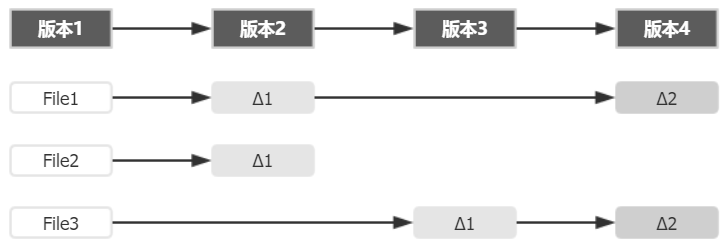
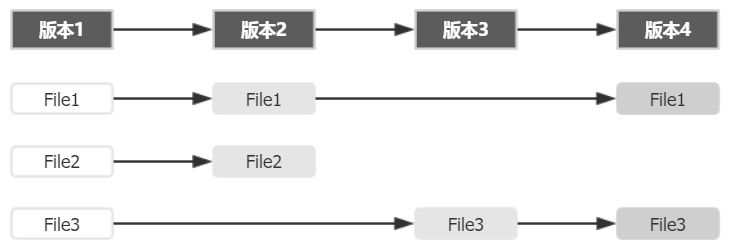
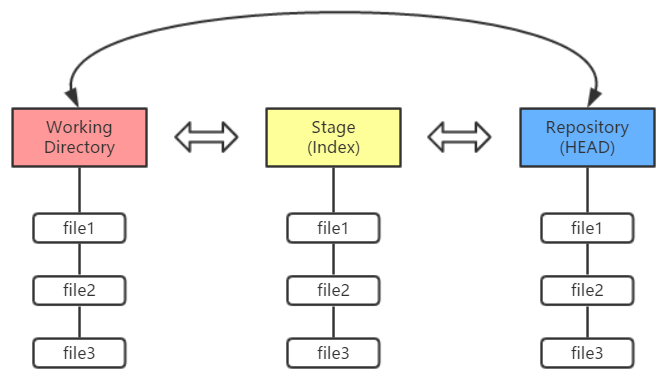
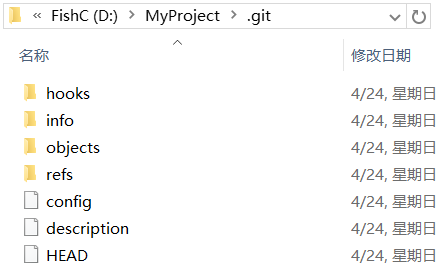
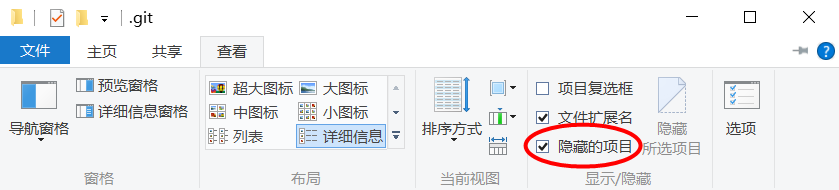
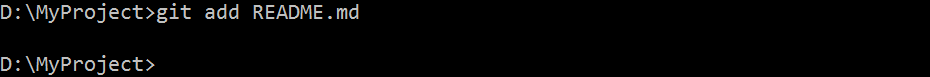
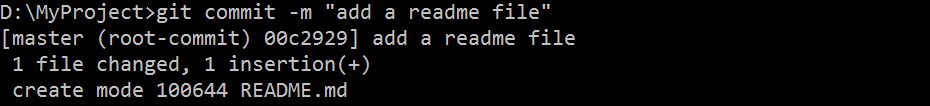
****Git 记录的是什么？****

如果你有使用 SVN 等其他版本控制系统的经验，你应该知道它们的工作原理是记录每一次的变动。  
  
   
  
差不多就是上面酱紫，每一次版本迭代，SVN 记录的是文件的变化内容。通常让我们自己来写一个版本管理工具也会首选这样的思维吧？就像写小说一样，每次就增加一个章节，修改若干错别字，最终装订成册……没必要为每次的修改都拷贝一整本书！这种存储方式也是有名堂的，叫增量文件系统（Delta Storage systems）。  
  
而 Linus 童鞋这次却决定剑走偏锋，以一种看似“异端”的方式来处理版本迭代：  
  
   
  
如上，如果每个版本中有文件发生变动，Git 会将整个文件复制并保存起来。这种设计看似会多消耗更多的空间，但在分支管理时却是带来了很多的益处和便利（分支管理我们后边会讲，不急）。  
  
****突然想到了一句话：普通的程序员是把很多的时间放在写代码和调 Bug 上，而优秀的程序员是将更多的精力放在设计上……****  
  
IMG_258   
  
****警告：前方高能，请先洗脸！！！****  
  
****三棵树****

     
  
你的本地仓库有 Git 维护的三棵“树”组成，这是 Git 的核心框架。  
  
这三棵树分别是：工作区域、暂存区域和 Git 仓库  
  
   
  
工作区域（Working Directory）就是你平时存放项目代码的地方。  
  
暂存区域（Stage）用于临时存放你的改动，事实上它只是一个文件，保存即将提交的文件列表信息。  
  
Git 仓库（Repository）就是安全存放数据的位置，这里边有你提交的所有版本的数据。其中，HEAD 指向最新放入仓库的版本（这第三棵树，确切的说，应该是 Git 仓库中 HEAD 指向的版本）。  
  
OK，Git 的工作流程一般是酱紫：  
  
1. 在工作目录中添加、修改文件；  
  
2. 将需要进行版本管理的文件放入暂存区域；  
  
3. 将暂存区域的文件提交到 Git 仓库。  
  
因此，Git 管理的文件有三种状态：已修改（modified）、已暂存（staged）和已提交（committed），依次对应上边的每一个流程。  
  
等等……  
  
看到这里，有些鱼油肯定有疑惑：“你说 Git 仓库用于存放每次的版本迭代，我可以理解。但为何还要多增加一个暂存区域呢？”  
  
我这里打个比方：像某些厂家开发一个产品，一般他们都留有一手，不会把该产品的所有特性一次性发布。通过产品的迭代，每年秋季你就可以开开心心地买到又有一两项新功能的“新”产品了。  
  
IMG_263 我好像知道的太多了……  
  
So，有时你并不想把工作目录中所有的新功能都提交到最新版本，你就可以先添加一些本次需要提交的文件到暂存区，然后从暂存区中提交它们……所以暂存区在江湖中有个外号叫“索引”（Index）。  
  
****记住这三棵树，因为后边教的所有 Git 操作基本上都是在这三棵树之间搞来搞去！****

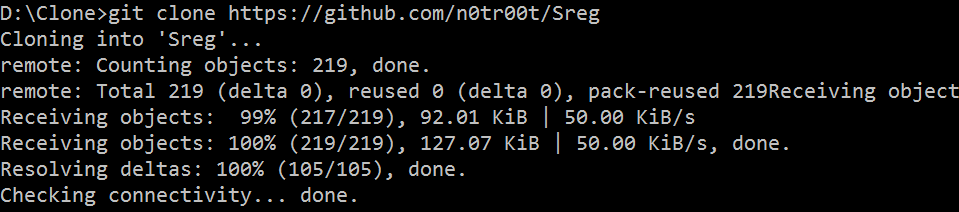
****初始化 Git****

首先要建立一个大本营（确定你的工作目录），这里我们在 D 盘新建一个叫做 MyProject 的新文件夹（这里你可以自己找个位置实验，但路径中尽量不要出现中文符号）。  
  
然后将命令行窗口的工作路径切换到刚才创建的 MyPorject 中，输入命令 git init 即可初始化 Git。  
  
   
  
然后你会看到 MyProject 文件夹中出现一个叫做 .git 的隐藏文件（这个文件夹就是 Git 用来跟踪管理版本迭代的）：  
  
   
  
看不到的童鞋请点击【查看】，勾选“隐藏的项目”，秘密立刻无所遁形：  
  
   
  
****Pay attention：这里边的东西眼看手勿动，否则项目毁了千万不要来找我(⊙o⊙)哦！！****  
  
好，现在****假装****我们在写一个大项目！  
  
写任何大项目之前，先要写一个 README.md 的文档（md 后缀是 Markdown 语言写的文本，现在可火啦 IMG_259）。  
  
打开后写入以下内容并保存：

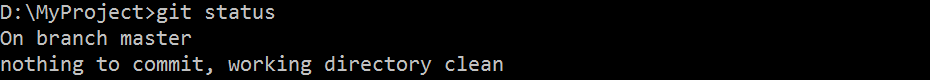
1. This is a big project!  
     
   ****将文件添加到暂存区域****  
     
   回到命令行窗口，输入 git add README.md 命令：  
     
      
     
   没有消息就是好消息，好消息就是你已经往成功迈出了一大步（姿势要帅）！  
     
   咱继续，现在你已经将 README.md 这个文件放入暂存区域了。  
     
     
   ****将文件提交到 Git 仓库****  
     
   输入 git commit -m "add a readme file" 将暂存区域里的东西提交到 Git 仓库中：  
     
      
     
   好了，这次有消息了。但并不是坏消息，只是 Git 告诉你有 1 个文件被改动（README.md），插入了 1 行内容。  
     
   及时反馈工作进度，是一个程序对用户的基本尊重！IMG_262   
     
   对了，commit 是提交的意思，-m 选项后边跟着的是本次提交的说明，就是大概描述这一版本做了哪些改动，以便今后可以迅速查看。  
     
   *需要注意的是：对于这个提交的说明，Git 是强制要求你必须写的。如果没有使用 -m 选项，Git 会自动打开一个编辑器，要求你在其中输入提交说明，输入完毕后保存退出即可（操作命令与 vim 编辑器一致）。*  
     
     
   总结一下，将工作目录的文件放到 Git 仓库只需要两步：  
     
   Step One -> git add 文件名  
     
   Step Two -> git commit -m "你干了啥"  
     
     
   ****彩蛋****

想不想知道在 Github 上看到非常赞的代码，要怎么偷偷地据为己有？  
  
回复“朕想知道”，小甲鱼来告诉你~

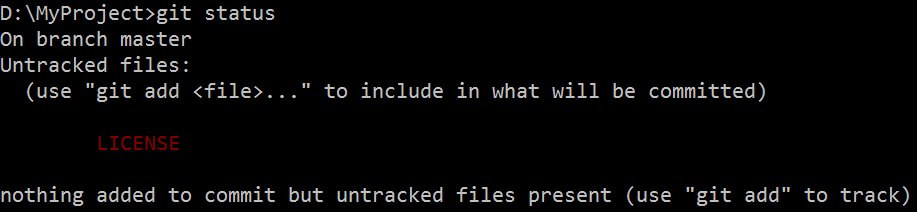
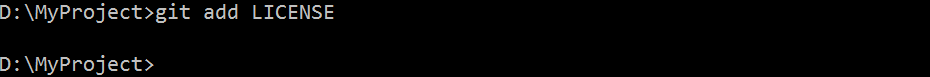
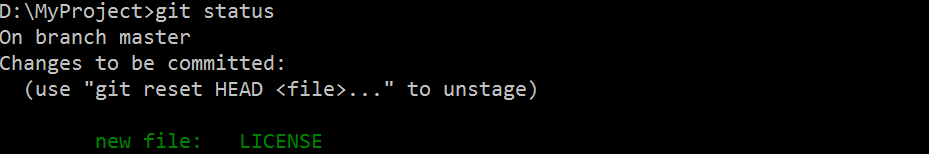
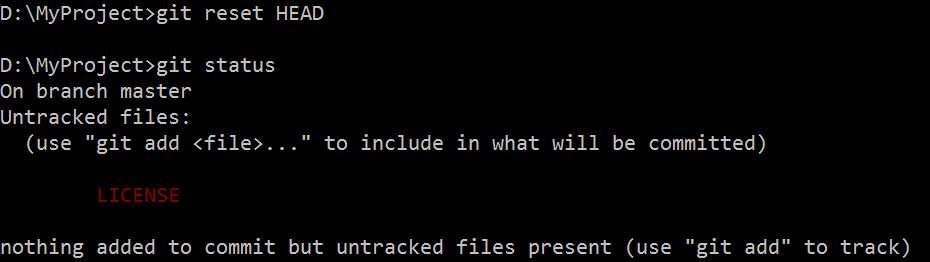
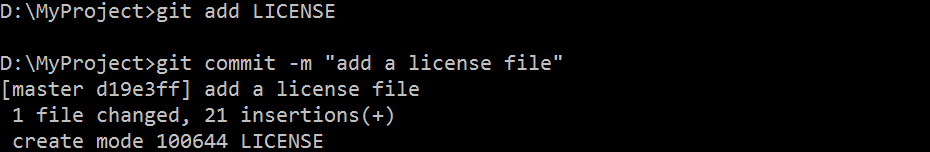
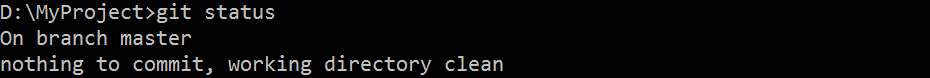
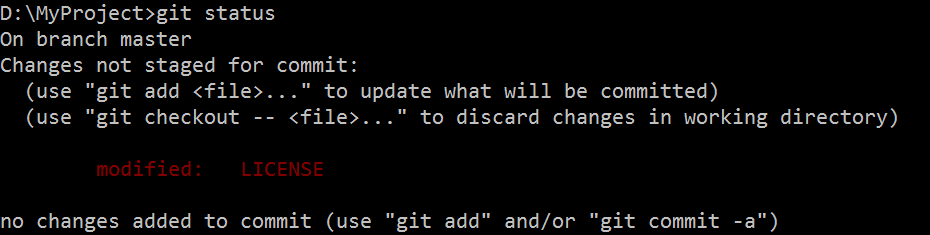
#### 本帖隐藏的内容

IMG_265 你确实很想知道……  
  
好吧，让我来告诉你~  
  
除了在空文件夹中初始化 Git，Git 还允许你将别人的库据为己有（是不是很给力，很霸道 IMG_266）  
  
要克隆一个现有的仓库，可不用跟当事人申请，直接输入 git clone 目标 命令即可。  
  
比如某天小甲鱼在 Github 上发现了一个叫 Sreg 的 Python 项目很赞，Sreg 可对使用者通过输入 email、phone、username 的返回用户注册的所有互联网护照信息！哇，So amazing！  
  
赶紧克隆下来：  
  
   
  
好了，源码已经全部如数拷贝下来了：  
  

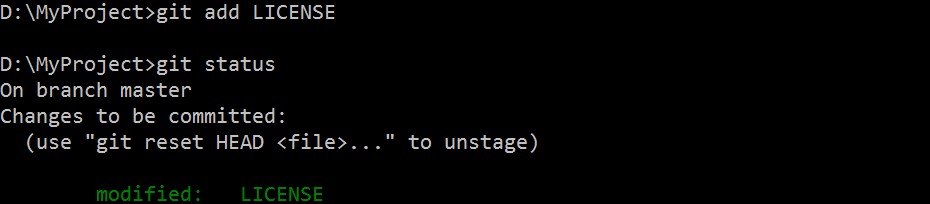
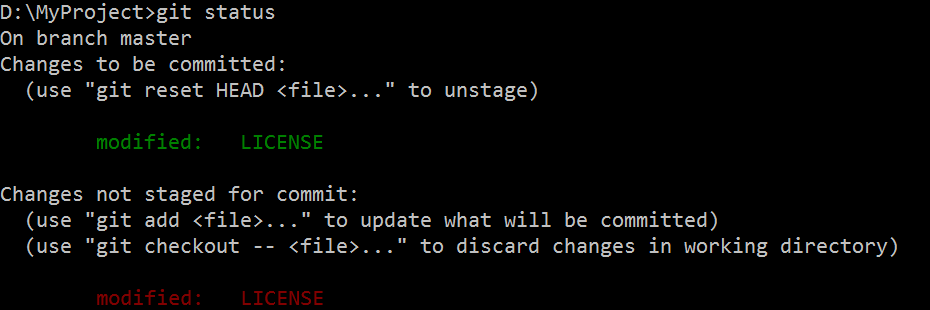

****状态****

随时随地，你都可以使用 git status 命令查看当前的状态……  
  
上一讲的代码如果你没有动过，那么输入 git status 命令后应该是下边这样：  
  
   
  
****On branch master**** 说明我们位于一个叫做“master”的分支里，这是默认的分支（相比其他版本管理程序，Git 的分支理念领先世界好几年 IMG_257 放心，在适当的时候小甲鱼会详细地给大家讲解分支的）。  
 ****nothing to commit, working directory clean**** 说明你的工作目录目前是“干净的”，没有需要提交的文件（意思就是自上次提交后，工作目录中的内容压根儿就没改动过）。  
  
既然要做****大项目****，那么应该增加一个版权声明。为了显得大公无私，我们采用 MIT 许可证给予用户最大的权利，让全世界的开发者都参与进来！  
  
在工作目录中增加 LICENSE 文件，内容如下：

Copyright (C) <year> <copyright holders>  
  
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:  
  
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.  
  
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

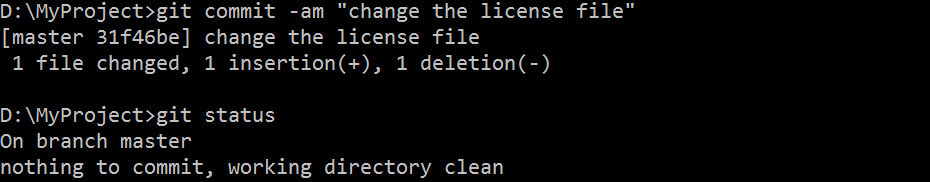
*小甲鱼友情提示：MIT 许可证几乎是最宽松的版权约定，一旦你的程序采用，也就意味着别人只要在软件包含上边的版权声明，就可以对你的程序为所欲为了（包括“使用、复制、修改、合并、出版发行、散布、再授权和/或贩售软件及软件的副本”）。*  
  
现在我们输入 git status 命令，看下 git 有什么想法：  
  
   
  
****Untracked files**** 说明存在未跟踪的文件（下边红色的那个）  
  
所谓的“未跟踪”文件，是指那些新添加的并且未被加入到暂存区域或提交的文件。它们处于一个逍遥法外的状态，但你一旦将它们加入暂存区域或提交到 Git 仓库，它们就开始受到 Git 的“跟踪”。  
  
这里圆括号中的英文是 git 给我们的建议：使用 git add <file> 命令将待提交的文件添加到暂存区域。（然后 Git 就可以对它们嘿嘿嘿 IMG_259）  
  
那我们不妨按照建议来操作：  
  
   
  
add 命令如果执行成功，并不会有提示消息。  
  
再来看一下 git 的当前状态：  
  
   
  
大家看到，首先是文件被绿了（都说是给 Git 嘿嘿嘿了嘛）  
  
****use "git reset HEAD <file>..." to unstage**** 的意思是“如果你反悔了，你可以使用 git reset HEAD 命令恢复暂存区域”。  
  
IMG_262 试试又不会怀孕：  
  
   
  
还真回去了！IMG_264  
  
好吧，再次添加到暂存区域，然后执行 git commit -m "add a license file" 命令：  
  
   
  
再次查看状态：  
  
   
  
OK，再次干净了。  
  
等等，你别走……  
  
突然发现版权那块忘了写上咱自家的名字了……  
  
打开 LICENSE 文件，将 ****Copyright (C) <year> <copyright holders>**** 改为 ****Copyright (C) 2016 FishC****，保存……  
  
执行 git status 命令：  
  
   
  
由于你对工作目录的文件进行了修改，导致这个文件和暂存区域的对应文件不匹配了，所以 Git 又给你提出两条建议：

* 使用 git add 命令将工作目录的新版本覆盖暂存区域的旧版本，然后准备提交
* 使用 git checkout 命令将暂存区域的旧版本覆盖工作目录的新版本（危险操作：相当于丢弃工作目录的修改）

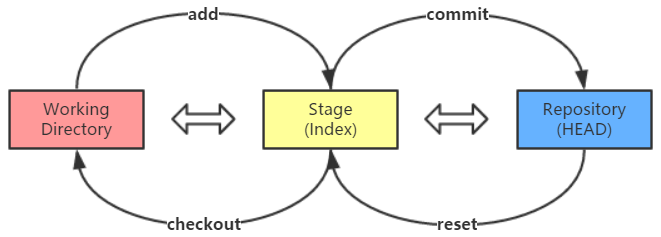
还有一种情况我们没分析，大家先把新版本的文件覆盖掉暂存区域的旧版本：  
  
   
  
然后我们打开 LICENSE 文件，将 ****FishC**** 改为 ****FishC.com****，保存……  
  
再次查看状态：  
  
   
  
这次诡异了：被绿的 LICENSE 说明文件存放在暂存区域（待提交），同时红色的 LICENSE 说明文件还在工作目录等待添加到暂存区域。  
  
这种情况你应该意识到这里存在两个不同版本的 LICENSE 文件，这时如果你直接执行 commit 命令，那么提交的是暂存区域的版本（FishC），如果你希望提交工作目录的新版本（FishC.com），那么你需要先执行 add 命令覆盖暂存区域，然后再提交……  
  
烦不烦？！IMG_270  
  
老是要 add 再 commit，很烦恼吧？  
  
有没有办法一步到位？  
  
答案当然是 Yes！  
  
  
****彩蛋****

回复“朕想知道”，小甲鱼来告诉你~

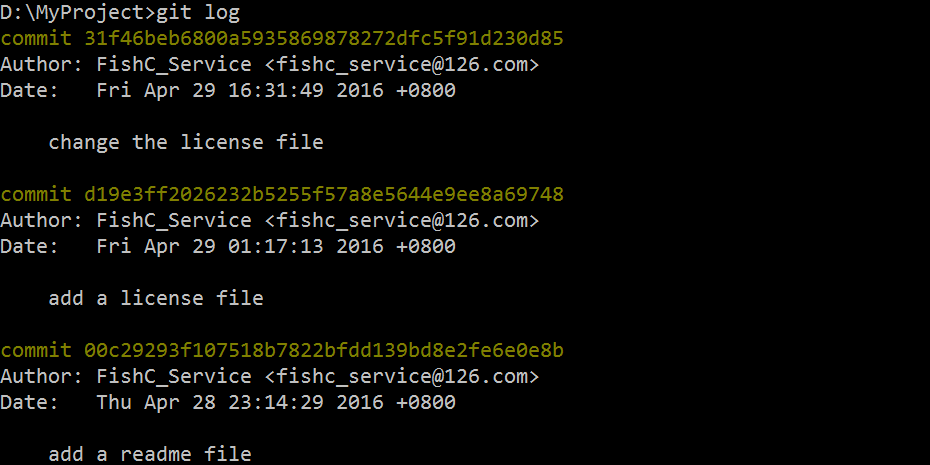
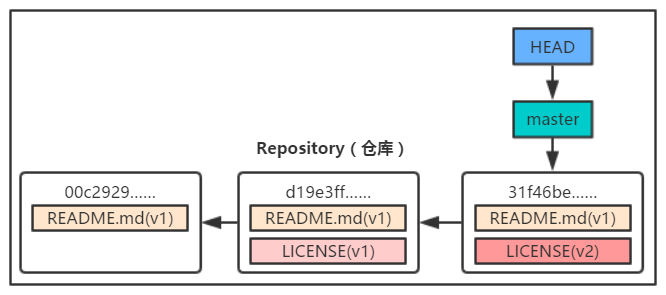
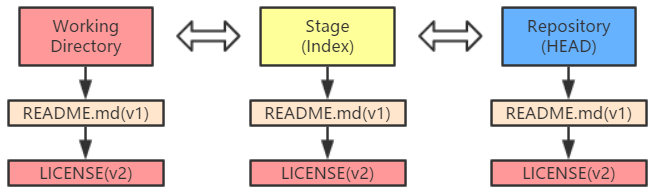
#### 本帖隐藏的内容

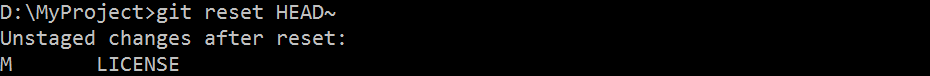
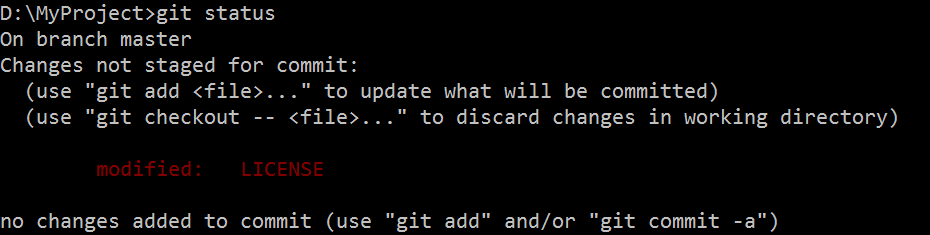
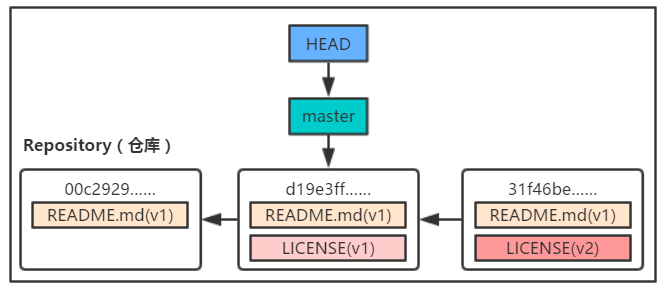
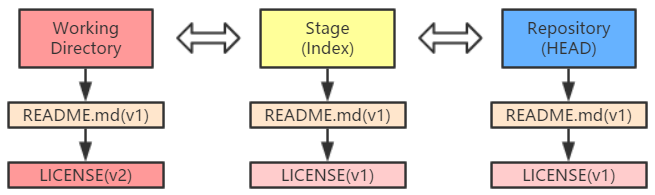
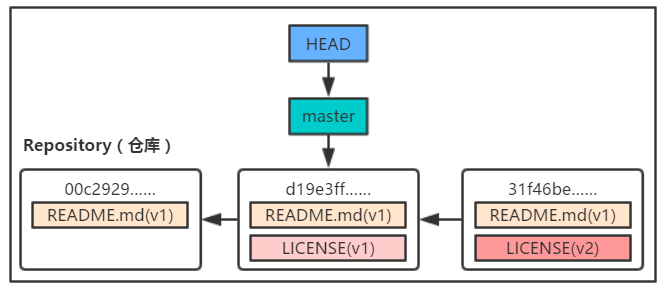
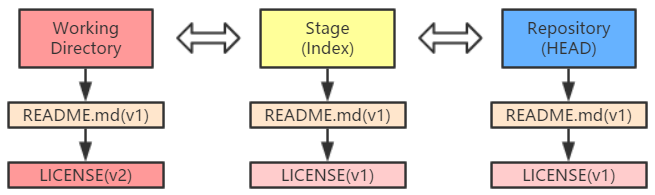
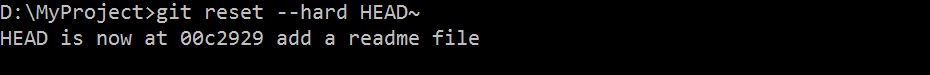
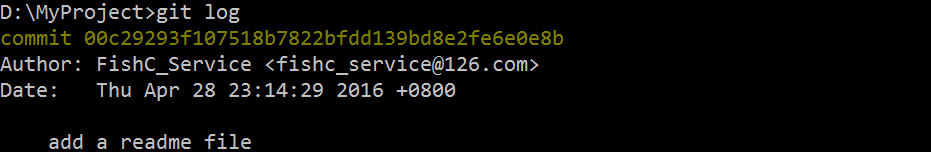
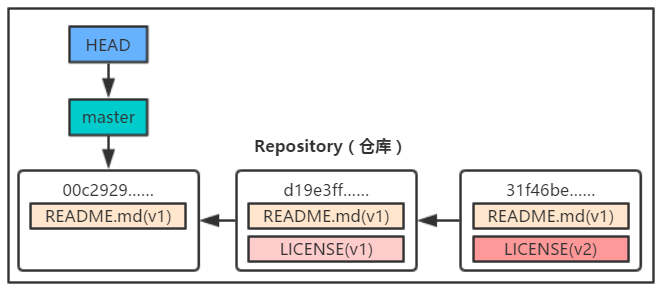
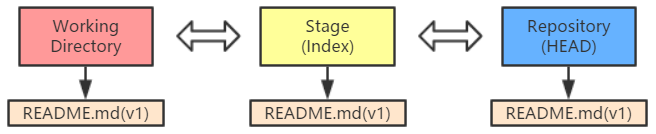
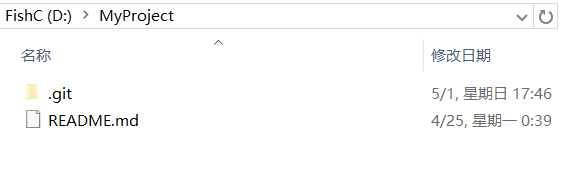
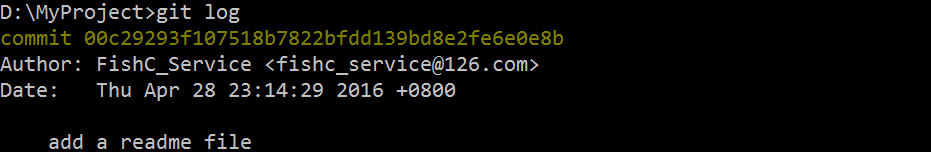
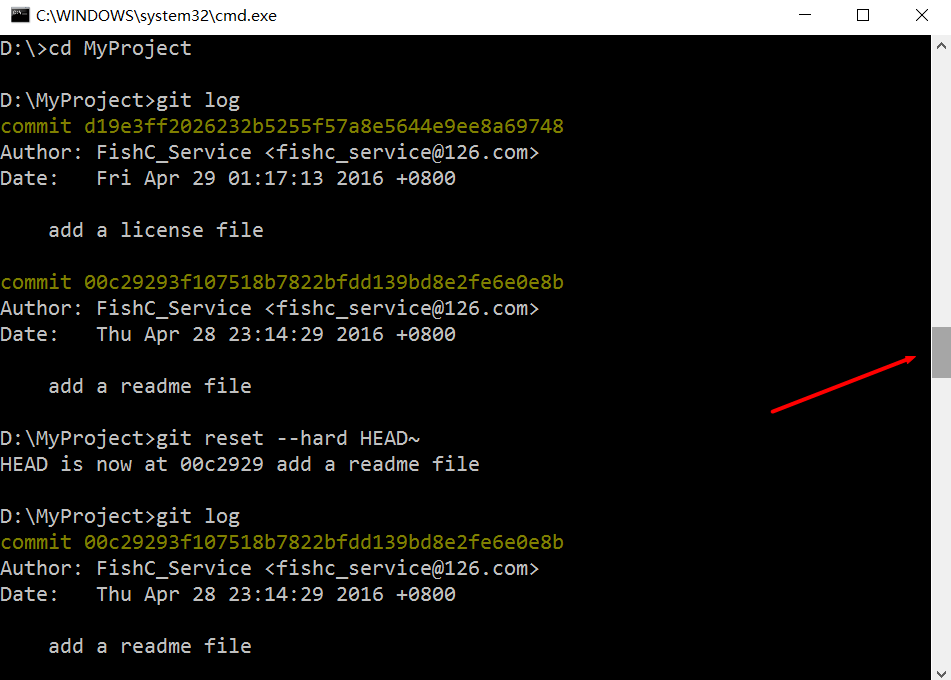
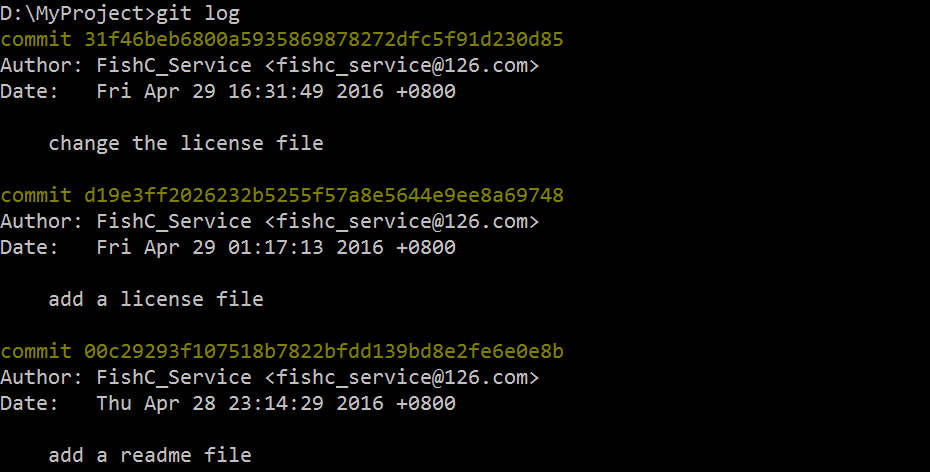
IMG_271 你确实很想知道……  
  
来，教你一步到位的技巧：在 commit 的时候，加上 -a 选项即可。  
  
执行 git commit -am "change the license file" 命令：  
  
   
  
****解释：****  
  
-a 选项其实是专为懒人添加的。添加了 -a 选项，Git 会自动帮你将工作目录中所有“已跟踪”的文件先 add 到暂存区域，然后再执行 commit 命令。另外，-am 是 -a -m 的缩写，多个选项可以写在一起，懒人嘛~

****回到过去****

上一讲我无意中接触到了两个有关回退的命令：reset 和 checkout  
  
来，小甲鱼帮你捋一捋思路：  
  
   
  
现在几个命令应该相当清晰了：

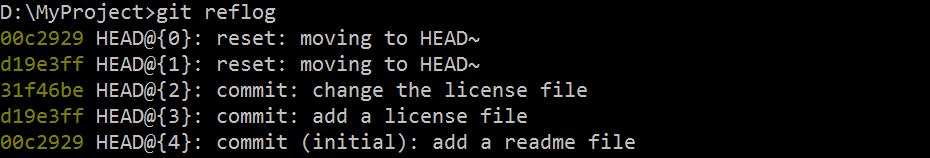
* git add 命令用于把工作目录的文件放入暂存区域
* git commit 命令用于把暂存区域的文件提交到 Git 仓库
* git reset 命令用于把 Git 仓库的文件还原到暂存区域
* git checkout 命令用于把暂存区域的文件还原到工作目录

前边两个命令我有信心你已经相当熟悉了，但后边两个千万别说你已懂，因为它们是 Git 里边最复杂的命令（它们的功能可不止上边文字描述的这么简单 IMG_257）  
  
****先给大家重点讲解 reset 命令，checkout 命令在分支管理中再细讲。****  
  
   
  
先执行 git log 命令查看历史提交：  
  
   
  
这里记录了我们之前的 3 次提交（排序是按时间从近到远的），****Author**** 后边是提交者，****Date**** 后边是提交日期，下边是当次提交的说明。那……草黄色的那个 commit +“乱码”是什么鬼？IMG_260   
  
咳咳~~  
  
这里并没有什么“乱码”，这个是 Git 为每次提交计算出来的 ID，它其实一个完整的 SHA-1 校验和（尽管你的文件内容可能跟我的完全一致，但这个值却不一样，这是因为账号、时间不同而导致）。你不需要知道 SHA-1 的原理，只需要知道它在任何时候都是唯一的，通过这个 ID，你就可以找到对应的那个版本。  
  
有鱼油可能会问：为何 ID 不是传统的 1、2、3、4、5…… 难道作者是为了装 X？  
  
非也非也，因为 Git 是分布式的版本控制系统，如果多人同时工作，那么使用 1、2、3、4、5 就很容易产生冲突，继而打群架……  
  
顺便说一些，你在论坛首页看到的一串 16 进制字符串也不是什么“乱码”，其实是个深藏已久的秘密：  
  
   
（有人居然说小甲鱼网页的 bug 放了一整年都不补上 IMG_262）  
  
根据 log 记录，现在我们将 Git 仓库如果可视化，应该是这样子：  
  
   
  
三棵树现在应该是下面酱紫：  
  
   
  
  
****回滚快照****

*注：快照即提交的版本，每个版本我们称之为一个快照。*  
  
现在我们利用 reset 命令回滚快照，并看看 Git 仓库和三棵树分别发生了什么。  
  
执行 git reset HEAD~ 命令：  
  
*注：HEAD 表示最新提交的快照（31f46），而 HEAD~ 表示 HEAD 的上一个快照（d19e3）*  
  
   
  
然后执行 git status 命令查看现在的状态：  
  
   
  
有哪位童鞋可以回答我：现在我们的快照（d19e3）回滚到了哪一棵树里？  
  
。  
。  
。  
  
答案是：第二棵树（暂存区域）！  
  
有些鱼油可能会持不同意见：不应该是回滚到第一棵树（工作目录）吗？你看，Git 不是写得很清楚吗 -> ****Changes not staged for commit****，它还好心提醒我们使用 add 命令将修改添加到暂存区域丫！  
  
其实真相是这样的：我们执行 git reset HEAD~ 命令之后，快照（d19e3）回滚到暂存区域，此时工作目录里存放的却是最新的文件（31f46）。由于 Git 会跟踪文件的变化，所以执行 git status 命令时，git 发现工作目录中的文件比暂存区域的要新（对比日期），所以才有这样的提示……  
  
好了，现在执行完 git reset HEAD~ 命令之后，Git 仓库应该是这样子：  
  
   
  
三棵树现在应该是下面酱紫：  
  
   
  
这里有一点要补充的：HEAD~ 表示 HEAD 的上一个快照（d19e3），HEAD~~（00c29）则表示 HEAD 的上上一个快照，如果希望表示上上上上上上上上上上一个快照（数了一下，这里有 10 个“上”IMG_269 ），那么可以直接用 HEAD~10 来表示。  
  
git reset HEAD~ 命令其实是 git reset --mixed HEAD~ 的缩写，因为 --mixed 选项是默认的，所以我们可以偷懒。  
  
我们发现，git reset HEAD~ 命令其实影响了两棵树：首先是移动 HEAD 的指向，将其指向上一个快照（HEAD~）；然后再将该位置的快照回滚到暂存区域。  
  
为了灵活地操纵这三棵树，Git 还为 reset 命令安排了 --soft 和 --hard 选项，可谓软硬兼施，不到你不服~  
  
****--soft 选项****  
加上 --soft 选项的结果是使得 reset 变“软”了，也就没有原来那么持久……IMG_270  
  
So， git reset --soft HEAD~ 命令就相当于只移动 HEAD 的指向，但并不会将快照回滚到暂存区域。  
  
这个选项有什么作用呢？  
  
事实它就是相当于撤消了上一次的提交（commit）。  
  
一不小心提交了，后悔了，那么你就执行 git reset --soft HEAD~ 命令即可（此时执行 git log 命令，也不会再看到已经撤消了的那个提交）。  
  
****--hard 选项****  
  
加上 --hard 选项的结果是使得 reset 变“硬”……  
  
你猜的不错，加上 --hard 选项，reset 不仅移动 HEAD 的指向，将快照回滚动到暂存区域，它还将暂存区域的文件还原到工作目录。  
  
来，上点图吧！  
  
刚才执行完 git reset HEAD~ 命令后，Git 仓库里的数据是这样：  
  
   
  
三棵树是这样：  
  
   
  
那么在这种状态下，我再执行 git reset --hard HEAD~ 命令：  
  
   
  
Git 仓库中就剩下最后一个快照了：  
  
   
  
还原案发现场，Git 仓库现在应该是这样：  
  
   
  
而三棵树现在应该都被回归到第一个版本（00c2929）：  
  
   
  
不信？自己瞧瞧你的文件夹：  
  
   
  
****最后总结一下：reset 回滚快照三部曲****  
  
1. 移动 HEAD 的指向（--soft）  
  
2. 将快照回滚到暂存区域（[--mixed]，默认）  
  
3. 将暂存区域还原到工作目录（--hard）  
  
  
****回滚指定快照****  
  
如果快照比较多，你又懒得去数有多少个“上”，那么你可以通过指定具体的快照 ID 来回滚该快照。  
  
比如 git reset 00c2929  
如上，你不必把辣么长的 ID 号都给输入进去，一般只要输入前几位（5 位或以上吧）就可以了。IMG_278   
  
  
****回滚个别文件****  
  
reset 不仅可以回滚指定快照，还可以回滚个别文件。  
  
命令格式为 git reset 快照 文件名/路径  
  
这样，它就会将忽略移动 HEAD 的指向这一步（因为你只是回滚快照的部分内容，并不是整个快照，所以 HEAD 的指向不应该发生改变），直接将指定快照的指定文件回滚到暂存区域。  
  
  
****不仅可以往回滚，还可以往前滚！****  
  
这里需要强调的是：reset 不仅是一个“复古”的命令，它不仅可以回到过去，还可以去到“未来”。  
  
唯一的一个前提条件是：你需要知道指定快照的 ID 号。  
  
现在执行 git log 命令只剩下一个最原始的提交了：  
  
   
  
但是将命令行窗口向上拉，我们可以喵到之前提交的几个版本 ID 号（聪明如我 IMG_280）：  
  
   
  
所以我们可以执行 git reset --hard 31f46be 命令：  
  
IMG_282   
  
再次执行 git log 命令：  
  
   
  
我们又回到了最新的版本！  
  
是的，我们就这样在历史的长河里滚来滚去……  
  
但是……故事还没完，如果某天你 reset --hard 将工作目录回滚到了某个版本，但特么的隔天你就后悔了（有封写给小花的情书也放里边）！此时命令行窗口早已关闭，你又没用小本本把每次 commit 的 ID 号给记下来，这可肿么办才好？  
  
  
****彩蛋****

回复“朕想知道”，小甲鱼来告诉你~

#### 本帖隐藏的内容

IMG_284 你确实很想知道……  
  
reflog 命令可以拯救你！  
  
执行 git reflog 命令，告诉我，你看到了什么：  
  
   
  
没错，Git 偷偷记录下了你每一次的操作（无耻小人），第一列就是每次执行完命令，HEAD 指向的版本 ID 号啦~

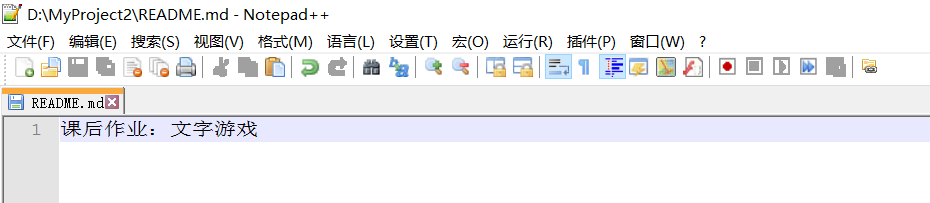
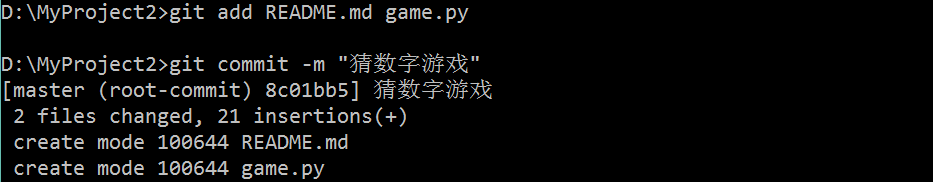
**版本比较（有彩蛋）**

****准备工作****

****Step One：****  
  
创建一个叫做 MyProject2 的新文件夹作为本次演示的项目，初始化 Git（方法跟 [Git实用教程4：实战](http://bbs.fishc.com/thread-71305-1-1.html" \t "https://fishc.com.cn/_blank) 一致，忘了的童鞋请回滚教程……）。  
  
****Step Two：****  
  
现在模拟我们在完成《零基础入门学习Python》的一道课后作业题……  
  
创建一个 game.py 的文本，将下边代码拷贝进去：

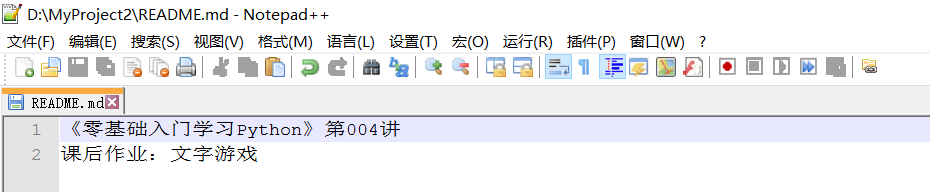
1. import random
2. print('------------------我爱鱼C工作室------------------')
3. secret = random.randint(1,10)
4. temp = input("不妨猜一下小甲鱼现在心里想的是哪个数字：")
5. guess = int(temp)
6. while guess != secret:
7. temp = input("哎呀，猜错了，请重新输入吧：")
8. guess = int(temp)
9. if guess == secret:
10. print("卧槽，你是小甲鱼心里的蛔虫吗？！")
11. print("哼，猜中了也没有奖励！")
12. else:
13. if guess > secret:
14. print("大了，大了~~~")
15. else:
16. print("小了，小了~~~")
18. print("游戏结束，不玩啦^\_^")

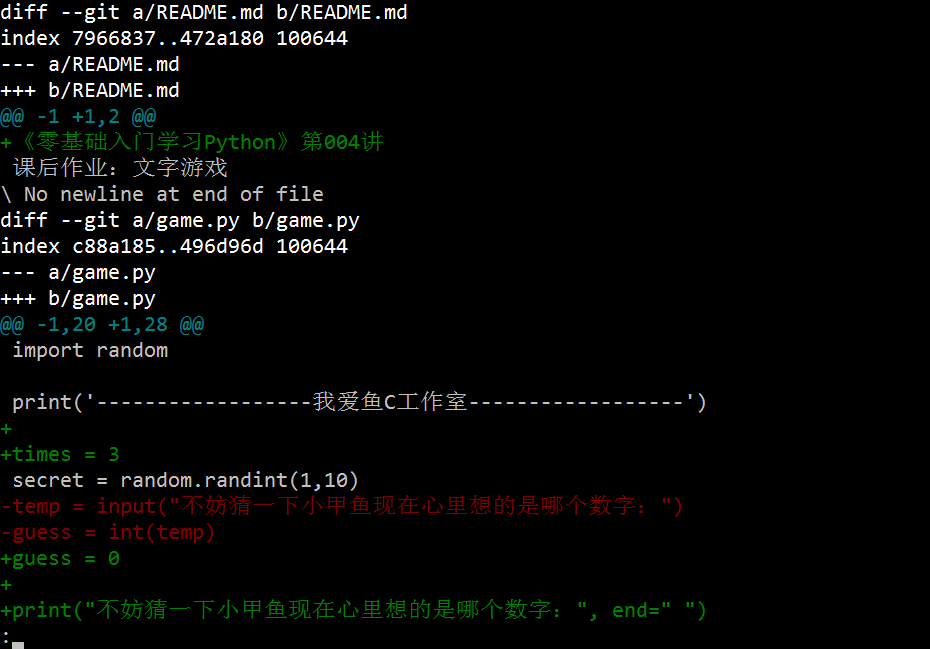
复制代码

再创建一个 README.md 文件，写清楚这是一个课后作业的项目：  
  
   
  
****Step Three：****  
  
执行 git add README.md game.py 命令将文件添加到暂存区域，接着执行 git commit -m "猜数字游戏" 提交项目的第一个快照（你不知道吗？Git 一直都是支持中文的呀 IMG_257）：  
  
   
  
****Step Four：****  
  
接着我们按照课后作业的要求改写代码如下：

1. import random
2. print('------------------我爱鱼C工作室------------------')
3. times = 3
4. secret = random.randint(1,10)
5. guess = 0
6. print("不妨猜一下小甲鱼现在心里想的是哪个数字：", end=" ")
7. while (guess != secret) and (times > 0):
8. temp = input()
9. guess = int(temp)
10. times = times - 1 # 用户每输入一次，可用机会就-1
11. if guess == secret:
12. print("哇哦，你是小甲鱼心里的蛔虫吗？！")
13. print("哼，猜中了也没有奖励！")
14. else:
15. if guess > secret:
16. print("哥，大了大了~~~")
17. else:
18. print("嘿，小了，小了~~~")
19. if times > 0:
20. print("再试一次吧：", end=" ")
21. else:
22. print("机会用光咯T\_T")
24. print("游戏结束，不玩啦^\_^")

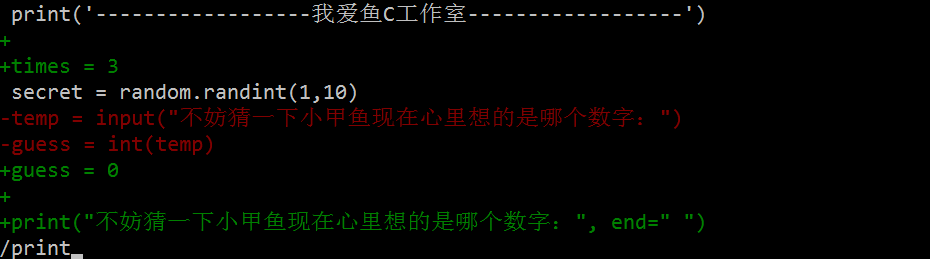
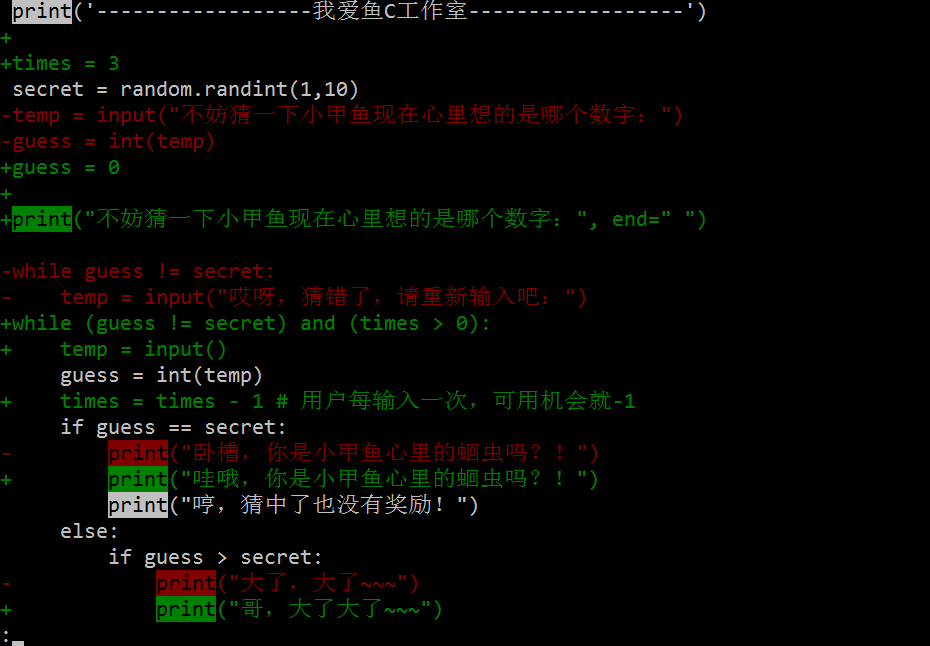
复制代码

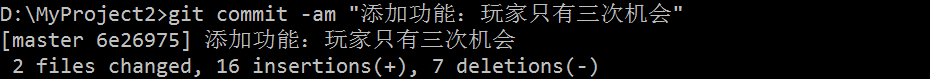
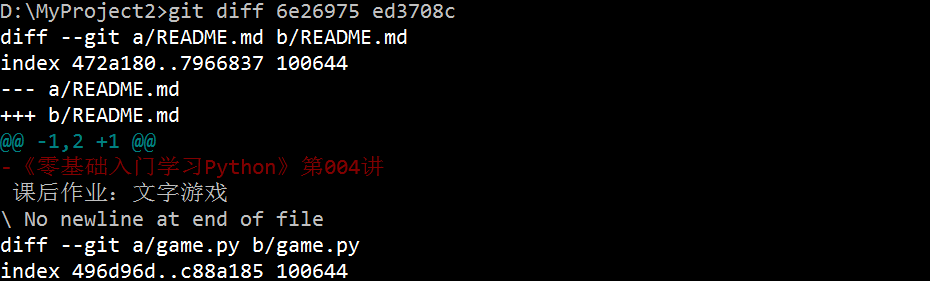
README.md 文件也改一改：  
  
   
  
  
****比较暂存区域与工作目录****

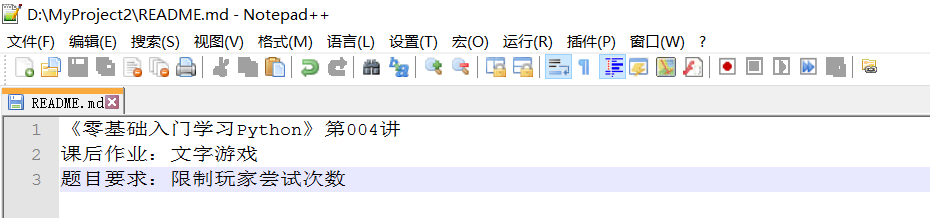
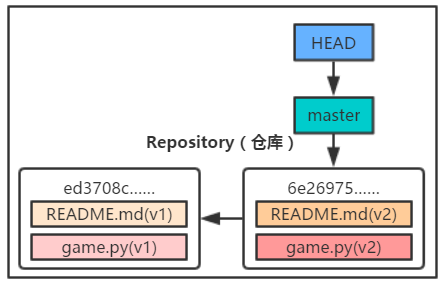
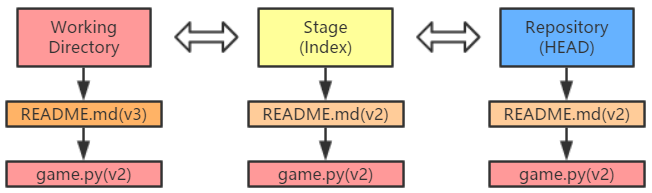
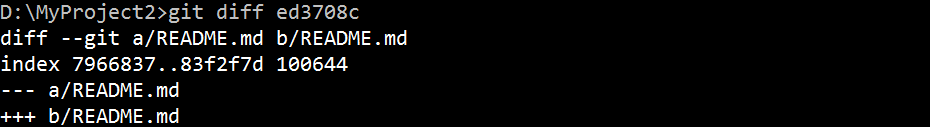
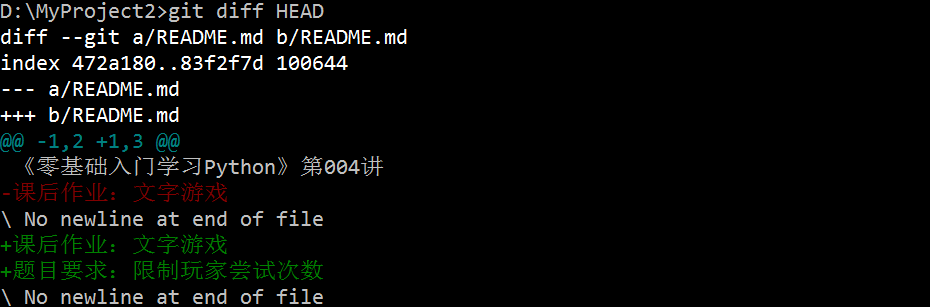
直接执行 git diff 命令是比较暂存区域与工作目录的文件内容：  
  
   
  
IMG_261 What？？？  
  
不急，我一个个给你解释……  
  
首先，diff 命令是 linux 上非常重要的工具，用于比较文件的内容，特别是比较两个版本不同的文件以找到改动的地方。diff 程序的输出被称为补丁 (patch)，因为 Linux 系统中还有一个 patch 程序，可以根据 diff 的输出将 a.c 的文件内容更新为 b.c。  
  
diff 现在已经是 Git 不可或缺的一部分了！值得一提的是，****现在的 diff 已经可以很好的支持二进制文件了****，比如 docx 文件以前不支持，现在都支持啦~\(≧▽≦)/~  
  
第一行：diff --git a/README.md b/README.md  
  
表示对比的是存放在暂存区域的 README.md 和工作目录的 README.md  
  
第二行：index 7966837..472a180 100644  
  
表示对应文件的 ID 分别是 7966837 和 472a180，左边暂存区域，后边当前目录。  
  
最后的 100644 是指定文件的类型和权限。喏，具体的定义我也帮你找出来了（Linux 系统的定义，看不懂没关系，我知道有人需要，所以顺便贴出来）：

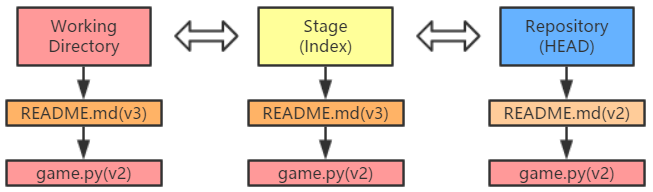
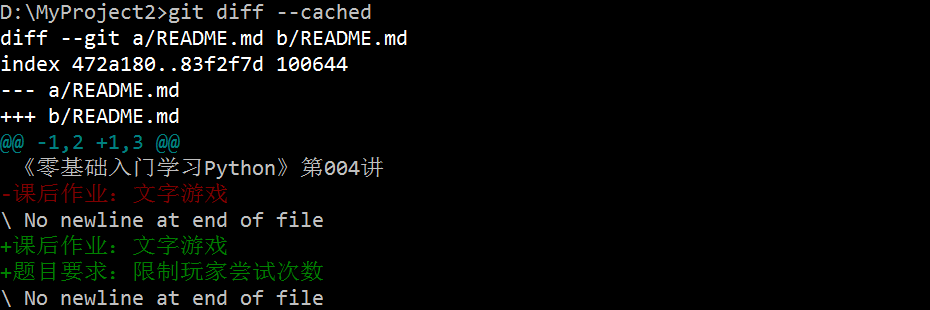
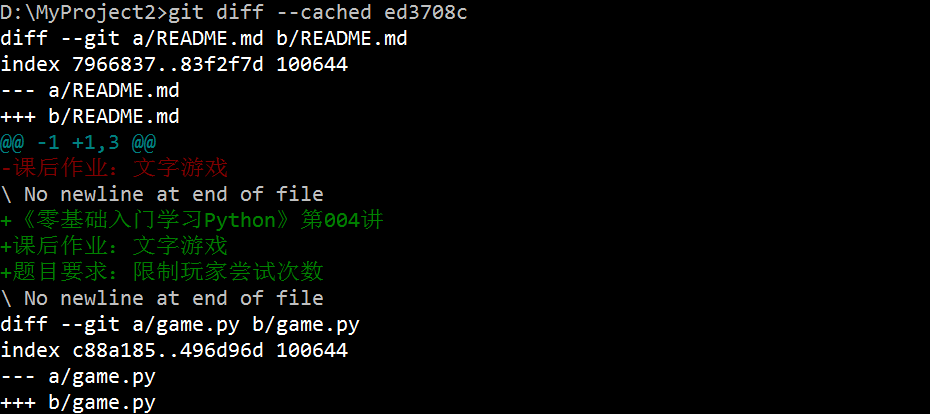
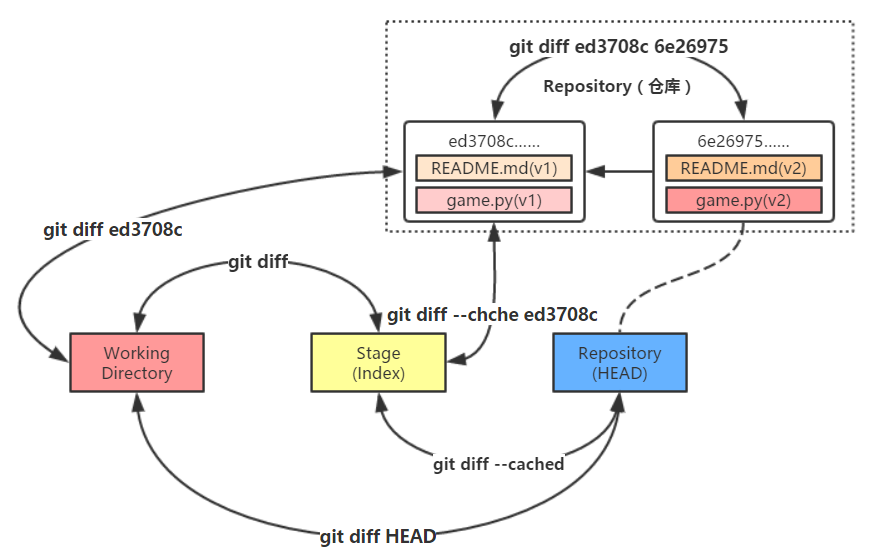
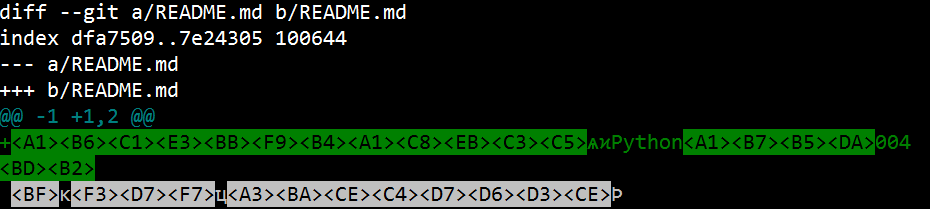
1. The following flags are defined for the st\_mode field:
2. S\_IFMT     0170000   bit mask for the file type bit fields
3. S\_IFSOCK   0140000   socket
4. S\_IFLNK    0120000   symbolic link
5. S\_IFREG    0100000   regular file
6. S\_IFBLK    0060000   block device
7. S\_IFDIR    0040000   directory
8. S\_IFCHR    0020000   character device
9. S\_IFIFO    0010000   FIFO
10. S\_ISUID    0004000   set UID bit
11. S\_ISGID    0002000   set-group-ID bit (see below)
12. S\_ISVTX    0001000   sticky bit (see below)
13. S\_IRWXU    00700     mask for file owner permissions
14. S\_IRUSR    00400     owner has read permission
15. S\_IWUSR    00200     owner has write permission
16. S\_IXUSR    00100     owner has execute permission
17. S\_IRWXG    00070     mask for group permissions
18. S\_IRGRP    00040     group has read permission
19. S\_IWGRP    00020     group has write permission
20. S\_IXGRP    00010     group has execute permission
21. S\_IRWXO    00007     mask for permissions for others (not in group)
22. S\_IROTH    00004     others have read permission
23. S\_IWOTH    00002     others have write permission
24. S\_IXOTH    00001     others have execute permission

复制代码

第三行：--- a/README.md  
  
--- 表示该文件是旧文件（存放在暂存区域）  
  
第四行：+++ b/README.md  
  
+++ 表示该文件是新文件（存放在工作区域）  
  
第五行：@@ -1 +1,2 @@  
  
以 @@ 开头和结束，中间的“-”表示旧文件，“+”表示新文件，后边的数字表示“开始行号，显示行数”  
  
IMG_262 一脸懵逼。。。  
  
不要紧，先看接着的六、七行的内容：  
  
IMG_263   
  
这是将两个文件合并显示的结果，前边有个 + 的绿色的那一行说明是新文件独有的，浅灰色的则是两个文件所共有的内容。  
  
所以，+1,2 表示新文件在合并显示中从第 1 行开始，显示 2 行。  
  
那为啥 -1 后边没有显示的行数？  
  
因为在合并显示的结果中，旧文件已经完全包含在新文件中了（也就是旧文件没有自己“独有的”内容）。  
  
第八行：\ No newline at end of file  
  
这是 Git 出于善意的提醒：文件不是以换行符结束。  
  
为什么会有这样多此一举的提醒呢？  
  
仔细推敲下你就会明白了：diff 将两个文件合并打印，到达最后一个字符的时候，无论文件中是否存在换行符，diff 都需要打印一个换行符！为啥？为了好看呗！！所以如果你的文件最后一个字符不是以换行符结尾，但 diff 又自行添加了一个换行符，所以它觉得有义务提醒你它这么做了！  
  
****TIPS：最后空一行是一个良好的编程习惯****  
  
再下边就是另外两个文件（game.py）的比较了，理解的方式跟上边一样。  
  
两个 @@ 之间表示下方合并显示的范围，@@ -1,20 +1,28 @@ 表示旧文件从第 1 行开始，显示 20 行；新文件从第 1 行开始，显示 28 行。  
  
不信你可以数数看：1，2，3，4，5，6  
  
咦……  
  
这个点点（:）是什么鬼？IMG_264   
  
意思是世界太大，窗口太小，没法全部显示，正在等待你的命令……  
  
****移动命令****  
  
这时候你可以输入下边的移动命令：  
  
按下 j 按键表示向下移动一行，按下 k 按键则是向上移动一行。当然你也可以使用上、下方向键，但小甲鱼不建议你这么做，因为按方向键意味着你的右手需要离开主键盘区……这一来一回很消耗时间和节奏的（论 Vim or Emacs 习惯的重要性 IMG_265）！  
  
怎么？嫌移动的速度太慢？  
  
好，试试 f 和 b 两个按键！  
  
f 表示向下一页（一个窗口的内容）；b 表示向上一页的移动。  
  
一页又太多了？半页可否？  
  
可以！d 表示向下移动半页；u 表示向上移动半页。  
  
****跳转命令****  
  
是否可以直接跳转到开头和结尾呢？  
  
可以，按 g 去到第一行，按 G 去到最后一行。  
  
不这么极端行不行？  
  
可以，先输入数字 3，再按下 g，表示去到第 3 行。  
  
****搜索命令****  
  
比如想看下关于 print 函数的内容有没有变动，可以使用搜索命令。  
  
输入斜杠（/）或问号（?），后边输入搜索的关键字：  
  
   
  
按下回车，所有匹配的关键字均高亮显示：  
  
   
  
然后斜杠（/）和问号（?）的区别主要是搜索方向的不同：斜杠（/）表示从当前位置向下搜索，问号（?）表示从当前位置向上搜索。  
  
接着输入 n 表示顺着当前的搜索方向快速跳转到下个匹配的位置，大写的 N 则是与当前搜索方向相反。  
  
****退出和帮助****  
  
在点点（:）后边输入 q，表示退出 diff；输入 h 表示进入帮助界面，你会看到很多命令和功能（当然都是鸟文），输入 q 可以退出帮助界面。  
  
其实习惯于 Linux 终端界面的鱼油们一定对这些并不陌生……  
  
这些命令也好，操作也罢，都是赤裸裸的文化入侵哈（让你们用 Windows，刚开始肯定无比蛋疼，但习惯 Linux 的大牛们不知道用得多欢快）~~~  
  
  
****比较两个历史快照****

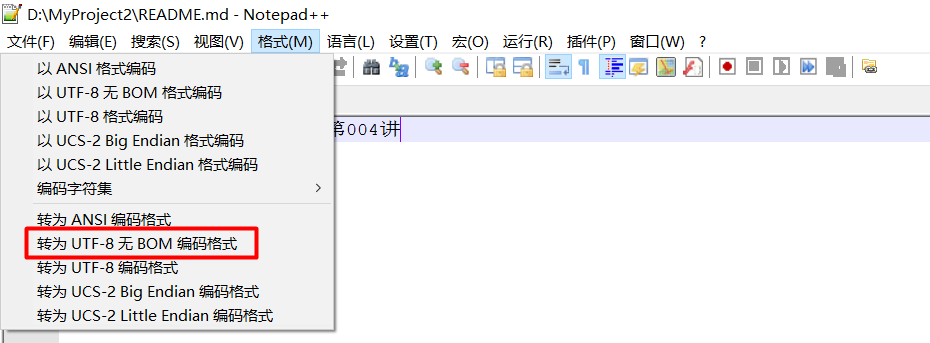
Git 仓库中存放一个个提交的快照，我们能不能对比其中的任意两个呢？  
  
当然可以！  
  
我们执行 git commit -am "添加功能：玩家只有三次机会" 命令，添加并提交工作目录中的所有文件：  
  
   
  
执行 git log 命令，可以看到现在 Git 仓库中已经有两个快照了：  
  
   
  
执行 git diff 6e26975 ed3708c 命令，即可比较 Git 仓库中两个快照的差异：  
  
   
  
  
****比较当前工作目录和 Git 仓库中的快照****

现在我们稍微改动一下 README.md 文件的内容：  
  
   
  
目前我们的 Git 仓库应该是酱紫：  
  
   
  
三棵树是这样：  
  
   
  
如果希望比较第一份快照（ed3708c）和当前目录的内容，输入 git diff ed3708c 命令即可：  
  
   
  
如果希望比较最新提交的快照和当前目录的内容，输入 git diff HEAD 命令即可：  
  
   
  
  
****比较暂存区域和 Git 仓库快照****

执行 git add README.md 命令，将第三版的 README.md 文件添加到暂存区域。  
  
然后三棵树就是这样了：  
  
   
  
如果希望比较最新提交的快照和暂存区域的文件，只需要执行 git diff --cached 命令：  
  
   
  
当然也可以指定其他快照，就是需要多写上一个 ID 值：  
  
   
  
最后，给大家奉上小甲鱼梳理的终极奥义图：  
  
   
  
IMG_280  
  
使用 Windows 编辑的朋友应该会遇到下边的怪事儿：  
  
   
  
这****绿色的****和****灰色的****的乱码是什么鬼？  
  
肿么跟小甲鱼演示的就不一样了？  
  
  
****彩蛋****

回复“朕想知道”，让小甲鱼来告诉你真相吧！

#### 本帖隐藏的内容

在 Windows 下，如果你看到乱码，乃是由于 Windows 的不团结导致编码问题，请使用全世界通用的 UTF-8 无 BOM 格式。  
  
使用 Nodepad++ 打开文件，点击“格式(M)”->“转为 UTF-8 无 BOM 编码格式”：  
  


****6.png**** (35.21 KB, 下载次数: 4)

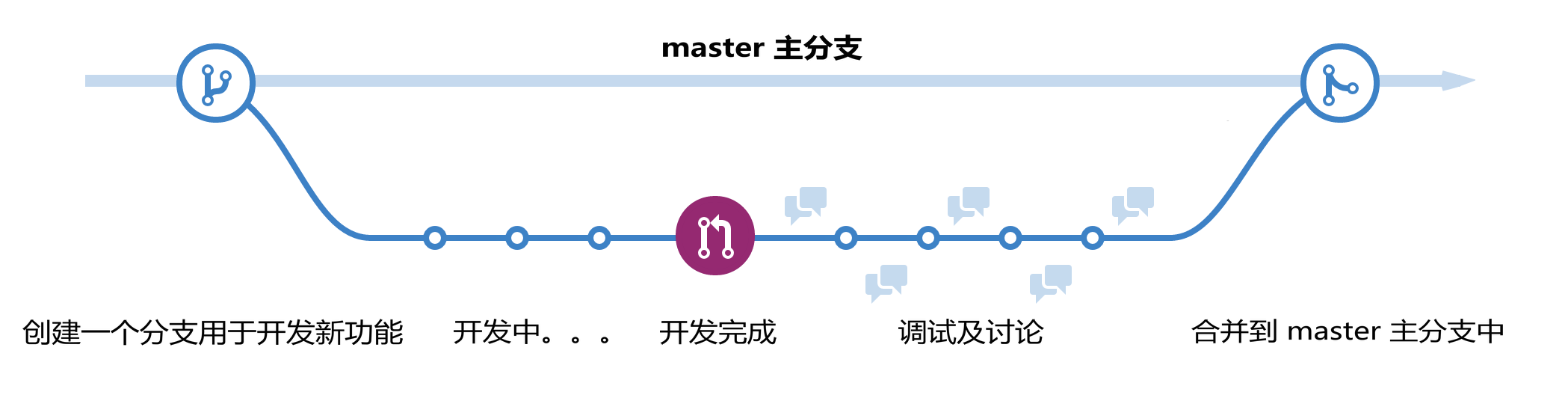
[下载附件](https://fishc.com.cn/forum.php?mod=attachment&aid=NDUyMzJ8ZThmODE0Nzh8MTU2MTk2ODMyOHw1NzY1NjV8NzE3MTg=&nothumb=yes" \t "https://fishc.com.cn/_blank)  [保存到相册](https://fishc.com.cn/javascript:;)

2016-5-8 02:03 上传

|  |  |
| --- | --- |
| |  | | --- | | ****修改最后一次提交****  实际开发中，你可能会遇到下边这两种情况：  Situation One：版本刚一提交（commit）到仓库，突然想起漏掉两个文件还没有添加（add）。就好比你是老司机，说开车就开车，到了半路发现还有俩乘客没上来……  Situation Two：版本刚一提交（commit）到仓库，突然想起版本说明写得不够全面，无法彰显你本次修改的重大意义……  辣么遇到上边这两种情况，你都避免不了要头疼。但记住一点，只要是在本地发生的事情，大多数情况下我们都是有办法恢复滴。  有鱼油猛地虎躯一震，似乎想起来什么 IMG_256   没错，在 [Git实用教程6：回到过去](http://bbs.fishc.com/thread-71464-1-1.html" \t "https://fishc.com.cn/_blank) 中我们讲到可以用 reset 命令回到过去。  但这有点像我们打游戏打不过 BOSS，退回最近存档重练的赶脚……对于希望“一次通关”的完美主义者来说多少有点不爽！那么有没有一步到位的方法呢？  答案是肯定的，可以执行带 --amend 选项的 commit 命令，Git 会“更正”最近的一次提交。  案例还是前边的 MyProject2（别问我为嘛不整点 niubi 轰轰的栗子，因为那样你就会忙着去看代码而忘了学 Git 了！就像你们去车展，难道是去看车的吗？）   回顾一下，上次经我们这么一折腾，仓库里总共提交了两个版本：  IMG_257   然后为了演示 diff 命令，我们将 README.md 文件改了又改，最后三棵树是酱紫：  IMG_258   这时，我们希望将暂存区域的 README.md 文件进行提交，但又没必要为此增加一个新的版本（你说是吧）。  怎么办呢？执行 git commit --amend 命令修改最后一次提交即可。  执行命令后会立刻跳转到下边界面：  IMG_259   看到没有，老外对我们满满的恶意 IMG_260   其实进入的界面是让你编辑提交说明而已，如果你 commit 的时候没有 -m 选项，都会进入这么一个窗口让你填写说明。  大家也看到了，这一块对中文的支持仍然是一个大写的不友善！  So，如果需要修改提交说明，那么可以执行 git commit --amend -m "新的提交说明" 命令。  如果不需要修改，请按下快捷键 Shift + z + z（其实就是连续两个大写 Z）来退出，或者可以按下冒号（:），然后输入 q! 退出。那么会 Git 会保留旧的提交说明。  IMG_261   此时，Git 仓库中仍然只有两个版本：  IMG_262   暂存区域的内容也提交上去了：  IMG_263   Clean!!!   ****删除文件****  “自从使用了 Git，妈妈再也不用担心我删错文件了！”—— 一个文件只要被 Git 盯上了，那么它就休想轻易地逃出魔掌！  比如我现在从工作目录中手动删除 README.md 文件，然后执行 git status 命令：  IMG_264   Git 马上意识到哪里不对劲，赶紧对比一下暂存区域……噢，原来你小子偷偷地把 README.md 删除了丫！哼，还想瞒天过海？IMG_265   Git 同时提醒使用 checkout 命令可以将暂存区域的文件恢复到工作目录：  IMG_266   看，README.md 又回来了：  IMG_267   那现在问题来了：****如何在 Git 中删除文件呢？****  假如你不小心把小黄图下载到了工作目录，然后又不小心提交到了 Git 仓库：  IMG_268   IMG_269   这下肿么办？让老板看到了这鱿鱼就吃定了……  这时，小甲鱼祭出了 rm 命令救你于水深火热之中！  执行 git rm yellow.jpg 命令：  IMG_270   你会发现此时工作目录中的小黄图（yellow.jpg）已经被删除……  但执行 git status 命令，你仍然发现 Git 还肯松手：  IMG_271   (⊙o⊙) Git 的意思是说它在仓库的快照中发现有个叫 yellow 的东西，但似乎在暂存区域和当前目录不见了！  好吧，事实证明要从 Git 中真正删除一个文件是件非常麻烦的事儿！  此时，你可以执行 git reset --soft HEAD~ 命令将快照回滚到上一个位置，然后重新提交，Git 就不会再提小黄图的事儿了：  IMG_272   ****注意：rm 命令删除的只是工作目录和暂存区域的文件（即取消跟踪，在下次提交时不纳入版本管理）****   你们别嫌我啰嗦，还有两种情况你们可能会遇到，比如我在工作目录中增加一个 test.py 文件，然后执行 git add test.py 命令将其添加到暂存区域，此时我修改 test.py 文件的内容，那么暂存区域和工作目录就是两个不同的 test.py 文件了：  IMG_273   此时如果我执行 git rm test.py 命令，Git 会下意识地阻止我：  IMG_274   因为两个不同内容的同名文件，谁知道你是不是搞清楚了都要删掉？还是提醒一下好，别等一下出错了又要赖机器……  根据提示，执行 git rm -f test.py 命令就可以把两个都删除。  还有另一种情况就是你可能只想删除暂存区域的文件（保留工作目录的），那么你可以执行 git rm --cached 文件名 命令实现目的。   ****重命名文件****  代码改着改着突然觉得文件的名字不合适了，于是重命名，这种情况平时不少见吧？  但你这么做，通常会把 Git 搞懵逼！  我们把 game.py 重命名为 wordgame.py，执行 git status 命令：  IMG_275   Git 说：OMG，仓库的快照中检查到有一个叫 game.py 的文件，你是不是删了（rm）？然后又整了个叫 wordgame.py 的新文件进来？  其实我们就只是改了一下名字……  好吧，先把名字改回来（你再执行 git status 又可以看到 clean 了）。  在 Git 里重命名，需要让 Git 来帮你做，这样它才便于跟踪。  执行 git mv game.py wordgame.py 命令：  IMG_276   木有消息，因为在 Git 看来，这点儿小事不值得一提~~  但你执行 git status 命令可以看到操作记录：  IMG_277    其实，mv 命令相当于执行下边三条命令：   1. ren game.py wordgame.py 2. git rm game.py 3. git add wordgame.py   复制代码  注：Windows 使用 ren 命令修改文件名，Linux 是使用 mv 命令……都说了，深深的文化入侵的恶意……   ****彩蛋****  噢，对了，你应该还不知道如何让 Git 忽略一些指定的文件吧？  我的意思是让 Git 识别某些格式的文件，然后自主不跟踪它们（不要总提示 XXX 还没加入暂存区域，要不要加入啊？要不要跟踪啊？要不要？要不要嘛？！！）  老规矩，回复“朕想知道”，小甲鱼来告诉你！ 本帖隐藏的内容 现在在工作目录中添加 1.temp、2.temp 和 3.temp 三个文件。  我们不希望后缀名为 temp 的文件被追踪，可 Git 却像个烦人的小妖精：  IMG_278   （要不要嘛，要不要嘛，嗯，要嘛，要嘛~~~）  让 Git 闭嘴的办法是有的，需要你在工作目录创建一个名为 .gitignore 的文件。  然后你发现 Windows 压根儿不允许你在文件管理器中创建以点（.）开头的文件（Linux 和 Mac 我们都习惯在终端下工作，这里就不啰嗦）：  IMG_279   Windows 需要在命令行窗口下才能创建以点（.）开头的文件，执行 echo \*.temp > .gitignore 命令，创建一个 .gitignore 文件，并让 Git 忽略所有 .temp 后缀的文件：  IMG_280   再看看 Git 的状态：  IMG_281   好了，Git 已经忽略了所有的 \*.temp 文件（你还可以把 .gitignore 文件也一并忽略）。  ****.gitignore 语法规范****  .gitignore 可以使用标准的 glob 模式匹配（glob 模式是指 shell 所使用的简化了的正则表达式）：   * 所有空行或者以注释符号 # 开头的行都会被 Git 忽略； * 星号（\*）匹配零个或多个任意字符； * [abc] 匹配任何一个列在方括号中的字符； * 问号（?）只匹配一个任意字符； * [a-z] 匹配所有在这两个字符范围内的字符； * 匹配模式最后跟反斜杠（/）说明要忽略的是目录； * 匹配模式以反斜杠（/）开头说明防止递归； * 要忽略指定模式以外的文件或目录，可以在模式前加上惊叹号（!）取反。   其实语法参考小甲鱼的这篇文章即可：[Python3 如何优雅地使用正则表达式](http://bbs.fishc.com/thread-57073-1-1.html" \t "https://fishc.com.cn/_blank)  嗯嗯，这里有一个十分详细的针对数十种项目及语言的 .gitignore 文件列表，拿走，不谢 IMG_282 -> [传送门](https://github.com/github/gitignore" \t "https://fishc.com.cn/_blank) | |

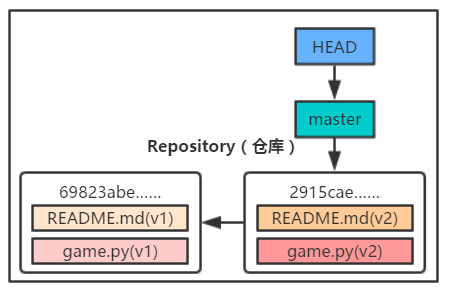
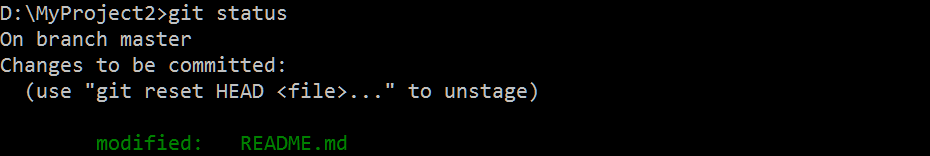
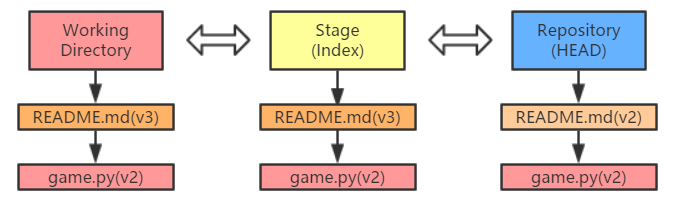
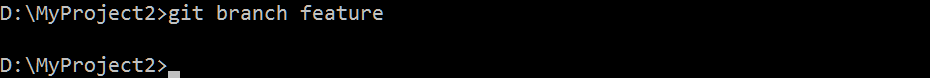
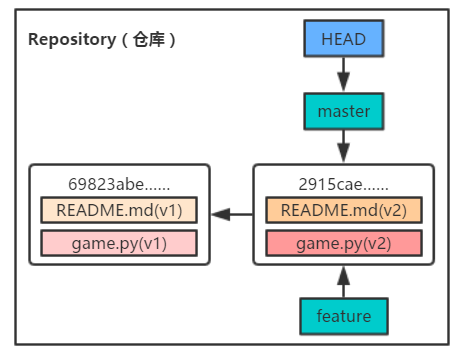
**创建和切换分支**

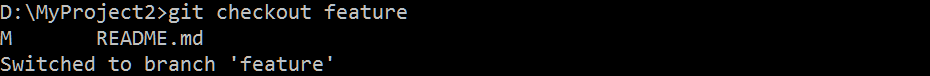
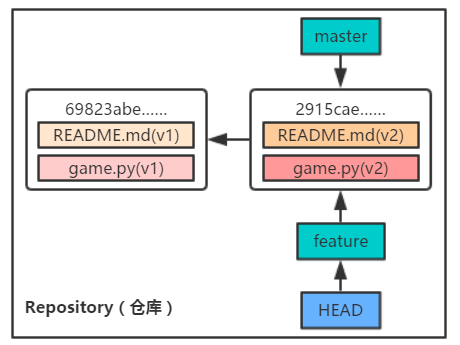
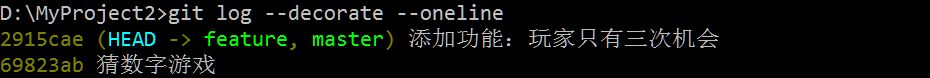
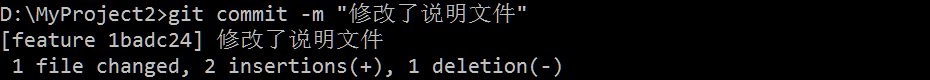
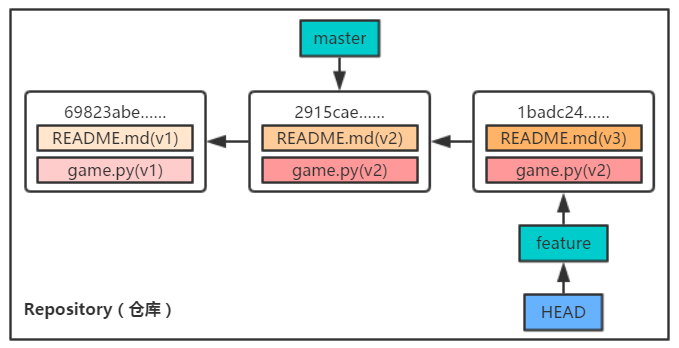
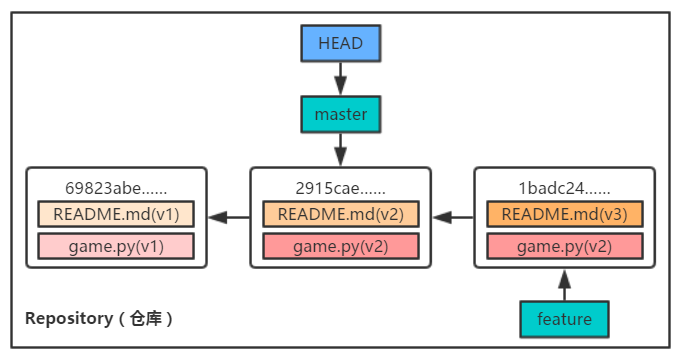
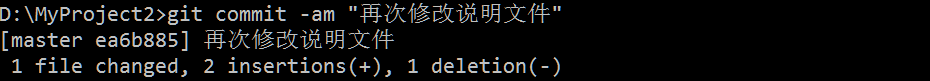
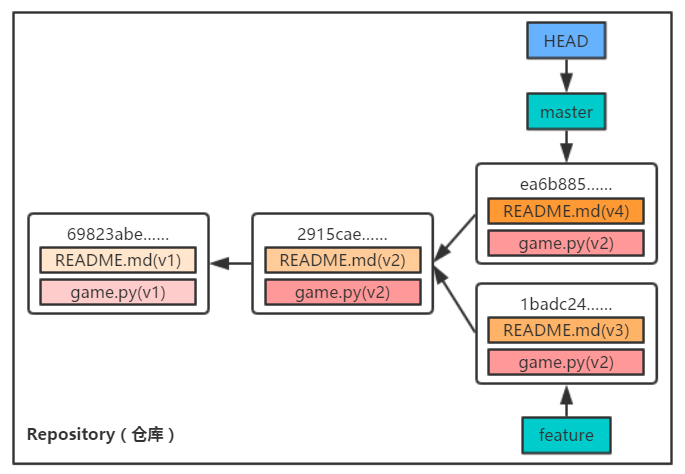
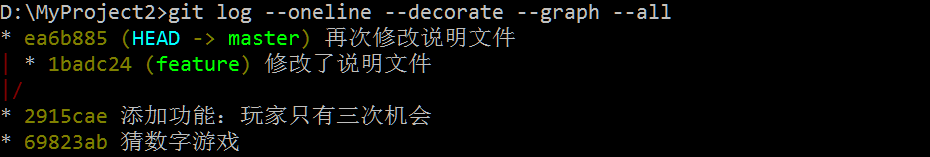
俗话说得好 —— 该来的总是会来的！  
  
之前我们一直把分支挂在嘴边，好多鱼油胃口都给小甲鱼吊到嗓门眼上了，这一节开始小甲鱼就来给大家讲讲 Git 的分支和分支管理！分支无疑是 Git 最值得傲娇的技能，不信你看它的 LOGO IMG_256 ：  
  
   
这不就是个大写的劈叉嘛~  
  
  
****分支是什么？****

假设你的大项目已经上线了（有上百万人在使用呢），过了一段时间你突然觉得应该添加一些新的功能，但是为了保险起见，你肯定不能在当前项目上直接进行开发，这时候你就有另（创）起（建）炉（分）灶（支）的需要了。  
  
放个大图先让你知道分支大概是咋回事：  
  
   
  
如果没有分支，整个产品的迭代周期就会因为新功能的开发而被延误（因为还有补 bug 这样比较着急的任务）……可见分支对于版本控制系统来说是多么重要！  
  
不过话说回来，目前几乎所有的版本控制系统都以某种形式支持分支，那么 Git 还有什么好嘚瑟的呢？  
  
记得在第一讲介绍 Git 的时候跟大家分享过资深玩家对 SVN 的吐槽：

克隆一份全新的目录以同样拥有 5 个分支来说，SVN 是同时复製 5 个版本的文件，也就是说重复 5 次同样的动作。而 Git 只是获取文件的每个版本的元素，然后只载入主要的分支（master）在我的经验，克隆一个拥有将近一万个提交（commit），5 个分支，每个分支有大约 1500 个文件的 SVN，耗了将近 1 小时！而 Git 只用了区区的 1 分钟！

“克隆一个拥有将近一万个提交（commit），5 个分支，每个分支有大约 1500 个文件的 SVN，耗了将近 1 小时！而 Git 只用了区区的 1 分钟！”……突然小甲鱼想起了一句话 —— 天下武功，唯快不破！  
  
我们深入一点思考，到底是什么成就了 Git“秒男”的称号呢？IMG_259   
  
在第三讲（[传送门](http://bbs.fishc.com/thread-71255-1-1.html" \t "https://fishc.com.cn/_blank)）中，我们介绍理论的时候，说 Git 采用一种看似“异端”的形式来处理版本迭代 —— 通常的版本控制系统是采用增量文件系统来管理版本迭代；而 Git 则是采用将每个版本都独立存储的方式 —— 看上去使用 Git 会耗费更多的空间，但来到分支管理这一块，却成了 Git 完胜其它版本控制系统的关键！  
  
因为对于其它版本控制系统而言，创建分支常常需要完全创建一个源代码目录的副本，项目越大，耗费的时间就越多；而 Git 由于每一个结点都已经是一个完整的项目，所以只需要创建多一个“指针”（像 master）指向分支开始的位置即可。  
  
  
****创建分支****

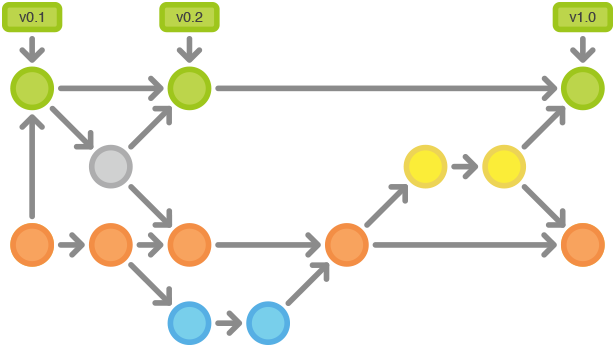
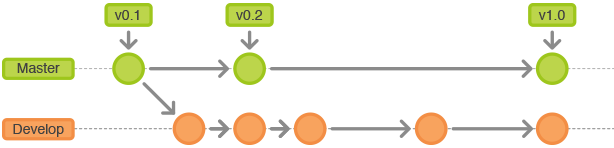
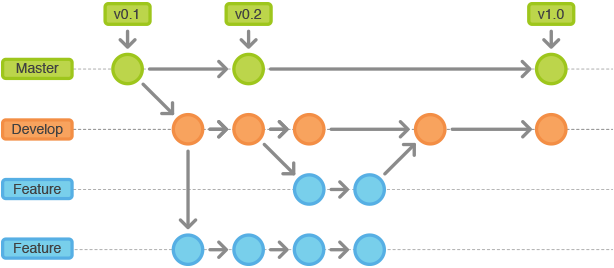
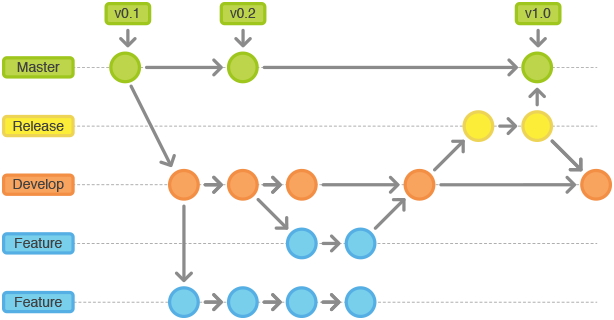
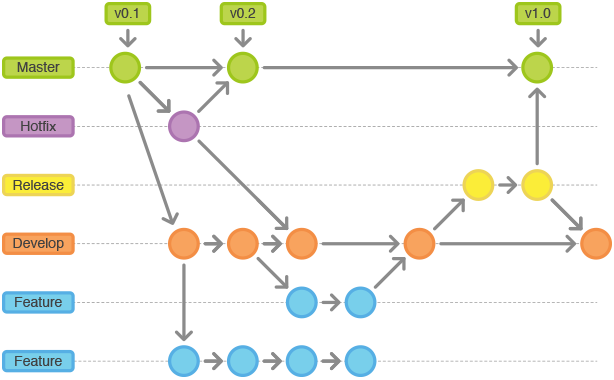
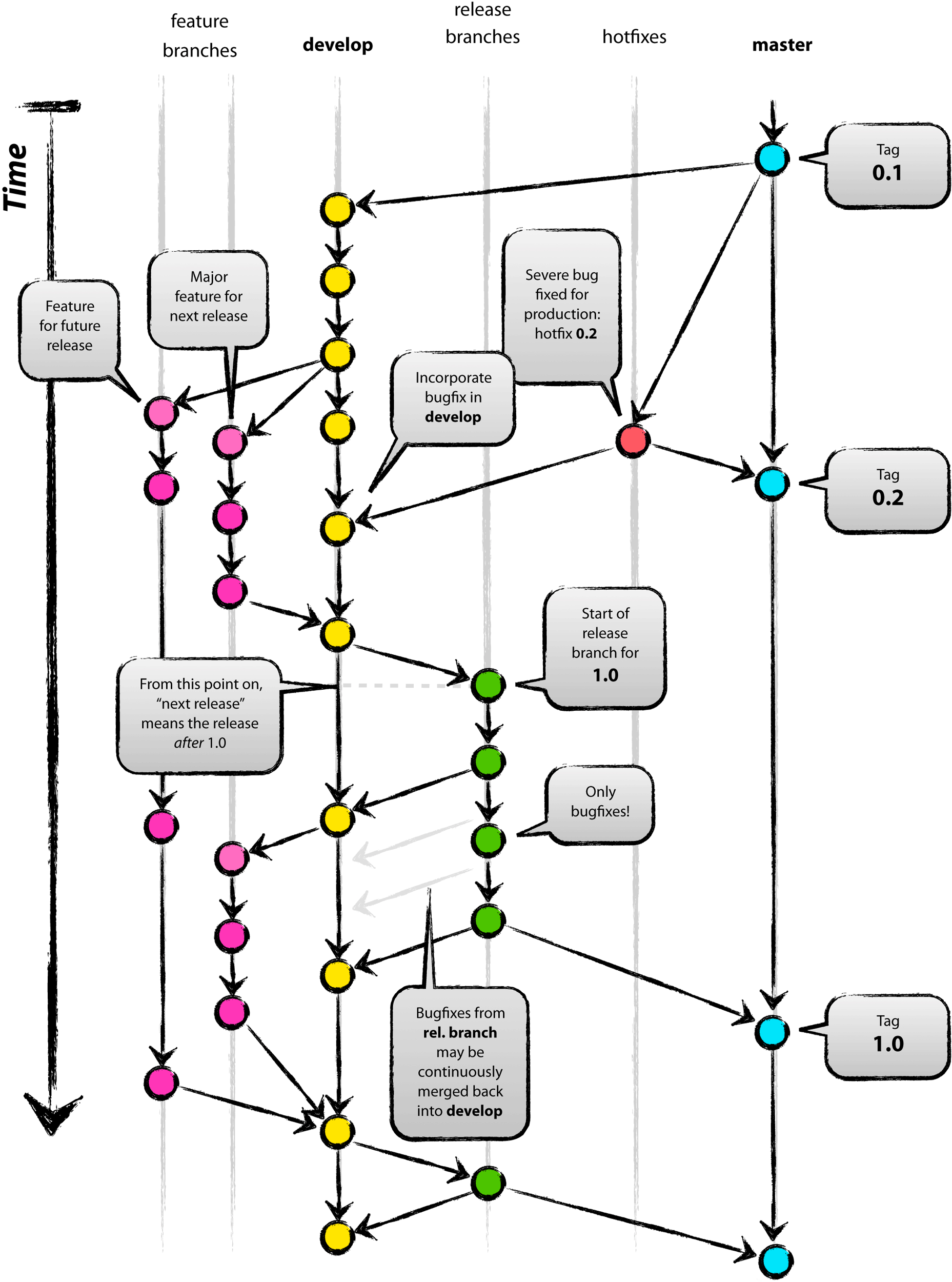
下边小甲鱼给大家演示如何创建和切换分支，我们来到之前的 MyProject2（如果你已经删了，没关系，重新创建一个项目，然后随便写点东西，提交然后修改，再提交，即可）：  
  
   
  
MyProject2 现在仓库里的情况如下：  
  
   
  
执行 git status 命令查看状态：  
  
   
  
可以看到 README.md 文件被修改并添加到暂存区域（还没有提交），所以当前三棵树应该是这样：  
  
   
  
好，现在我们开始创建分支！  
  
创建分支，使用 git branch 分支名 命令：  
  
   
  
没有任何提示说明分支创建成功（一般也不会失败啦，除非创建了同名的分支会提醒你一下），此时可以执行 git log --decorate 命令查看：  
  
*如果希望以“精简版”的方式显示，可以加上一个 --oneline 选项（即 git log --decorate --oneline），这样就只用一行来显示一个快照记录。*  
  
   
  
可以看到最新的快照后边多了一个 (HEAD -> master, feature)  
  
它的意思是：目前有两个分支，一个是主分支（master），一个是刚才我们创建的新分支（feature），然后 HEAD 指针仍然指向默认的 master 分支。  
  
所以目前仓库中的快照应该是这样：  
  
   
  
  
****切换分支****

现在我们需要将工作环境切换到新创建的分支（feature）上，使用的就是之前我们欲言又止的 checkout 命令。  
  
执行 git checkout feature 命令：  
  
   
  
这样 HEAD 指针就指向 feature 分支了：  
  
   
  
什么？！口说无凭！IMG_269   
  
好吧，你们要的证据在这里：  
  
   
  
看到没有？HEAD 指针已经指向了 feature 分支：（HEAD -> feature）  
  
现在我们进行一次提交（暂存区域还有一个更改的文件没有提交呢）：  
  
   
  
现在仓库中的快照应该是酱紫：  
  
   
  
然后我们将 HEAD 指针切回 master 分支：  
  
IMG_273   
  
细心的朋友会发现上一次对 README.md 文件的修改已经荡然无存了，这是因为我们的工作目录已经回到 master 分支的状态中：  
  
   
  
现在对 README.md 文件进行修改（随便改改），然后执行 git commit -m "再次修改说明文件"：  
  
   
  
好了，目前仓库中的快照应该变成了酱紫：  
  
   
  
什么？！不相信？IMG_277   
  
好吧，让 Git 自己来告诉你！  
  
执行 git log --oneline --decorate --graph --all 命令：  
  
*--graph 选项表示让 Git 绘制分支图，--all 表示显示所有分支*  
  
   
  
  
****彩蛋****

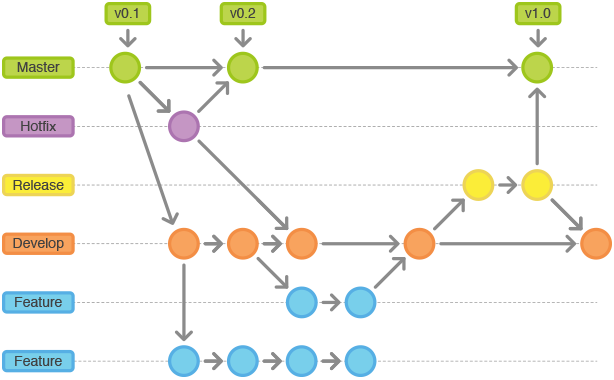
想不想知道 Git 在实际开发中是如何进行分支管理的呢？  
  
回复“朕想知道”，小甲鱼来告诉你！

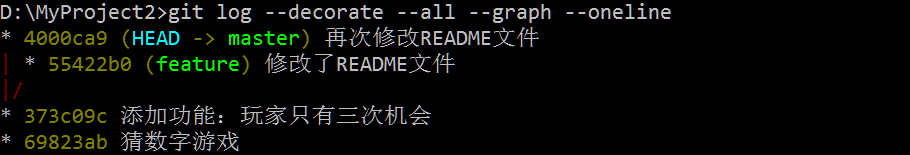
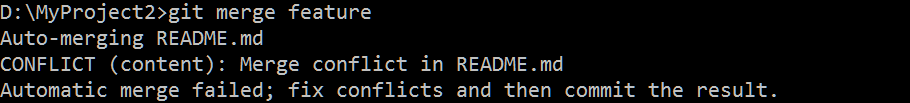
#### 本帖隐藏的内容

****Git 工作流程****

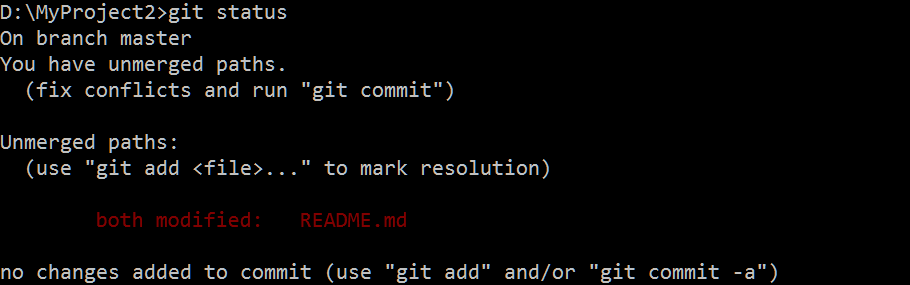
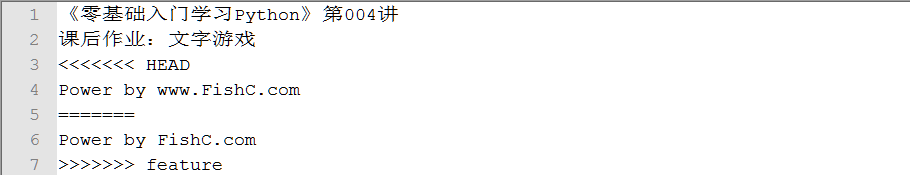
   
  
Git 工作流使用一个中间仓库作为所有开发者的交流地点，开发程序的小伙伴在本地工作，然后将各自的分支推送到中间仓库。  
  
  
****开发分支（develop）****  
  
   
  
代替单一的 master 主分支，上图的工作流使用了两个分支来处理项目发布和日常开发。  
  
master 主分支通常只是用于对外发布项目的新版本，日常开发应该在另一条分支上完成。我们把开发用的分支叫做 develop 分支。  
  
  
****功能分支（feature）****  
  
   
  
每一个新功能应该使用单独一个功能分支进行开发，功能分支应该从开发分支中分离出来，功能开发完成后合并到开发分支。  
  
提示1：功能分支不应该跟 master 分支有任何交流。  
  
提示2：功能分支可以采用 feature-\* 的形式命名。  
  
  
****预发布分支（release）****  
  
   
  
在项目正式发布之前，你可能需要一个预发布的版本进行测试。于是你可以从开发分支中分离出预发布分支，用于内部或公开的测试。  
  
提示1：预发布分支应该同时合并到主分支和开发分支中。  
  
提示2：预发布分支可以采用 release-\* 的形式命名。  
  
  
****维护分支（hotfix）****  
  
   
  
项目正式发布后难免会出现 bug，这时就需要创建一个分支，进行 bug 的修补。  
  
提示1：维护分支应该从主分支中分离出来，bug 被修补后，再合并到主分支和开发分支中。  
  
提示2：维护分支可以采用 fixbug-\* 的形式命名。  
  
  
****常设分支****  
  
常设分支就主分支（master）和开发分支（develop）两个即可，另外的功能分支（feature）、预发布分支（release）和维护分支（hotfix）属于临时分支，用完之后应该及时删除。  
  
  
So，在正式开发中，Git 的分支管理如下：

**合并和删除分支**

话说天下大势，分久必合，合久必分。  
  
我们知道，在 Git 仓库里，所有的快照事实上就是按照提交时间排放的。我们把串联这些快照的时间轴称之为分支，默认情况下 Git 只有一条 master 主分支。但是在现实开发中，从来不存在一条分支定乾坤的事情。  
  
那么实际开发中的分支应该是长这样的：  
  
   
  
注：这些分支的命名方法以及具体的创建逻辑，大家可以看一下上一篇 —— [Git实用教程9：创建和切换分支（有彩蛋）](http://bbs.fishc.com/thread-73806-1-1.html" \t "https://fishc.com.cn/_blank)—— 的彩蛋部分。  
  
从上图中我们看出，一个项目根据其需求会产生很多分支，但最终都会被合并回去。  
  
这一讲，我们就来谈谈分支的合并。  
  
  
****合并分支****

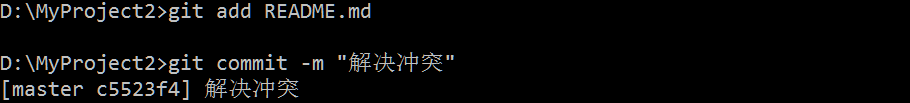
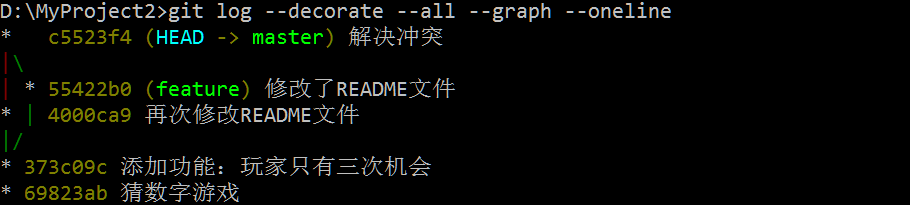
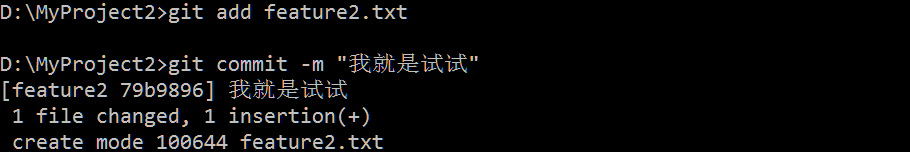
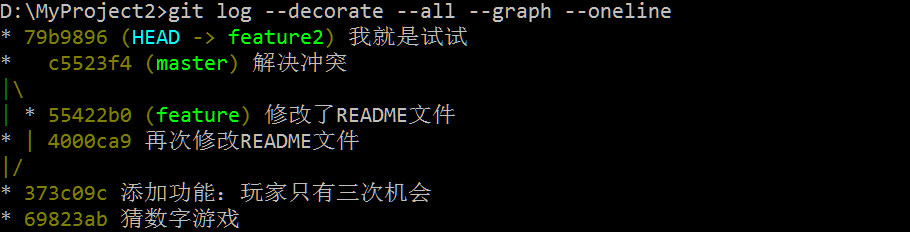
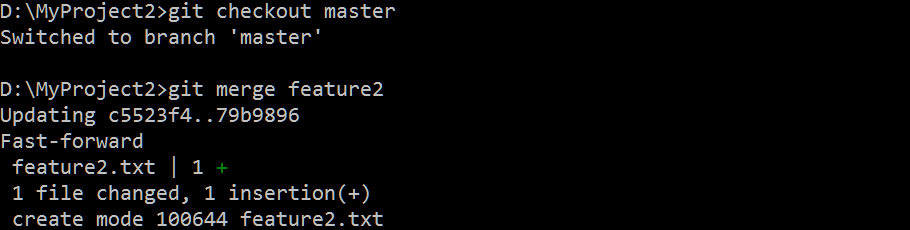
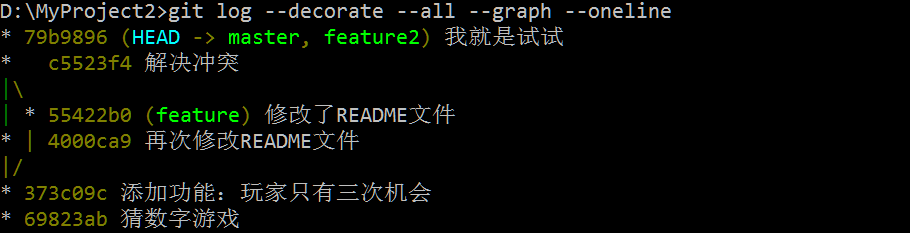
当一个子分支的使命完结之后，它就应该回归到主分支中去。  
  
上次经我们这么一折腾之后，分支的状态如下：  
  
   
  
合并分支我们使用 merge 命令，执行 git merge feature 命令，将 feature 分支合并到 HEAD 所在的分支（master）上：  
  
   
  
Holy crap!! IMG_259 这就叫出师不利……  
  
从 Git 提示的内容来看，我们知道这次的合并并没有成功，Git 说：

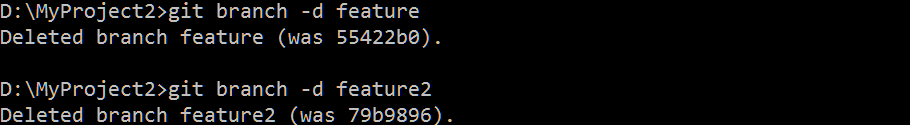
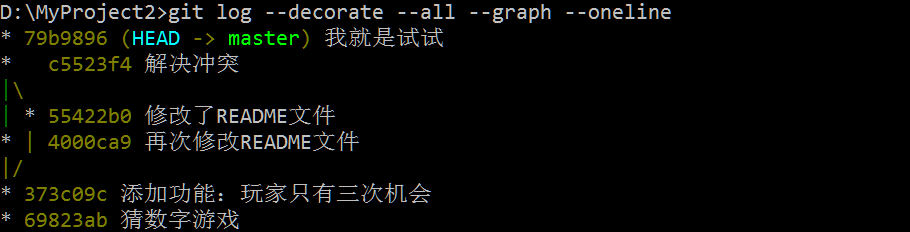
合并 README.md 文件的时候出现冲突。  
  
所以自动合并失败；请修改冲突的内容并重新提交快照。

意思是说现在你需要先解决冲突的问题，Git 才能进行合并操作。所谓冲突，无非就是像两个分支中存在同名但内容却不同的文件，Git 不知道你要舍弃哪一个或保留哪一个，所以需要你自己来决定。  
  
此时执行 git status 命令也会显示需要你解决的冲突：  
  
   
  
然后 Git 会在有冲突的文件中加入一些标记，不信你打开 README.md 文件看看：  
  
   
  
以“=======”为界，上到“<<<<<<< HEAD”的内容表示当前分支，下到“>>>>>>> feature”表示待合并的 feature 分支，之间的内容就是冲突的地方。  
  
现在我们将 README.md 统一修改如下（同时去掉了 <<<<<<< HEAD 等内容）：

1. 《零基础入门学习Python》第004讲
2. 课后作业：文字游戏
3. Power by www.FishC.com

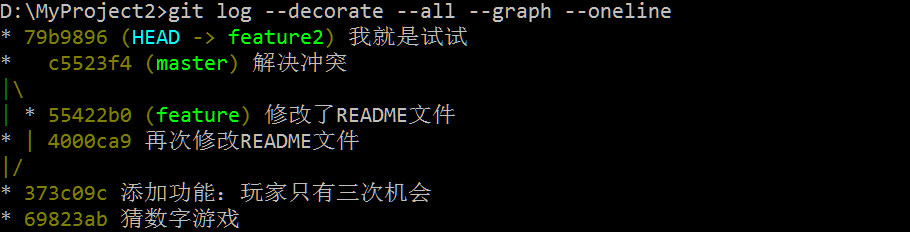
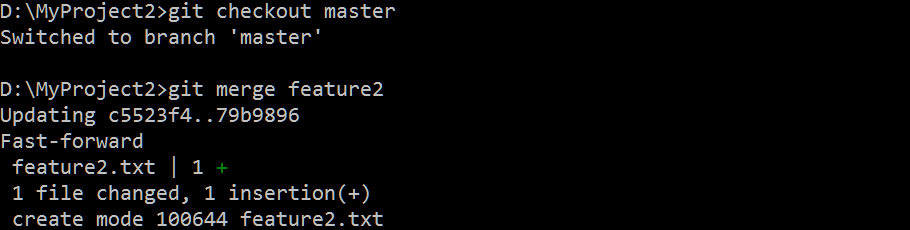
复制代码

保存文件，然后提交快照：  
  
   
  
执行 git log --decorate --all --graph --oneline 命令，可以看到此时的分支已经自动合并了：  
  
   
  
当然，如果不存在冲突，就不用搞这么多了……  
  
我再做一次给你看！  
  
执行 git checkout -b feature2 命令（相当于 git branch feature2 和 git checkout feature2 两个命令的合体）：  
  
IMG_264   
  
在工作目录随便创建一个文本文件（feature2.txt）并提交快照：  
  
   
  
执行 git log --decorate --all --graph --oneline 命令：  
  
   
  
可以看到，feature2 分支比 master 分支快了一步。现在我们切换回 master 分支，并将 feature2 分支合并进来：  
  
   
  
这次 Git 只显示了 Fast-forward（快进）这词儿，这是因为 feature2 这个分支的父结点是 master 分支，所以 Git 只需要简单移动 master 的指向即可。  
  
执行 git log --decorate --all --graph --oneline 命令：  
  
   
  
  
****删除分支****

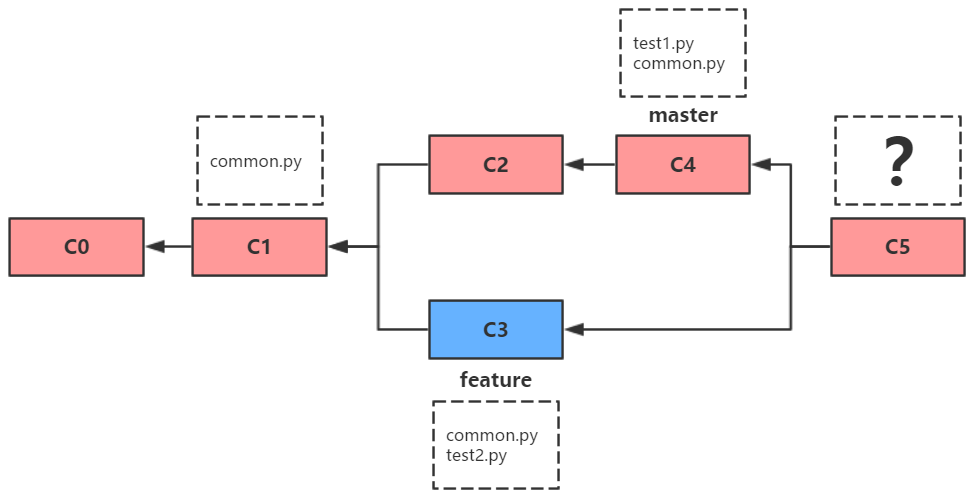
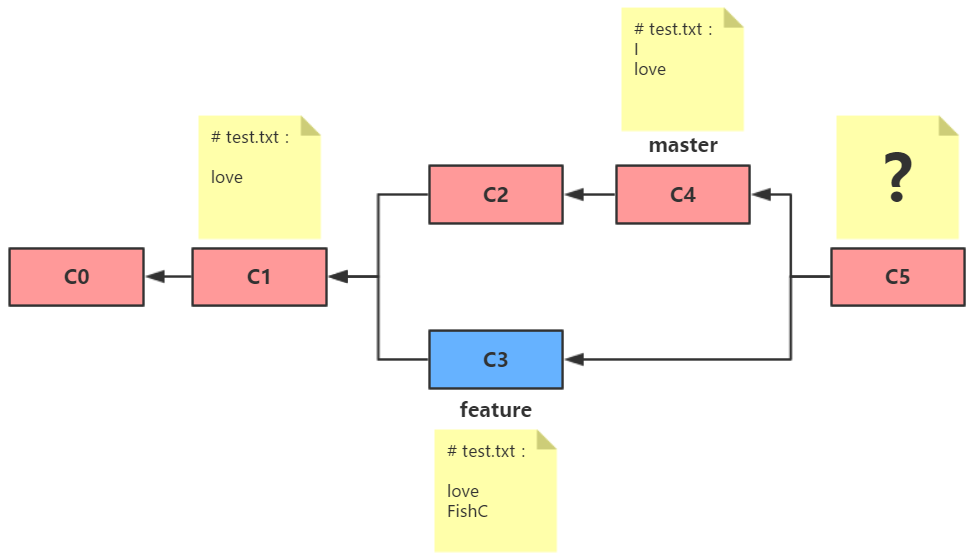
对于不再需要的分支，我们还是把它们删了吧！  
  
删除分支，使用 git branch -d 分支名 命令：  
  
   
  
执行 git log --decorate --all --graph --oneline 命令：  
  
   
  
这时 feature 和 feature2 已经消失了……  
  
由于 Git 的分支原理实际上只是通过一个指针记载，所以创建和删除分支都几乎是瞬间完成。  
  
*注意：如果试图删除未合并的分支，Git 会提示你“该分支未完全合并，如果你确定要删除，请使用 git branch -D 分支名 命令。*  
  
  
****彩蛋****

对于分支的合并，Git 又是领先业界好几年！IMG_271   
  
Git 使用自动三方合并的方式，就算在一段较长的时间内，反复把一个分支合并入另一个分支，也不是什么难事。因为通常情况下，Git 会自动解决一些它能够理解的“冲突”，不需要你来干预；但如果是 SVN 之流则需要你亲自处理每一次合并的冲突。  
  
那你知道什么是三方合并（three-way merge）吗？  
  
回复“朕想知道”，小甲鱼就告诉你！IMG_272

#### 本帖隐藏的内容

上面的例子我们演示了 Git 的两种合并方式：Fast-forward 和 Three-way merge。  
  
  
****Fast-forward****  
  
所谓的 Fast-forward 就是当待合并的分支位于目标分支的直接上游时，Git 只需把目标分支的指针直接移动即可实现合并。  
  
比如下面图片中 master 分支是位于 feature2 分支的直接上游：  
  
   
  
将 feature2 分支合并到 master 分支，只需要移动 master 的指针即可：  
  
   
  
  
****Three-way merge****  
  
如果待合并的两个分支不在同一条线上，那么进行合并就需要解决一个根本的问题 —— 冲突！

我想有鱼油可能就会问了：****为何两个分支在同一条线上就不会冲突呢？****  
  
因为 Git 的快照是按时间顺序提交的，所以在同一条线上的两个快照，它们是有先后顺序的，尽管两者可能出现同名文件不同内容，Git 会认为这是“改变”而不是“冲突”。

对于冲突的处理，Git 相对来说是比较机智的。  
  
举个例子：  
  
   
  
合并 C3 和 C4 得到 C5，但 C5 应该如何处理“冲突”呢？  
  
SVN 会把问题抛给用户，让用户自行解决；Git 则显得更为高明，它会找到第三个快照，然后综合三者特点自动解决冲突。  
  
那第三个快照应该如何决定呢？  
  
没错，应该找两者的共同“祖先”作为参照物，一比较就知道两个分支都干了些什么。  
  
图片中 C3 和 C4 的共同祖先是 C1，可以看到 C3 和 C4 分别增加了 test2.py 和 test1.py 两个文件。因为对比之后发现三者并没有冲突，所以 C5 应该是三者的合体，即同时拥有 common.py、test1.py 和 test2.py 三个文件（有兴趣的鱼油不妨测试一下）。  
  
  
另外，值得一提的是，Git 这种合并方式也适用于同名文件的不同更改。  
  
举个例子：  
  
   
  
这里 C3 和 C4 都只有一个文件（test.txt），但是内容却不一样。如果这样合并，你猜 Git 会不会报“冲突”？  
  
答案是不会的！  
  
因为 Git 找到它们的共同祖先 C1，可以看到 C3 和 C4 都是在 C1 的基础上进行添加（C4 在第 2 行添加了“I”，C3 在第 4 行增加了“FishC”，C1 第 3 行的“love”是它们共同拥有的），同时在每一行并没有产生冲突的地方，所以自动合并后的 C5 是这样的：

1. # test.txt
2. I
3. love
4. FishC

复制代码

****注意：如果 Git 检测到同一行有不同的内容，还是会报冲突并让你自行决定谁去谁留的****