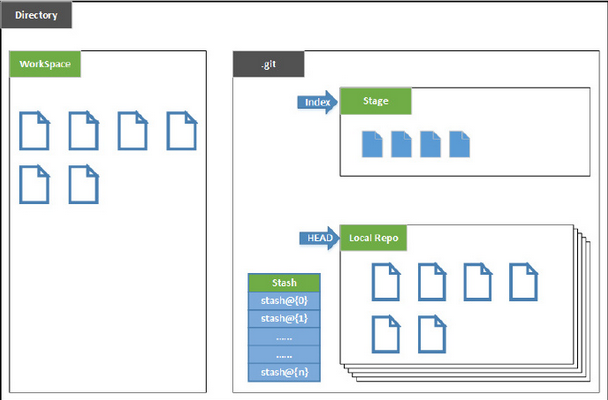
Git本地Repo

# Git基本概念

在开始Git的使用之前，需要先介绍一些概念，通过这些概念对Git有些基本的认识，这将会对我们后面的操作有很大的帮助。

在Git中，每个版本库都叫做一个仓库（repository），每个仓库可以简单理解成一个目录，这个目录里面的所有文件都通过Git来实现版本管理，Git都能跟踪并记录在该目录中发生的所有更新。

现在我们已经知道什么是repository（缩写repo）了，假如我们现在建立一个仓库（repo），那么在建立仓库的这个目录中会有一个".git"的文件夹。这个文件夹非常重要，所有的版本信息、更新记录，以及Git进行仓库管理的相关信息全都保存在这个文件夹里面。所以，不要修改/删除其中的文件，以免造成数据的丢失。进一步的讲解请参考下面一张图，大概展示出了我们需要了解的基本知识（注意，".git"目录中还有很多别的东西，图中并没有涉及，这里也不做解释了）。

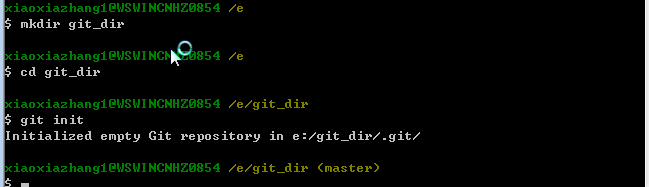


根据上面的图片，下面给出了每个部分的简要说明：

* Directory：使用Git管理的一个目录，也就是一个仓库；包含我们的工作空间和Git的管理空间。
* WorkSpace：从仓库中checkout出来的，需要通过Git进行版本控制的目录和文件；这些目录和文件组成了工作空间。
* .git：存放Git管理信息的目录，初始化仓库的时候自动创建。
* Index/Stage：暂存区，或者叫做待提交更新区；在提交进入repo之前，我们可以把所有的更新放在暂存区。
* Local Repo：本地仓库，一个存放在本地的版本库；HEAD会指示当前的开发分支（branch）。
* Stash：是一个工作状态保存栈，用于保存/恢复WorkSpace中的临时状态。

# 创建仓库

通过"Git Bash"命令行窗口进入到想要建立版本仓库的目录，通过"git init"就可以轻松的建立一个仓库。这时，我们的仓库目录中会自动的产生个".git"文件夹，这个就是我们前面提到的Git管理信息的目录。

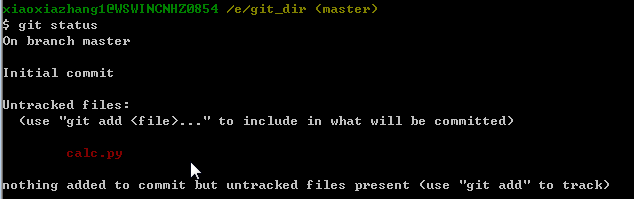


# 添加

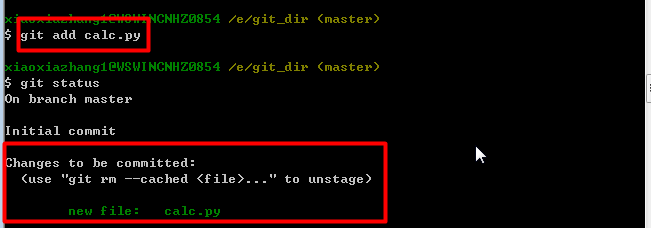
现在我们在仓库中新建一个"calc.py"的文件，文件内容如下。

|  |
| --- |
| def add(a, b):      print a + b    if \_\_name\_\_ == "\_\_main\_\_":      add(2, 3) |

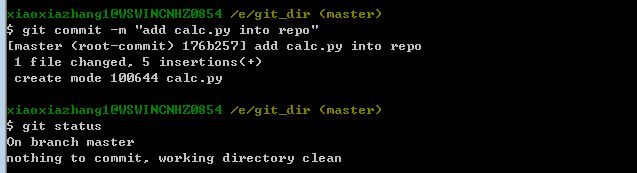
通过"git status"可以查看WorkSpace的状态，看到输出显示"calc.py"没有被Git跟踪，并且提示我们可以使用"git add <file>…"把该文件添加到待提交区（暂存区）。注意：**这时的更新只是在WorkSpace中**。



使用"git add calc.py"或者"git add ."，然后继续查看WorkSpace的状态。这是发现文件已经被放到暂存区。注意：这时的更新已经从WorkSpace保存到了Stage中。



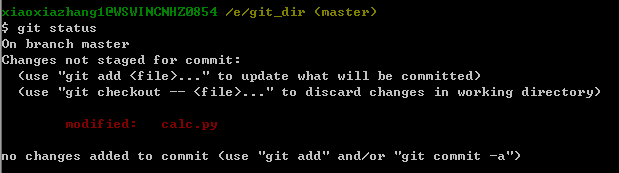
最后，我们就可以通过"git commit -m"来提交更新了。-m后面跟的是对commit的描述（message）。注意：**这时的更新已经又从Stage保存到了Local Repo中**。



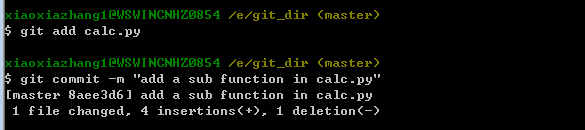
# 更新

假设现在需要对"calc.py"进行更新，修改文件后，查看WorkSpace的状态，会发现提示文件有更新，但是更新只是在WorkSpace中，没有存到暂存区中。

|  |
| --- |
| def add(a, b):      print a + b    def sub(a, b):      print  a - b    if \_\_name\_\_ == "\_\_main\_\_":  add(2, 3) |

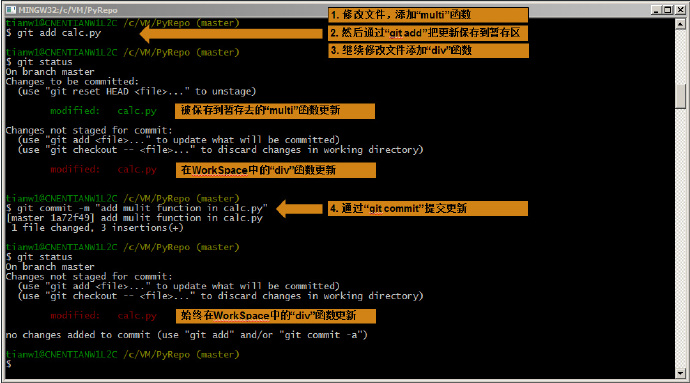


同样，通过add、commit的操作，我们可以把文件的更新先存放到暂存区，然后从暂存区提交到repo中。



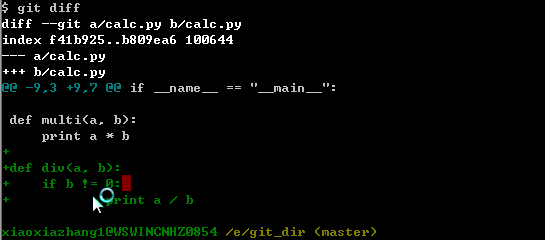
注意：**只有被add到暂存区的更新才会被提交进入repo。**比如下面的一系列操作，操作结束后只有"multi"函数的更新会被提交到repo中，"div"函数的更新还在WorkSpace中。这点应该也是比较容易理解的。

|  |
| --- |
| def multi(a, b):      print a \* b    def div(a, b):      if b != 0:          print a / b |



# git diff

"git diff"是一个很有用，而且会经常用到的命令。基于上面的例子，我们通过"git diff"来查看WorkSpace和Stage的diff情况，当我们把更新add到Stage中，diff就不会有任何输出了。



当然，我们也可以把WorkSpace中的状态跟repo中的状态进行diff，命令如下，关于HEAD，将在后面解释。git diff HEAD~n

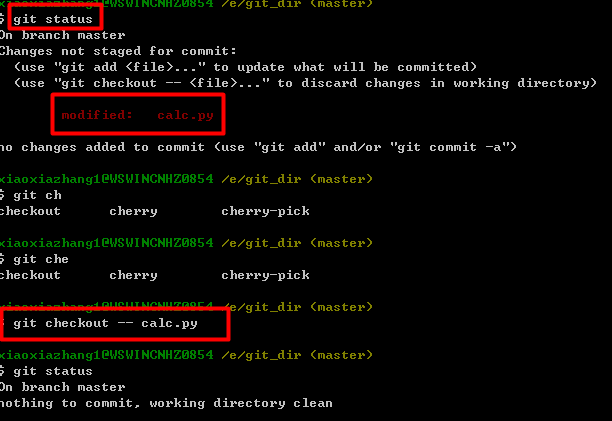
# 撤销更新

根据前面对基本概念的了解，更新可能存在三个地方，WorkSpace中、Stage中和repo中。下面就分别介绍一下怎么撤销这些更新。

## 撤销WorkSpace中的更新

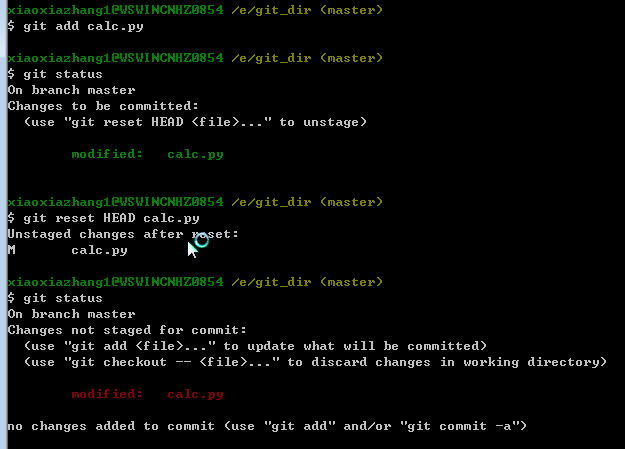
接着上面的例子，我们想撤销WorkSpace中的"div"函数的更新，可以看到"git status"的输出中有提示，我们可以使用"git checkout -- <file>…"（注意一定不要漏掉--）来撤销WorkSpace中的更新。

注意：在使用这种方法撤销更新的时候一定要慎重，因为通过这种方式撤销后，更新将没有办法再被找回。



## 撤销Stage中的更新

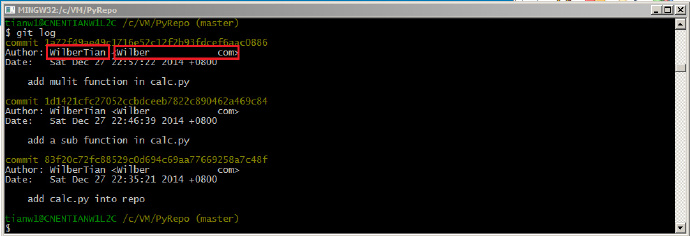
加入我们在WorkSpace中重新添加了"div"函数的更新，并且使用了"git add"把这个更新提交到了暂存区。这时，"git status"的输出中提示我们可以通过"git reset HEAD <file>…"把暂存区的更新移出到WorkSpace中。



如果想继续撤销WorkSpace中的更新，请参考上面一步。

## 撤销repo中的更新

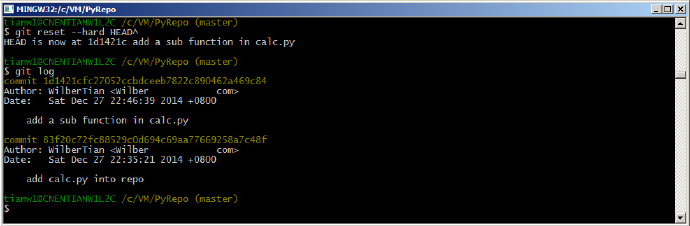
介绍撤销repo中的更新之前，我们先看一下"git log"这个命令，通过这个命令我们可以查看commit的历史记录。可以看到我们进行的三次提交



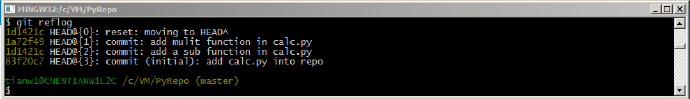
假设我们现在要撤销"add mulit function in calc.py"这个提交，有两种方式：**使用HEAD**指针和使用commit id。

在Git中，有一个HEAD指针指向当前分支中最新的提交，在上面的例子中HEAD就是对应1a72f49ae49c1716e52c12f2b93fdcef6aac0886（commit id）这次提交。所以可以使用下面的命令来撤销"add mulit function in calc.py"这个提交。注意，当前版本，我们使用"HEAD^"，那么再前一个版本可以使用"HEAD^^"，如果想回退到更早的提交，可以使用"HEAD~n"。（也就是，HEAD^=HEAD~1，HEAD^^=HEAD~2）

|  |
| --- |
| git reset --hard HEAD^  git reset --hard 1a72f49ae49c1716e52c12f2b93fdcef6aac0886 |

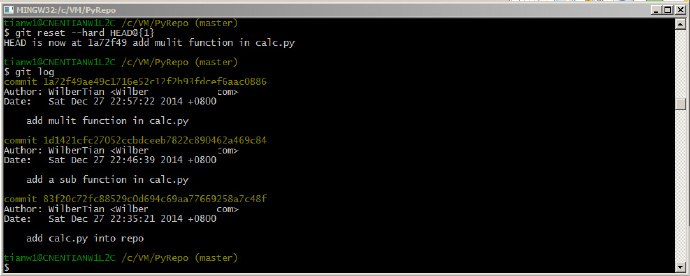


那么问题就来了，我现在又想要恢复"add mulit function in calc.py"这个提交了，当然Git是支持这样的操作。下面来看看"git reflog"这个命令。"git log"只是包括了当前分支中的commit记录，而"git reflog"中会记录这个仓库中所有分支的所有更新记录，包括已经撤销的更新。



有了这个，我们就可以查找到"add mulit function in calc.py"提交，然后可以通过下面命令来恢复对"add mulit function in calc.py"的撤销操作。

|  |
| --- |
| git reset --hard HEAD@{1}  git reset --hard 1a72f49 |



**--hard和--soft的区别**

前面在使用reset来撤销更新的时候，我们都是使用的"--hard"选项，其实与之对应的还有一个"--soft"选项，区别如下：

* --hard：撤销并删除相应的更新
* --soft：撤销相应的更新，把这些更新的内容放的Stage中

# 删除文件

在Git中，如果我们要删除一个文件，可以使用下面的命令,"git rm"相比"rm"只是多了一步，把这次删除的更新发到Stage中。

rm <file>

git rm <file>

# 总结

