# 总论

HEAD 文件、（尚待创建的） index 文件，和 objects 目录、 refs 目录。这些条目是 Git 的核心组成部分。 objects 目录存储所有数据内容；refs 目录存储指向数据（分支）的提交对象的指针； HEAD 文件指示目前被检出的分支； index 文件保存暂存区信息。

# 工作区、暂存区和版本库

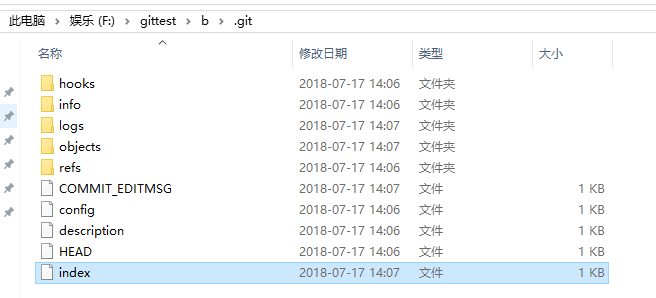
## 工作区

就是你在电脑里能看到的目录，执行git init的目录



## 暂存区

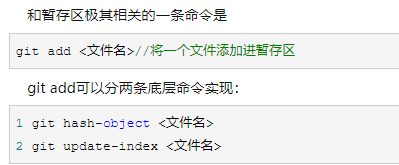
1. 所谓的暂存区只是一个简单的索引文件而已。



（2）暂存区这个索引文件里面包含的是文件的目录树，像一个虚拟的工作区，在这个虚拟工作区的目录树中，记录了文件名、文件的时间戳、文件长度、文件类型以及最重要的SHA-1值，文件的内容并没有存储在其中，所以说 它像一个虚拟的工作区。

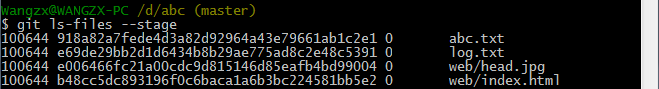
（3）索引指向的是.Git/objects下的文件。

（4）暂存区的作用：除非是绕过暂存区直接提交，否则Git想把修改提交上去，就必须将修改存入暂存区最后才能commit。每次提交的是暂存区所对应的文件快照。

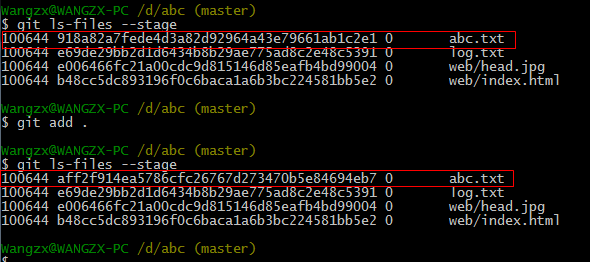


运行第一条命令，Git将会根据新生成的文件产生一个长度为40的SHA-1哈希字符串，并在.git/objects目录下生成一个以该SHA-1的前两个字符命名的子目录，然后在该子目录下，存储刚刚生成的一个新文件，新文件名称是SHA-1的剩下的38个字符。第二条命令将会更新.git/index索引，使它指向新生成的objects目录下的文件。

Index存储多个entry，每个entry存储了被add的文件的hash值和path，index（索引）是一个存放了已排序的路径的二进制文件，并且每个路径都对应一个SHA1哈希值。在Git系统中，可以通过“git ls-files --stage”来显示index文件的内容：



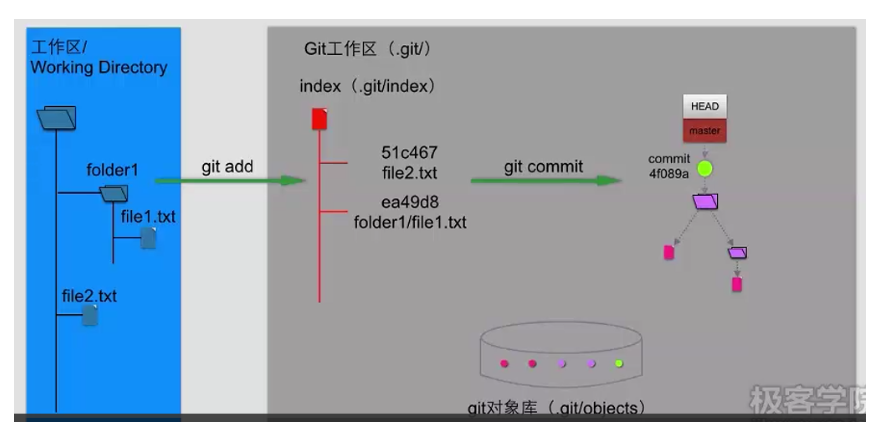
从命令的输出可以看到，所有的记录都对应仓库中的文件（包含全路径）。上面显示的哈希值就是abc.txt的blob对象的哈希值。现在我们更新abc.txt文件，并通过“git add”添加到暂存区，这时发现index中的abc对象的哈希值已经变化了。



在第一次add都还没有进行的时候，我们的暂存区还没有被创建出来，git的对象库中也是空的。当我们执行git add的时候，.git目录下面多了个index文件，这个index文件就是我们所说的暂存区，每一个索引都有一个40位的16进制SHA-1 Hash以及他对应的文件名，这里我们就截取6位，除了这两个之外，索引中还包含了他的文件模式和权限和stage number （表示他的合并状态），时间戳等等。

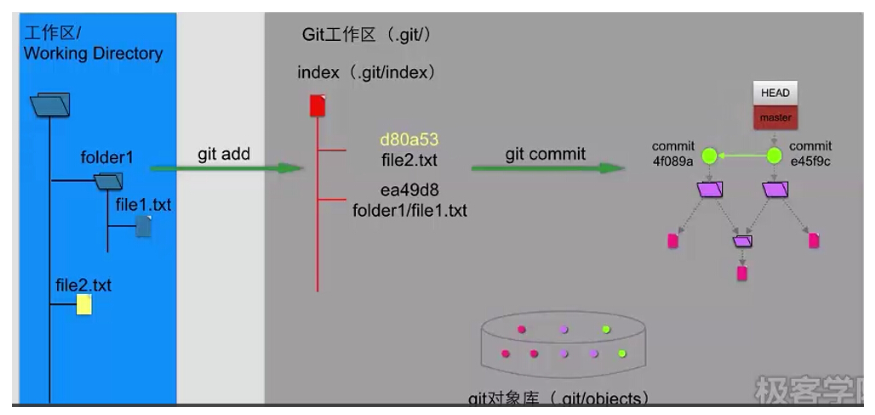
Index的索引通过hash值，和对象库中的blob对象与文件名相对应，这个index文件中除了一些索引之外他还维护了一些提前计算好的tree对象的内容（也就是工作区中的一个个tree对象），在我们要提交的时候，他就可以很迅速的根据我们已经提前算好的内容生成一个tree对象然后添加到历史记录里。

比如下图，当我们做git commit的时候生成了两个tree对象，这时git对象库中有四个对象，然后生成了一个commit对象指向我们顶层的tree对象，我们的分支名以及head标签就指向了这个commit对象：表示我们处于master分支，而且master的最后一个提交是4f809a这样的一个commit。

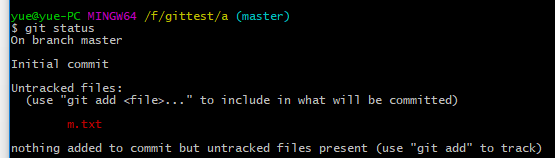


上图的两个tree对象都是在commit的时候生成的。

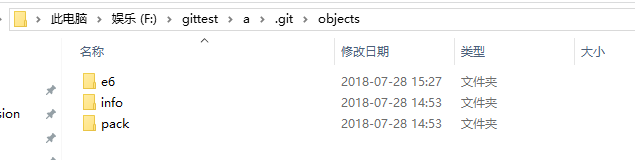
对于下图，修改工作区时，当我们写该file2这样的一个文件，当我们执行git add的时候 我们会发现在对象库中新创建了一个对象，那么暂存区中指向file2的索引就会被指向这个新对象的索引替换掉，这时暂存区就会重新计算一下顶层tree对象的内容，当我们执行git commit的时候，我们直接使用这个已经计算好的内容，创建出一个tree对象，然后生成一个新的commit对象指向这个新生成的顶层目录，并且维护了他的父提交，最后head引用和master引用就指向了这个提交，表明现在在master分支上，而master的最后一个提交是commit e45f9c这个提交。



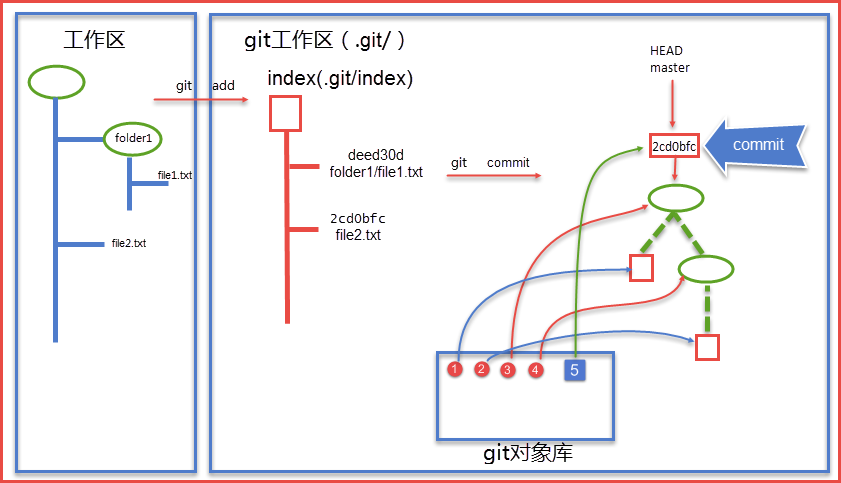
如果一个文件在工作区，但是从未使用add指令添加进暂存区，也就是将该文件从未更新到index索引中，那么Git是会一直显示未跟踪的文件。

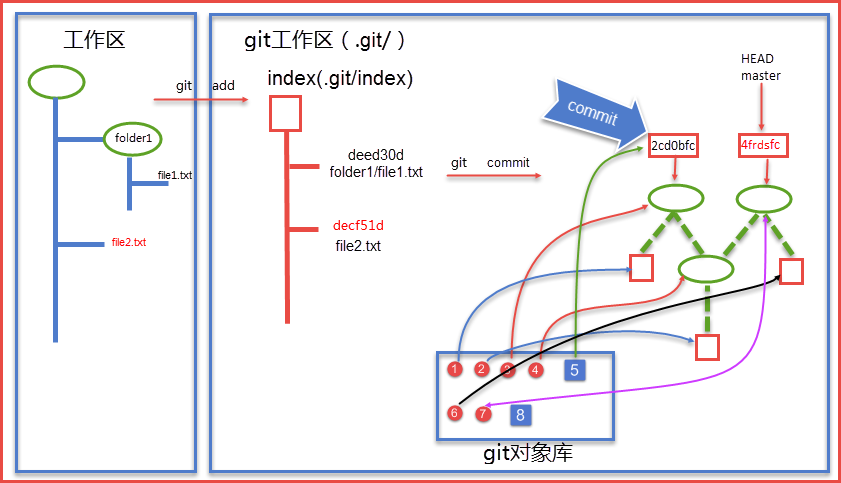


## 版本库



Blob文件集合





## Git add

可以用它开始跟踪新文件

把已跟踪的文件放到暂存区

用于合并时把有冲突的文件标记为已解决状态等

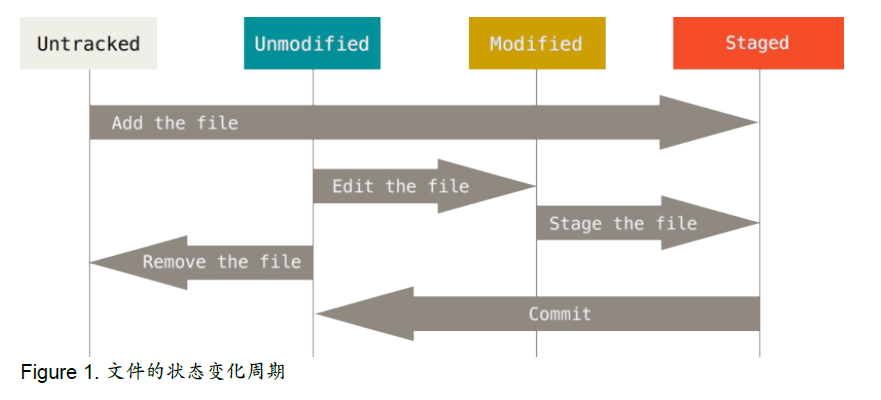
## Git commit



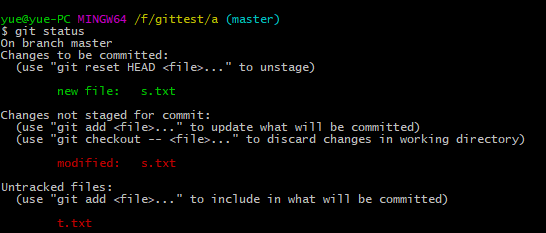
Master (root-commit) ff7881a s中的ff7881a是commit对象的哈希值的前几位。

Create mode 100644, 100代表regular file，644代表文件权限。

## Git status



当前分支的各个文件所在状态

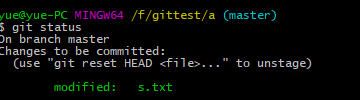


s.txt第一次add加入版本库，只生成了版本1，使用git add s.txt后，生成这个版本的blob对象。在index文件中把s.txt和blob对象关联起来。之后再次修改了s.txt，生成了版本2，但是没有使用git add s.txt,此时这个文件的hash值和blob对象不一致。

执行git status，s.txt会出现两次，Changes to be committed的s.txt是object文件夹中的blob对象，也就是版本1，只要s.txt在历史上没有提交过，changes to be committed后面就是new file。Changes not staged for commit的s.txt是工作区中的s.txt，也就是版本2。

如果此时执行commit,commit指向的tree是object文件夹中的blob对象，也就是版本1。

如果此时执行git add s.txt,object中会生成版本2的blob，index中s.txt对应的hash值是版本2的hash值。之后在使用git status, changes to be committed后面就是modified。



## Git diff(其他文件和工作区的文件相比较)

| **命令** | **作用** |
| --- | --- |
| git diff | 工作区 vs 暂存区 |
| git diff head | 工作区 vs 版本库 |
| git diff --cached | 暂存区 vs 版本库 |

### staging area和working area的文件

git diff

比较index里面和工作区的所有有区别的文件。

git diff a.txt

比较index里面和工作区的a.txt文件。

### master分支和working area的文件

git diff master

比较master指向的commit指向的tree里面的文件和工作区的文件做比较。

Git diff master a.txt

比较指定a.txt

### HEAD指向的内容和working area的文件

Git diff head

Head指向的分支指向的commit指向的tree里面的文件和工作区的文件做比较。

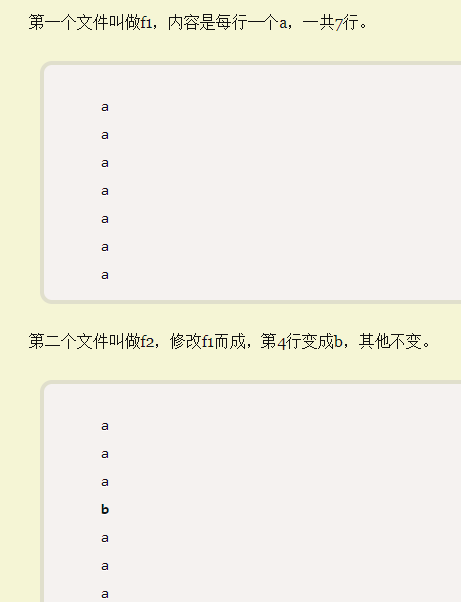
Git diff head^

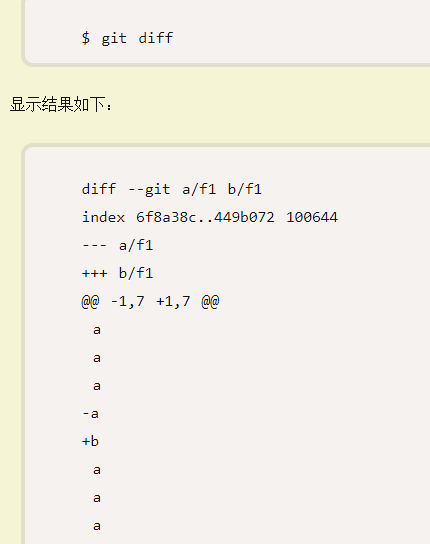
Head指向的commit的父commit的树和工作区的比较。

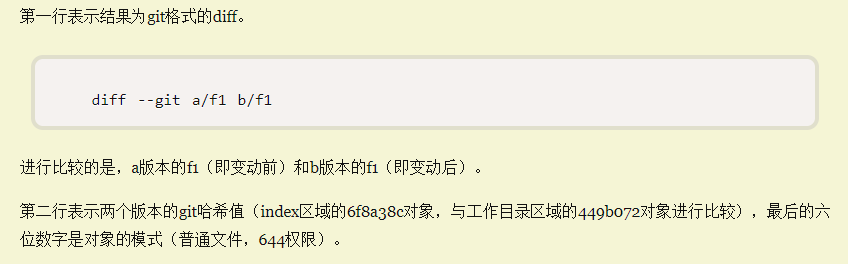
HEAD 表示当前版本，也就是最新的提交。上一个版本就是 HEAD^ ，上上一个版本就是 HEAD^^ ，往上100个版本写100个 “ ^ ” 比较容易数不过来，所以写成 HEAD~100 。HEAD~2 相当于 HEAD^^ 。

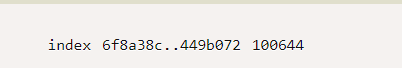


### Git diff的解释

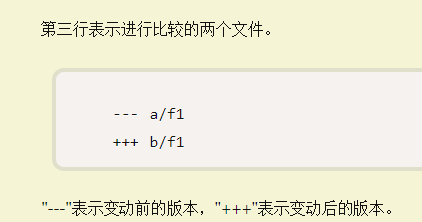






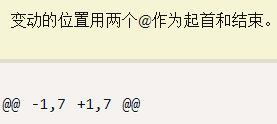


如果使用git diff,比较的是暂存区和工作区的文件，使用git diff head比较的是版本库和工作区的文件。

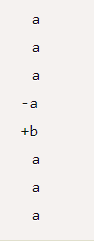


变动前的版本是暂存区或者版本库中的文件,即源文件。

变动后的文件是工作区的文件，即目标文件。



前面的"-1,7"分成三个部分：减号表示第一个文件，也就是源文件（即f1），"1"表示这个文件的第1行，"7"表示连续7行。合在一起，就表示下面展示的是第一个文件从第1行开始的连续7行。同样的，"+1,7"表示下面展示的是变动后的工作区的目标文件，从第1行开始的连续7行。



通常working area的文件都是被当作目标文件来看待。

-开头的行，是只出现在源文件中的行

+开头的行，是只出现在目标文件中的行

空格开头的行，是源文件和目标文件中都出现的行

上面展示的就是文件的两个版本有区别的地方，第一个版本的1到7行和第二个版本的1到7行。

每一行最前面的标志位，空表示无变动，减号表示第一个文件删除的行，加号表示第二个文件新增的行。所以空格和减号一起组合就是暂存区或者版本仓库的源文件，空格和加号一起组合就是工作区的目标文件。



# Blob对象

一个git对象对应一个哈希值，哈希值和这个git对象的存储位置相关。可以使用底层命令hash-object生成一个git对象。hash-object的参数可以是字符串或者一个文件。相同的文件内容（哪怕在不同的路径和文件名）就会有相同的git对象。相同的文件名，不同的文件内容，也会生成不同的git对象。

在底层命令中，使用git hash-object -w m.txt可以生成一个针对m.txt的blob对象。

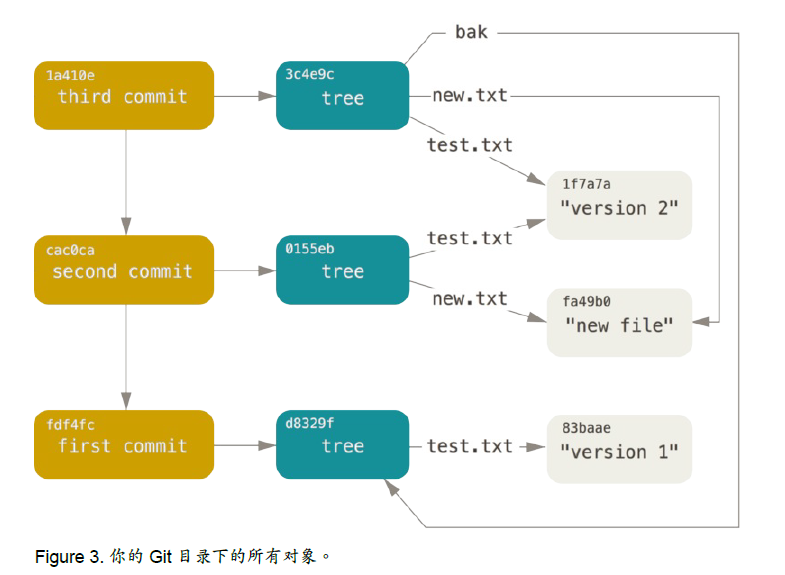
在高层命令中，每次把文件设置成stage状态（执行add操作），就会生成一个blob对象。同样一份文件，每修改一次，执行一次add操作，就会生成一个blob对象。

但是blob对象和文件的关系是通过文件内容的哈希联系起来的，和文件名以及路径没有关系。同一份文件存储在不同的位置，会只生成一个blob对象，无法做到依靠文件名找到对应的blob文件。

# 暂存区

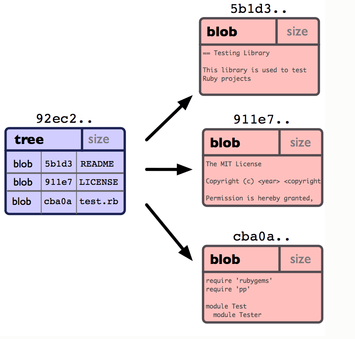
所谓的暂存区只是一个简单的索引文件而已。暂存区这个索引文件里面包含的是文件的目录树，像一个虚拟的工作区，在这个虚拟工作区的目录树中，记录了文件名、文件的时间戳、文件长度、文件类型以及最重要的SHA-1值，文件的内容并没有存储在其中，所以说它像一个虚拟的工作区。

一个暂存区就是一个树对象。提交对象的格式很简单：它先指定一个顶层树对象，代表当前项目快照；然后是作者/提交者信息（依据你的 user.name 和 user.email 配置来设定，外加一个时间戳）；留空一行，最后是提交注释。

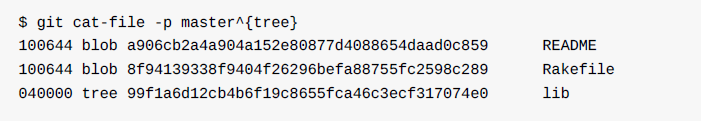


# Tree对象

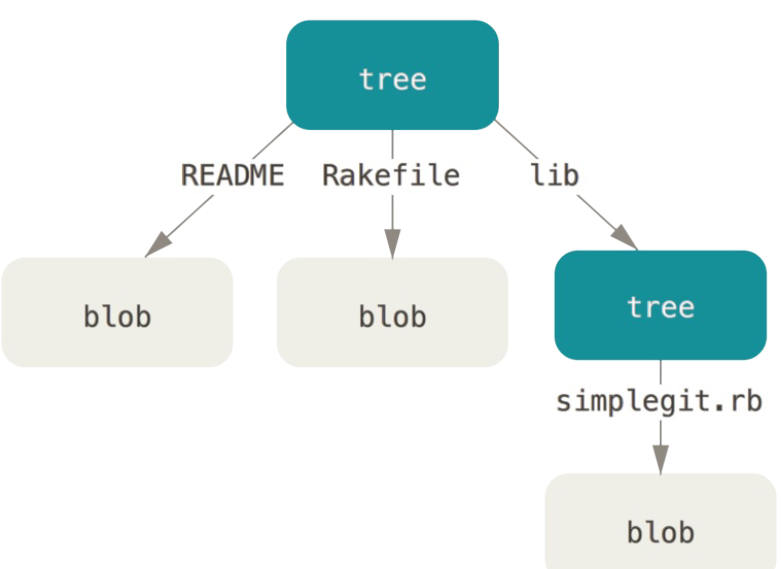
因此引入了Tree对象，一个树对象包含了一条或多条树对象记录（tree entry），一个树对象的每条记录（tree entry）含有一个指向git blob对象或者git tree子树对象的 SHA-1 指针，以及相应的模式、类型、和**文件名信息**。



Tree entry把blob对象和其对应的文件名连接了起来。



master^{tree} 语法表示 master 分支上最新的提交（commit对象）所指向的树对象。所以这个指令显示的是一个tree里面的entry。



上图的Lib是一个tree对象。

在高层命令中，如果暂存区中存在目录m，m目录中存在t.txt。

如果暂存区中存在一个文件夹，文件夹里面有多个文件，提交的时候，文件夹会产生一个子树对象。

每次commit都会生成一个父tree对象对应这次提交，同时生成一个子tree对象对应文件夹m，父tree对象的一个tree entry指向子tree对象（子tree对象对应文件夹m），子tree对象中的一个tree entry指向一个blob对象，指向a.txt。

高层指令上，每次提交生成一个commit对象，指向一个tree，每个tree的entry指向所有被add进缓存区的blob和tree。不管是提交过的还是新加入的，只有这个文件在历史上被add了进缓存区，每次commit的tree都会指向这个文件的最新版本。

比如第一次add了文件a.txt，然后提交，这个时候生成一个commit1指向tree1，tree1指向a.txt的blob对象。之后生成了文件b.txt，然后add了文件b.txt，之后提交，生成了一个commit2，指向了tree2，tree2指向a.txt和b.txt的blob对象。

# Commit对象

Commit对象指向一个tree对象，以及上一个commit对象

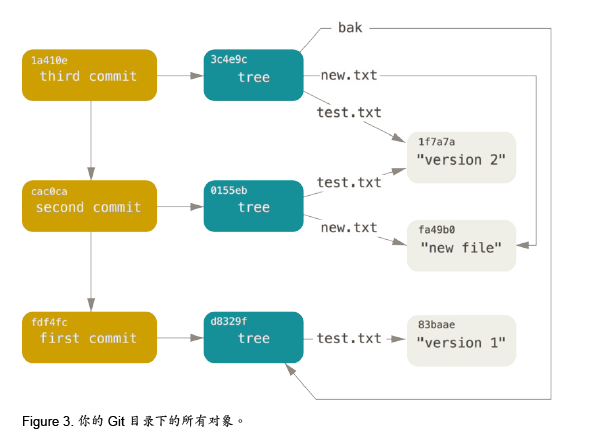
tree 8cc8991dc245cd3407a5d1e287e4c1fe67807743

parent bd1e8b5336b87ed1f9ae1ebb1614a08f4ae9420f

author yue <ayanamishan@sina.com> 1532494444 +0800

committer yue <ayanamishan@sina.com> 1532494444 +0800

总体结构：



# 对象的内容与hash值计算方法

Git 以对象类型作为开头来构造一个头部信息，本例中是一个“blob”字符串。 接着 Git 会添加一个空格，随后是数据内容的长度，最后是一个空字节（null byte），构成了一个头部信息。Git 会将上述头部信息和原始数据拼接起来，并计算出这条新内容的 SHA-1 校验和。之后，SHA-1 值的前两个字符作为子目录名称，后 38 个字符则作为子目录内文件的名称

# 分支与引用

F:\gittest\c\.git\refs\heads下面有一个master的文件，里面有一个commit的hash值，master代表master分支，文件里面的hash值就是这个分支当前的提交。



若想在第二个提交上创建一个名为test的分支，可以这么做：

$ git update-ref refs/heads/test cac0ca

这个分支将只包含从第二个提交开始往前追溯的记录：

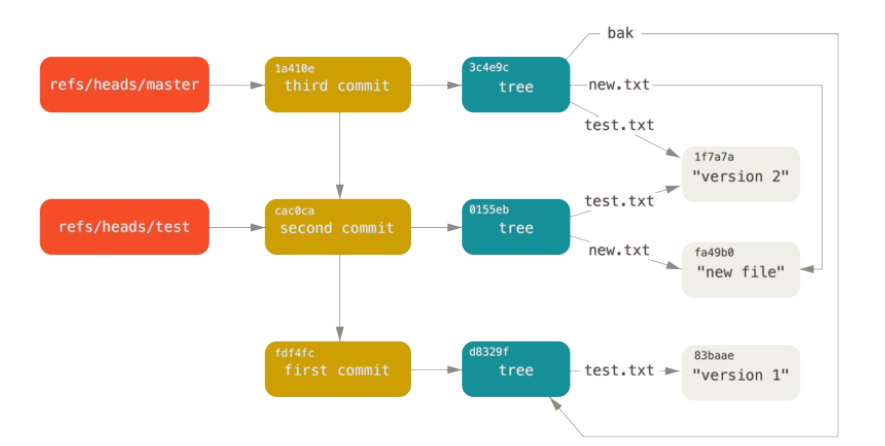
Git 引用：

$ git log --pretty=oneline test

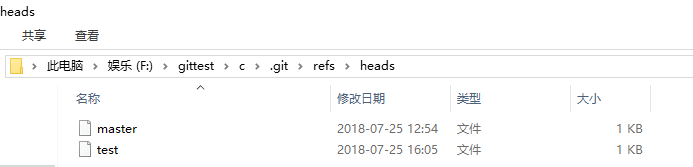
cac0cab538b970a37ea1e769cbbde608743bc96d second commit

fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit

至此，我们的 Git 数据库从概念上看起来像这样：



当运行类似于 git branch test这样的命令时，在F:\gittest\c\.git\refs\heads下面出现一个新的文件：

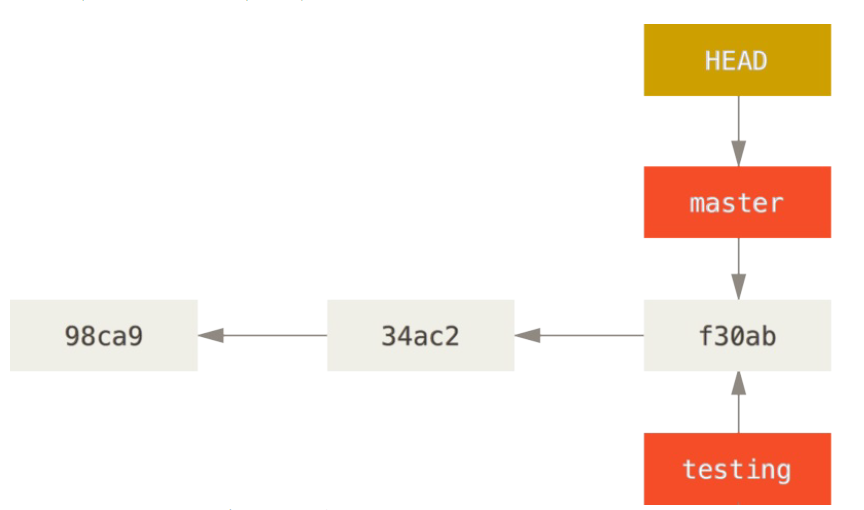


Test中存储了当前所在分支最新提交（head指针指向的引用所指向的commit）对应的 SHA-1 值。

# Head引用

## 分支

当你执行 git branch branchname时，会在head指向的分支的commit上面，在生成一个分支。比如git branch testing，就会生成一个引用，指向master指向的commit对象。



HEAD 文件是一个符号引用（symbolic reference），指向目前所在的分支。 所谓符号引用，意味着它并不像普通引用那样包含一个 SHA-1 值——它是一个指向其他引用的指针。 如果查看 HEAD 文件的内容，一般而言我们看到的类似这样：

$ cat .git/HEAD

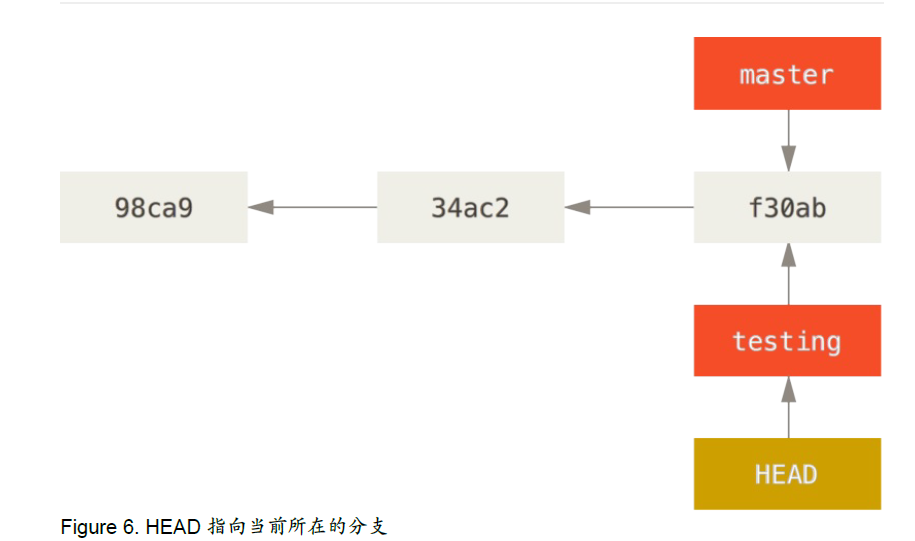
ref: refs/heads/master

也就是说，head指针指向一个分支的ref。

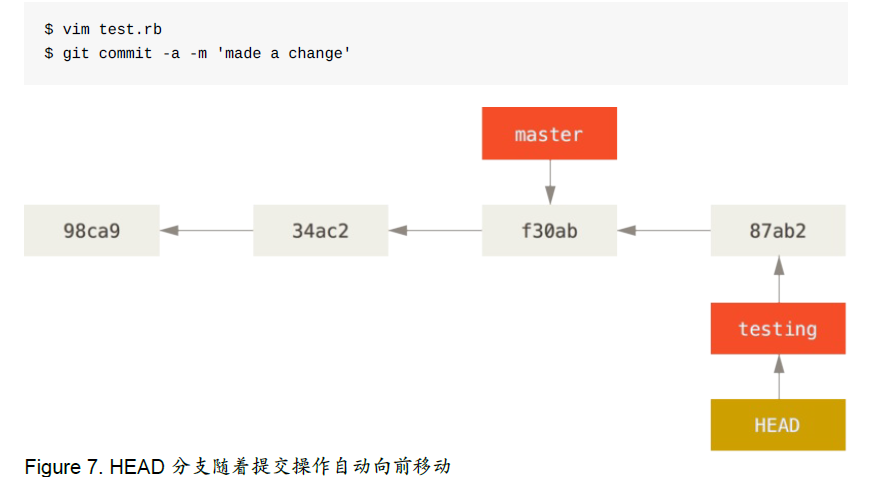
如果执行 git checkout test ，Git 会像这样更新 HEAD 文件：

$ cat .git/HEAD

ref: refs/heads/test

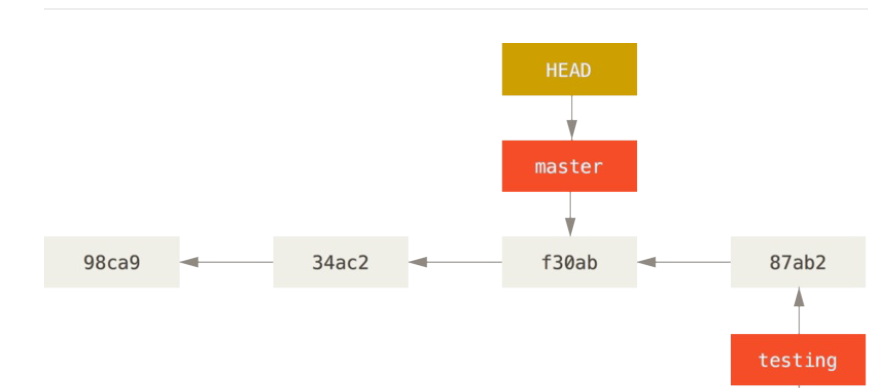


当我们执行 git commit 时，该命令会创建一个提交对象，并用 HEAD 文件中那个引用所指向的 SHA-1 值（commit对象的值）设置其父提交字段。

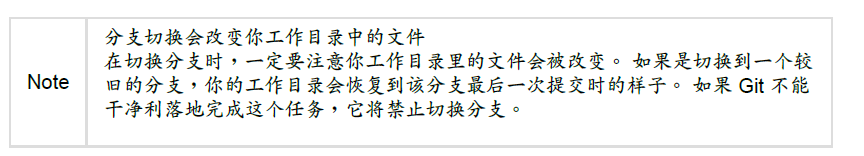


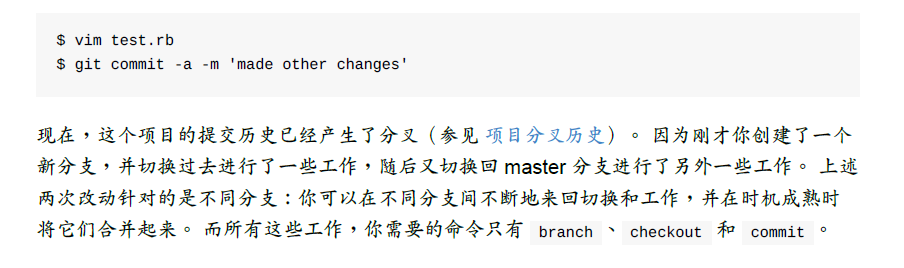
testing 分支向前移动了，但是 master 分支却没有，它仍然指向运行 git checkout 时所指的对象。 这就有意思了，现在我们切换回 master 分支看看：

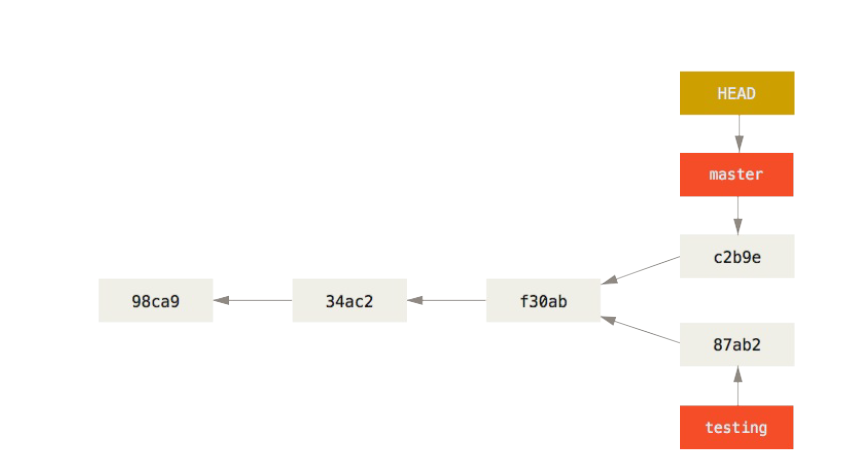
$ git checkout master



这条命令做了两件事。 一是使 HEAD 指回 master 分支，二是将工作目录恢复成 master 分支所指向的快照内容。







每次commit的时候，生成一个commit对象，head指向的分支指向的原commit就是父commit。

## Git checkout

### 分支切换

使用git checkout testing的时候，把head指向对应的分支testing，工作区和暂存区都被testing指向的commit指向的tree同步。

### 覆盖工作区

git checkout -- filename

用暂存区中的filename文件来覆盖工作区中的filename文件.相当于取消自上次执行git add filename以来(如果执行过)的本地修改

git checkout -- .或者git checkout . （别忘了点）

会取消本地所有存在于暂存区的文件的修改,相当于用暂存区的所有文件直接覆盖本地文件，工作区中暂存区不存在的文件不受影响。

Git checkout branch是工作区的文件全部清空，然后再把branch的commit的tree移到工作区。

git checkout -- .或者git checkout .不会清空工作区，只会把缓存区的文件覆盖工作区的文件。

Git checkout HEAD filename

HEAD的版本库的文件覆盖工作区和暂存区的文件。

Git checkout HEAD~1 filename

Head前一个commit指向的tree的文件覆盖当前工作区和缓存区，head指针和commit不变。

如果使用reset –hard的话，head指针会移动。

Git checkout HEAD .

使用head的commit指向的树覆盖当前的工作区和缓存区。

### Stash

如果修改了工作区或者缓存区，在提交之前无法使用checkout切换分支。但是没有完成的工作又不想提交，就使用git stash把当前工作暂存，工作区和缓存区回到上一个commit，之后使用checkout切换分支。

1.使用git stash 保存当前的工作现场， 那么就可以切换到其他分支进行工作，或者在当前分支上完成其他紧急的工作，比如修订一个bug测试提交。

2.如果一个使用了一个git stash，切换到一个分支，且在该分支上的工作未完成也需要保存它的工作现场。再使用git stash。那么stash 队列中 就有了两个工作现场。

3.可以使用git stash list。查看stash队列。

4.如果在一个分支上想要恢复某一个工作现场怎么办：先用git stash list查看stash队列。确定要 恢复哪个工作现场 到当前分支。然后用git stash pop stash@{num}。num 就是你要恢复的工作现场的编号。

5.如果想要清空stash队列则使用git stash clear。

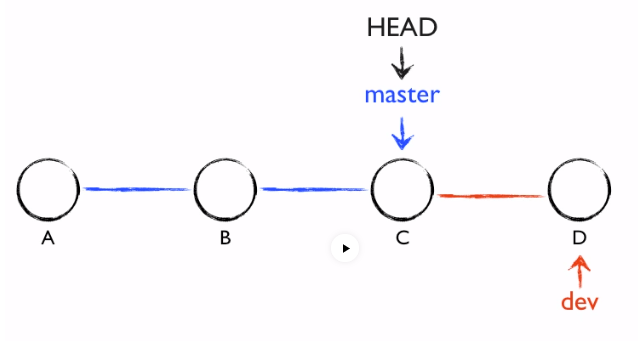
6.同时注意使用git stash pop命令是恢复stash队列中的stash@{0}即最上层的那个工作现场。而且使用pop命令恢复的工作现场，其对应的stash 在队列中删除。

使用git stash apply stash@{num}方法 除了不在stash队列删除外其他和git stash pop 完全一样。

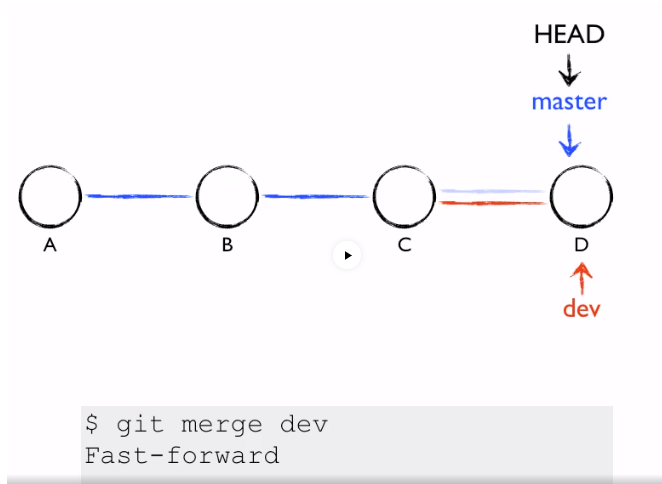
Git stash只在本地存储，不会上传服务器。

## 分支合并

### Fast forward



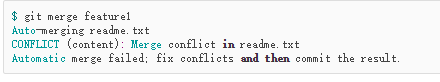
head在master分支的时候，使用git merge dev。



## 冲突



如果master分支和feature1分支进行合并，两个分支中对同一个文件进行修改，合并的时候，就会冲突。



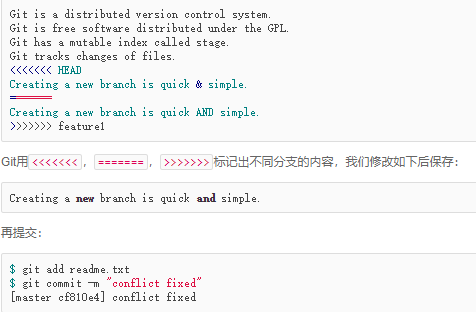
Git merge feature1 –m “automerge”

要加m参数，不然报错。

此时，冲突之后，状态依然和上面一样，没有产生新的commit，暂存区也不会改变。



但是工作区中冲突的文件被改写了：





用带参数的git log也可以看到分支的合并情况：

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

\* cf810e4 (HEAD -> master) conflict fixed

|\

| \* 14096d0 (feature1) **AND** simple

\* | 5dc6824 & simple

|/

\* b17d20e branch test

\* d46f35e (origin/master) remove test.txt

\* b84166e add test.txt

\* 519219b git tracks changes

\* e43a48b understand how stage works

\* 1094adb append GPL

\* e475afc add distributed

\* eaadf4e wrote a readme file

冲突之后，修改的内容只要确认没有问题，就可以用add和commit提交。这次提交的时候，git不会在检测冲突。Head指向的master指向的commit的评价就是git commit的时候的评价。

可以使用git merge –abort取消合并。

如果没有发生冲突，可以自动合并。指向git merge就是生成一个新的commit，评价是git merge的时候的m参数。



## Git log

查看提交历史

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

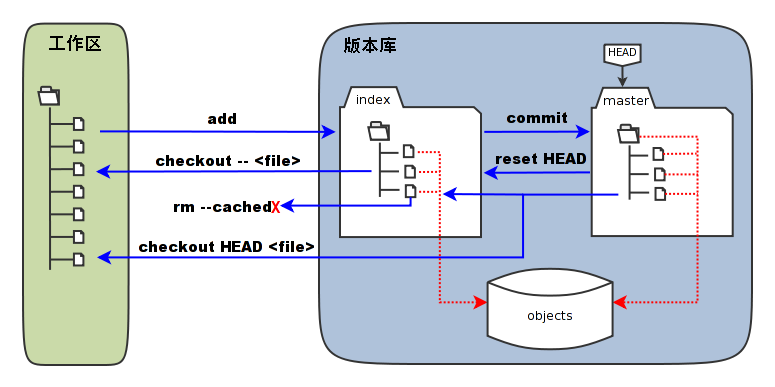
Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

使用graph参数可以看到图形

# 版本回退



## Git checkout

切换分支的时候，head移动。

将缓存区文件覆盖工作区的时候，将版本库的文件覆盖缓存区和工作区的时候，不会移动head。

## Git reset

如果你仔细研究reset命令本身就知道，它本身做的事情就是重置HEAD(当前分支的版本顶端）到另外一个commit。假设我们有一个分支（名称本身无所谓，所以我们就简单称为"super-duper-feature”分支吧），图形化表示如下：



如果我们执行：

git reset HEAD

任何事情都不会发生，这是因为我们告诉GIT重置这个分支到HEAD，而这个正是它现在所在的位置。

git reset HEAD~1

当我们再执行上面的命令时（HEAD~1是“the commit right before HEAD”的别名，或者说：put differently "HEAD‘s parent"），我们的分支将会如下所示



如果我们执行git reset HEAD~2,则意味着将HEAD从顶端的commit往下移动两个更早的commit。

* Parameters

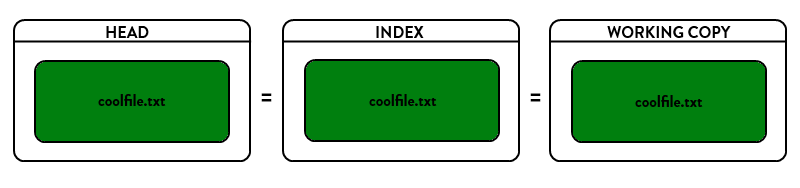
1. soft

--soft参数告诉Git重置HEAD到另外一个commit，但也到此为止。如果你指定--soft参数，Git将停止在那里而什么也不会根本变化。这意味着index,working copy都不会做任何变化，所有的在original HEAD和你重置到的那个commit之间的所有变更集仍然在stage(index)区域中。



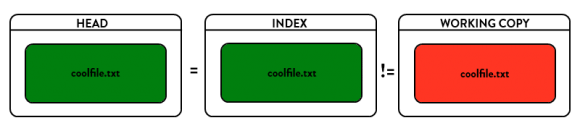
2.hard

--hard参数将会blow out everything.它将重置HEAD返回到另外一个commit(取决于~12的参数），重置index以便反映HEAD的变化，并且重置working copy也使得其完全匹配起来。这是一个比较危险的动作，具有破坏性，数据因此可能会丢失！如果真是发生了数据丢失又希望找回来，那么只有使用：[git reflog](http://blog.csdn.net/ibingow/article/details/7541402)命令了



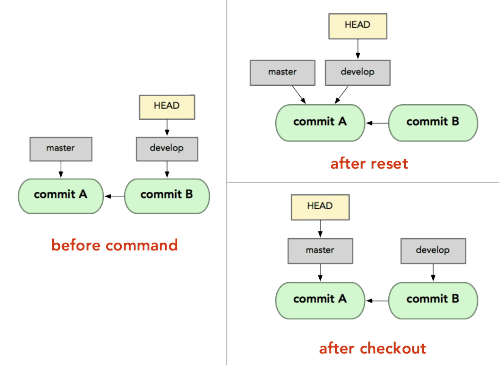
3.mixed(default）

--mixed是reset的默认参数，也就是当你不指定任何参数时的参数。它将重置HEAD到另外一个commit,并且重置index以便和HEAD比配，但是也到此为止。working copy不会被更改。所以所有从original HEAD到你重置到的那个commit之间的所有变更仍然保存在working copy中，被标示为已变更，但是并未staged的状态



Mixed可以指定一个文件的路径，hard和soft不能指定文件的路径。

## Reset和checkout在移动head的区别



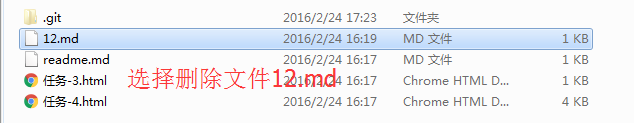
对于修改暂存区和缓冲区的文件

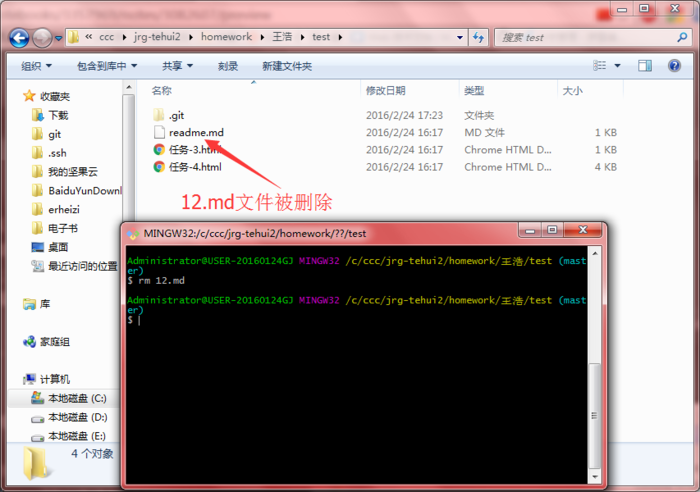
Checkout只会把版本库或者暂存区的文件取出来，写回。

Reset会把版本库或者暂存区的东西先清除，在把版本库或者暂存区的文件取出来，写回。

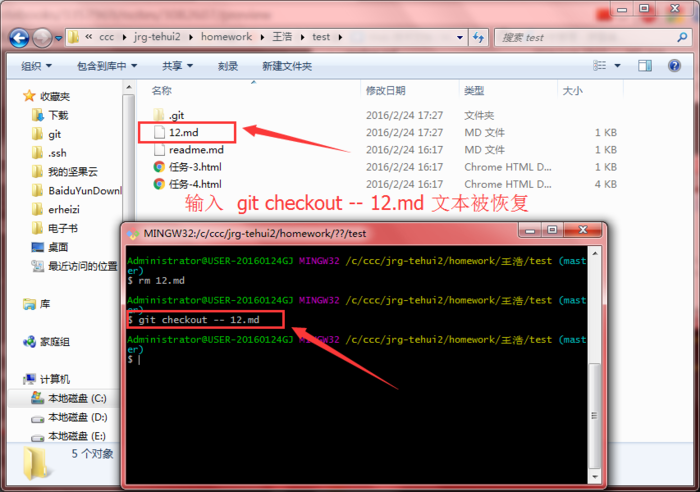
# 删除

## 删除了本地文件，但未添加到暂存区

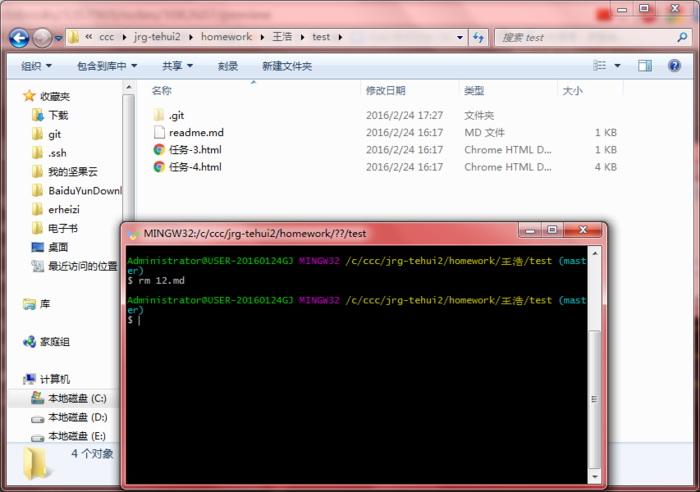




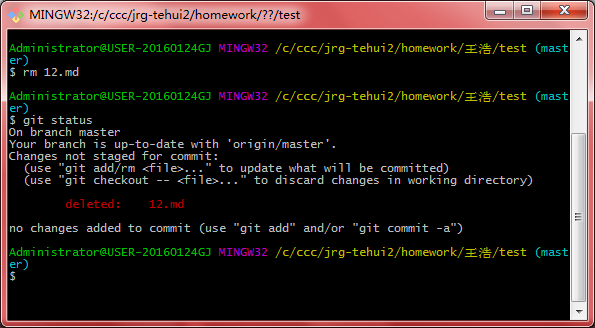
此时发现删除错误了要进行恢复操作



## 删除了本地文件，并且已经把删除操作添加到了暂存区



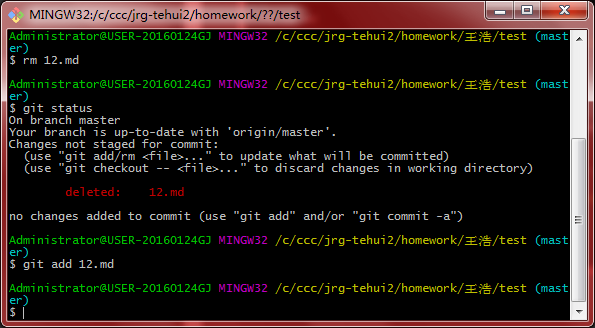
通过git status查看状态



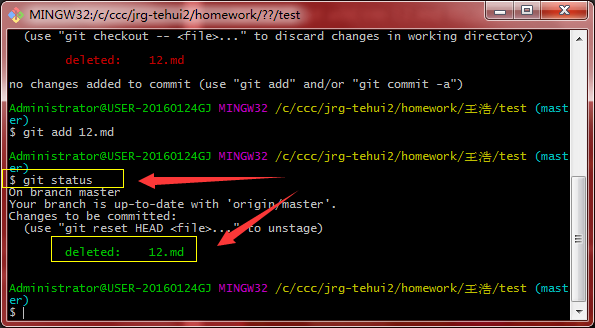
看提示可以知道，此时如果操作 git checkout 12.md 可以恢复删除文件

如果操作 git add/rm 12.md 可以把删除状态添加到暂存区

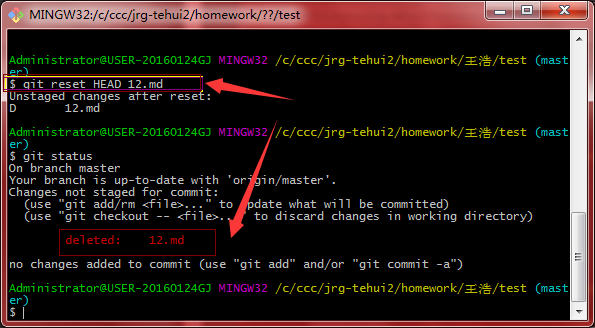
我们进行添加操作

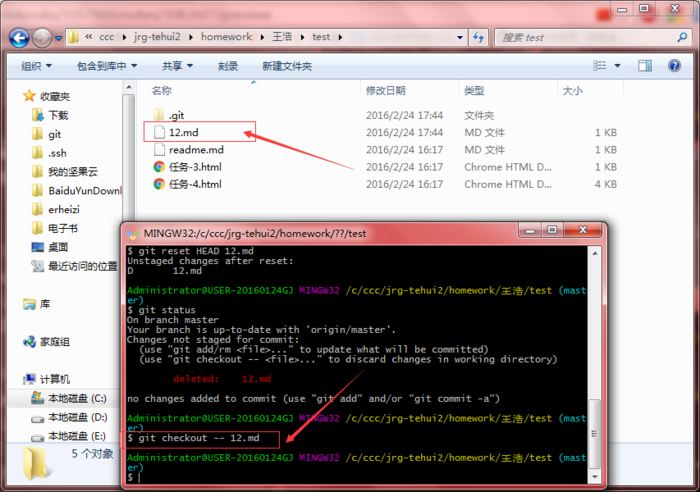


查看状态，可以知道删除操作已经被添加到了暂存区，



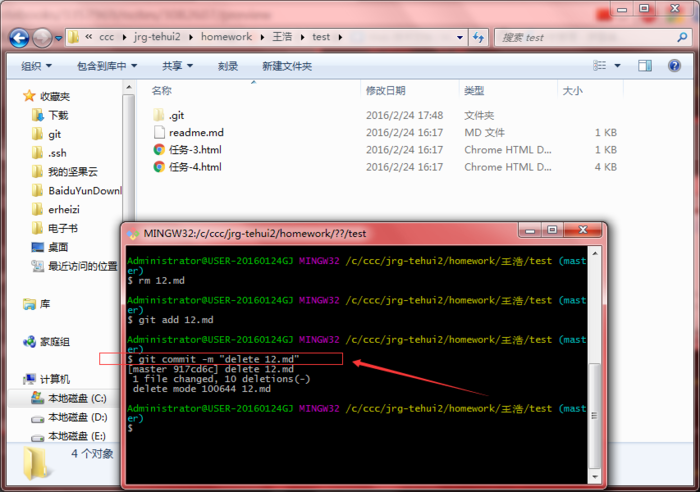
此时如果要撤销操作，从提示可以知道，进行 git reset HEAD 12.md可以把删除操作退回到本地删除状态，然后按照上面1的操作就可以恢复文件





或者使用git checkout head .也可以。

## 从暂存区把删除操作提交到了本地git库



查看状态 git status

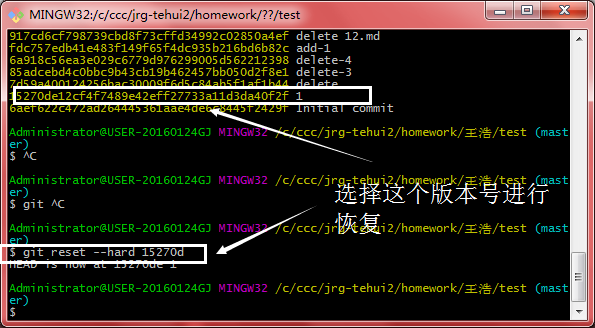
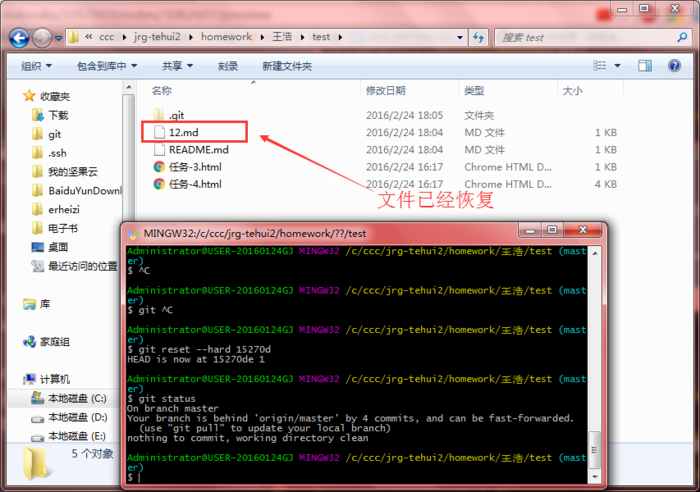
可以看到有一个提交改动，此时如果要撤销，就必须对git进行版本回滚操作

通过 git log 命令查看git库的所有版本信息



看到一大串类似 6aef622c4……8445f2429f的用16进制表示的字符串是每次 commit 的ID版本号，此时我们进行版本回滚操作

选择ID的前几位字符串就足够表示版本的唯一性，输入命令 git reset --hard 6aef62

# 远程库（github）

## 设定ssh

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：

$ ssh-keygen -t rsa -C "youremail@example.com"

你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面：

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容：



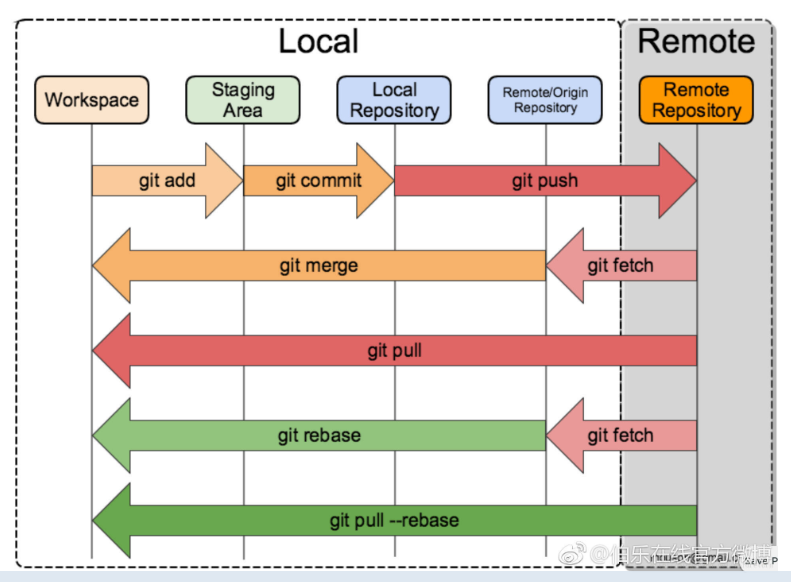
点“Add Key”，你就应该看到已经添加的Key：



为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

## 远端库数据流程



## 本地git推送到github

找到“Create a new repo”按钮，创建一个新的仓库：



这里用的名字是gituploadtest

目前，在GitHub上的这个gituploadtest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到本地，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，在本地的仓库下运行命令：

git remote add origin [git@github.com:ayanamiyue/gituploadtest.git](mailto:git@github.com:ayanamiyue/gituploadtest.git)

添加后，远程库的网址是[git@github.com:ayanamiyue/gituploadtest.git](mailto:git@github.com:ayanamiyue/gituploadtest.git)，远程库的名字是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。总之，就是把origin和[git@github.com:ayanamiyue/gituploadtest.git](mailto:git@github.com:ayanamiyue/gituploadtest.git)对应起来。

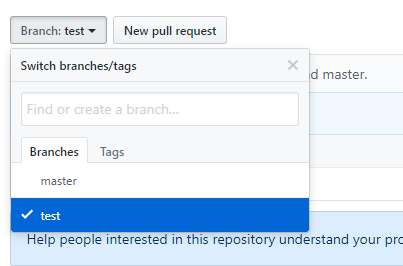
下一步，就可以把本地库的所有内容推送到远程库上：

git push -u origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果master不存在，则会被新建。

git push -u origin test

把test分支推送到远端。



git push --all origin

上面命令表示，将所有本地分支都推送到origin主机。

## 从git克隆一个库到本地

如果是其他人的项目，使用fork到自己的项目。

使用clone命令把自己的项目下载到本地

git clone [git@github.com:ayanamiyue/multibranches.git](mailto:git@github.com:ayanamiyue/multibranches.git)

开发者键入git clone后，git会将远端所有的代码和branchs下载到开发者本地。

有人会问，可是我git branch只看到一个branch（默认情况应该是master branch。不过仓库服务端可以进行配置其他任意branch为默认主分支）。git branch的确只能看到一个branch（在本地没有进行checkout -b参操作的情况下），因为其他branch需要用git branch -r（Git branch -a）才能看到。

这时有人又会问，git branch -r看到的不是远端的分支吗？好吧，其实本文的唯一有价值的地方就在这里了。git branch -r看到的不是远端服务的branchs，而是你的本地的远端 branchs，git clone后在remote/origin repository区保存一份和此时远端分支一模一样的代码和branchs，而且该区域是不能修改的。

Git 的 clone 命令将远端服务器的所有blob和所有的branch下载到本地的remote/origin repository区。之后在把origin/master分支的blob文件拷贝到本地版本库，然后把本地的分支设为master。

如果想在本地把remote/origin repository区的其他分支拷贝到本地版本库，可以使用git checkout -b master origin/master把其他分支拷贝到版本库并且切换本地分支，相信大家都用过类似的命令，这个是从远端拉出一个branch吗？不是，这是从你的本地的远端区域里的master branch拉出一个branch。而新拉出的branch就是存储在你的local repository区域里，这个区域是可以修改的，也就是git branch看到的那些branchs。

## Git fetch

一旦远程主机的版本库有了更新(Git术语叫做commit)，需要将这些更新取回本地，这时就要用到git fetch命令。

$ git fetch <远程主机名>

上面命令将某个远程主机的更新，全部取回本地。

默认情况下，git fetch取回所有分支(branch)的更新。如果只想取回特定分支的更新，可以指定分支名。

$ git fetch <远程主机名> <分支名>

比如，取回origin主机的master分支。

$ git fetch origin master

所取回的更新，在本地主机上要用”远程主机名/分支名”的形式读取。比如origin主机的master，就要用origin/master读取。

git branch命令的-r选项，可以用来查看远程分支，-a选项查看所有分支。

1. $ git branch -r
2. origin/master
4. $ git branch -a
5. \* master
6. remotes/origin/master

上面命令表示，本地主机的当前分支是master，远程分支是origin/master。

取回远程主机的更新以后，可以在它的基础上，使用git checkout命令创建一个新的分支。

git fetch其实是更新下你的本地的远端代码区域（remote/origin repository）的数据同服务端的代码保持一致。

## Git pull

$ git pull <远程主机名> <远程分支名>:<本地分支名>

Shell

比如，要取回origin主机的next分支，与本地的master分支合并，需要写成下面这样 -

$ git pull origin next:master

Shell

如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略。上面命令可以简写为：

$ git pull origin next

Shell

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

$ git fetch origin

$ git merge origin/next

Shell

在某些场合，Git会自动在本地分支与远程分支之间，建立一种追踪关系(tracking)。比如，在git clone的时候，所有本地分支默认与远程主机的同名分支，建立追踪关系，也就是说，本地的master分支自动”追踪”origin/master分支。

Git也允许手动建立追踪关系。

$ git branch --set-upstream master origin/next

Shell

上面命令指定master分支追踪origin/next分支。

如果当前分支与远程分支存在追踪关系，git pull就可以省略远程分支名。

$ git pull origin

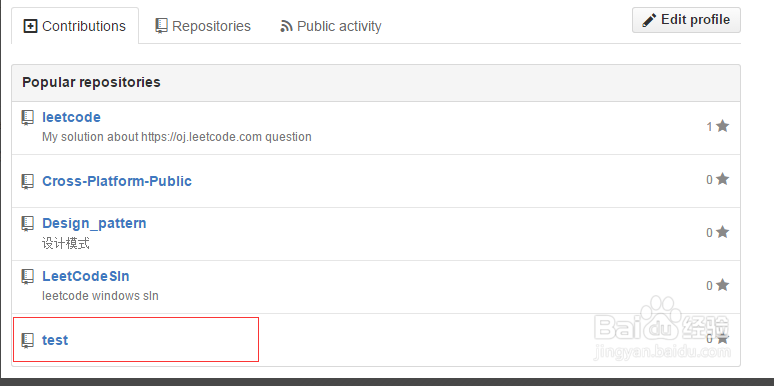
Shell

上面命令表示，本地的当前分支自动与对应的origin主机”追踪分支”(remote-tracking branch)进行合并。

如果当前分支只有一个追踪分支，连远程主机名都可以省略。

## 删除仓库

选择要被删除的仓库



选择setting

