建一个新文件。

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：



需要友情提示的是，你看到的一大串类似cb5f63...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。

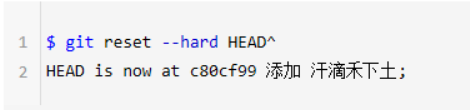
每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线：



好了，现在我们启动时光穿梭机，准备把readme.txt回退到上一个版本，也就是[添加 汗滴禾下土 ]的那个版本，怎么做呢？

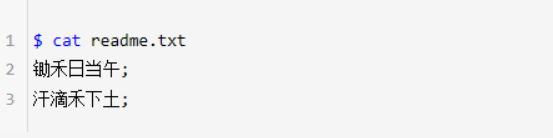
首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交cb5f63708e7...（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

现在，我们要把当前版本【添加 谁知盘中餐;】回退到上一个版本【添加 汗滴禾下土;】，就可以使用git reset命令：



--hard参数有啥意义？这个后面再讲，现在你先放心使用。

看看readme.txt的内容是不是版本 [添加 汗滴禾下土]：



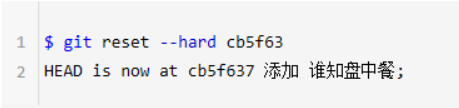
果然被还原了。

还可以继续回退到上一个版本【创建一个新文件】，不过且慢，然我们用git log再看看现在版本库的状态：注：git log 只能显示以前的版本。



最新的那个版本【添加 谁知盘中餐;】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【添加 谁知盘中餐;】的commit id是cb5f63708e7...，于是就可以指定回到未来的某个版本：



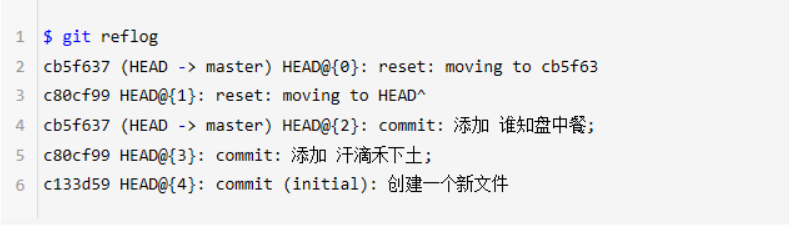
果然，我胡汉三又回来了。

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【添加 谁知盘中餐;】：git-head 改为指向【添加 汗滴禾下土;】：git-head-move

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。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？找不到新版本的commit id怎么办？

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 汗滴禾下土;】版本时，再想恢复到【添加 谁知盘中餐;】，就必须找到【添加 谁知盘中餐;】的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：



终于舒了口气，从输出可知，添加 谁知盘中餐;】的commit id是cb5f637，现在，你又可以乘坐时光机回到未来了。

小结

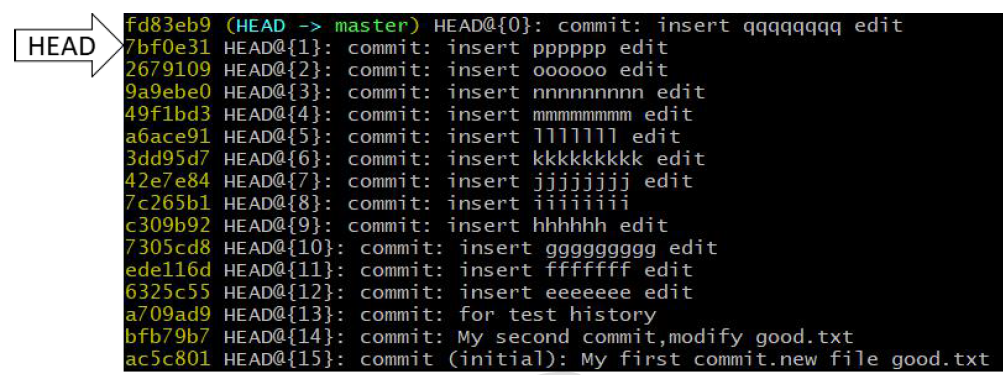
现在总结一下：

HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。

穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。

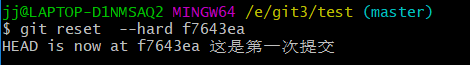
要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

### 基于索引值操作



格式：git reset --hard [局部索引值]

注：并不用去全部打出索引值。



### 使用^符号：只能后退

git reset --hard HEAD^

注：一个^表示后退一步，n 个表示后退n 步

### 使用~符号：只能后退

git reset --hard HEAD~n

注：表示后退n 步

### reset命令删除

--soft参数：

在本地库移动HEAD 指针

--mixed参数

在本地库移动HEAD指针

重置暂存区

--hard参数

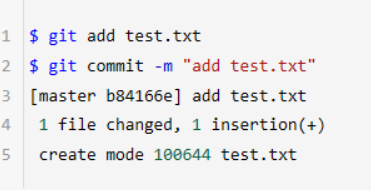
在本地库移动HEAD指针

重置暂存区

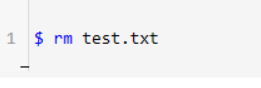
重置工作区

## 文件删除

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：



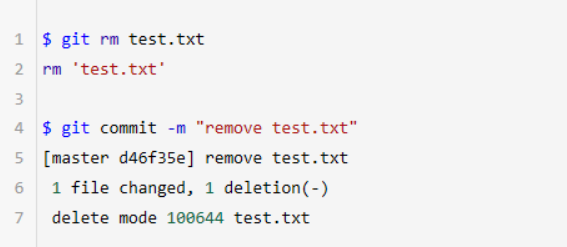
一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：



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：



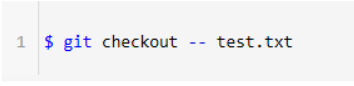
现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：



现在，文件就从版本库中被删除了。

小提示：先手动删除文件，然后使用git rm <file>和git add<file>效果是一样的。

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。

## 比较文件

格式：git duff[文件名]：将工作区中的文件和暂存区进行比较

## git的分支