**[Git回滚的常用手法](http://hittyt.iteye.com/blog/1961386)**

**博客分类：**

* [编程工具](http://hittyt.iteye.com/category/144381)
* [杂记](http://hittyt.iteye.com/category/141453)

**传统VCS的回滚操作**

对于版本控制系统VCS来说，回滚这个操作应该是个很普通也是很重要的需求。

如果你是传统VCS，比如SVN或者P4来说，revert是个最直观，也是最直接的手段，当然前提是你的修改还没有被提交到远程的中央仓库。

如果你已经ci了你的code到了远程中央仓库，那revert恐怕也无能为力，只能借助其他命令workaroud这个问题，比如：你用SVN的话，就得来个逆向merge操作，把所有的修改都merge回去。

但这样做也有一些弊端：

这次merge会作为一次全新的commit记录记录下来，也就是说它不能真正从你的历史记录里面抹掉你那次不想要的修改。通常情况下其实也没啥大不了的，除非你个人洁癖就是不想看到以前的那次commit记录或者你真的干了啥不想让别人知道的事情。

**Git时代的回滚操作**

但当发展到git时代，这种回滚操作的复杂度，已经随着git模型本身的特点，变得不那么简单了。

熟悉git的人都知道，为了分布式的需求，git将每一个网络节点作为了一个完整的VCS，也就是每个单台的host在没有网络的前提下，都是一个不受任何影响可以满足除了和其他节点同步（比如：git pull/push这类）之外的几乎所有操作。

为了达到这种效果，git不仅在本地有一个完整的local repository，而且将原本简单的working tree（或者叫working directory）也切成了两块区域——working tree和index（也叫stage）。

这样，光从本地修改的角度来看，你的修改就可能存在三块区域中，working tree、index或者commit之后的历史对象区域。下面我们一个一个各个区域一般都怎么回滚。

working tree内的回滚

这个属于最简单一种情形，本质上说也是和传统VCS中revert直接对应的一种场景，只是这里不叫revert了，而是git checkout，这种情形很简单，这里就不做截图展示了。列出依稀常用的命令形式如下：

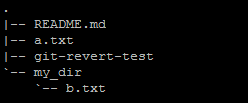
* git checkout file1 （回滚单个文件）
* git checkout file1 file2 ... fileN （一次回滚多个文件，中间用空格隔开即可）
* git checkout . （直接回滚当前目录一下的所有working tree内的修改，会递归扫描当前目录下的所有子目录）

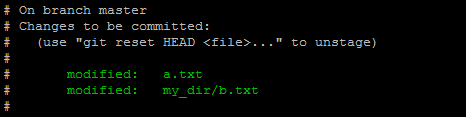
index内的回滚

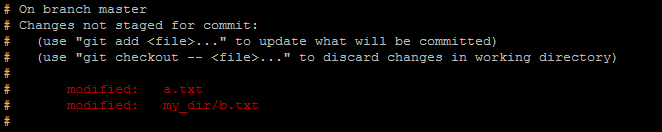
这部分回滚也不复杂，因为这部分的回滚，只要你勤快点使用git status命令，命令的输出上都会给你提示你需要干啥。只是这个过程一般被分为了两步：

1. 将index区域中修改过的文件移除index，也就是恢复到working tree中。这部用git reset来解决。
2. 一旦文件重新回到working tree中，回滚操作就是上面提到的git checkout喽。

这个看个截图直观点：

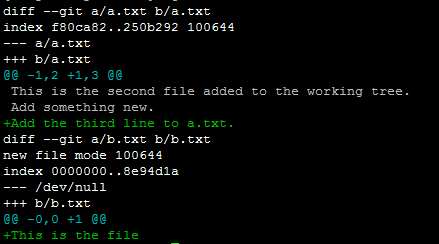
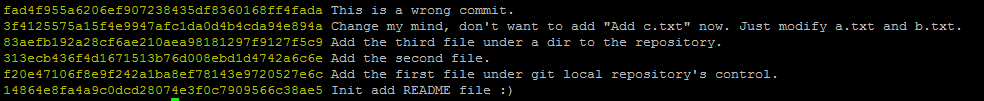
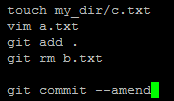
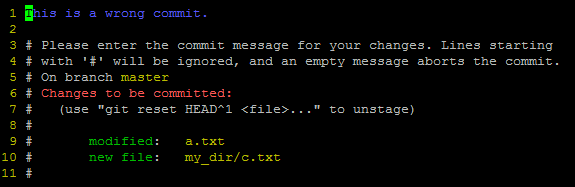
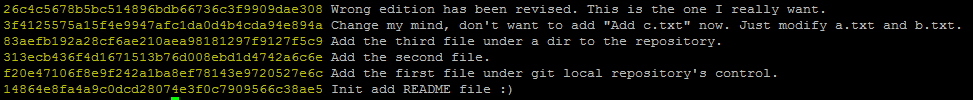
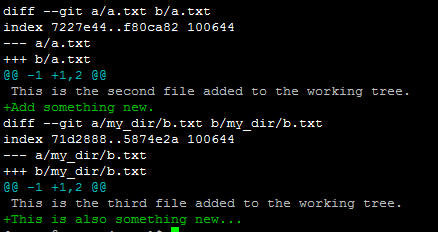
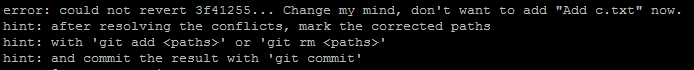
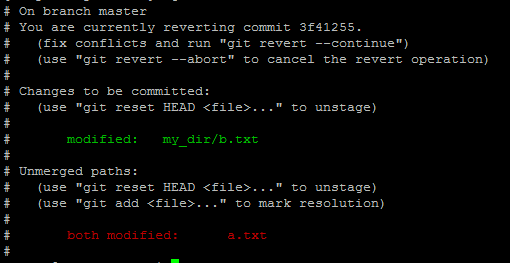
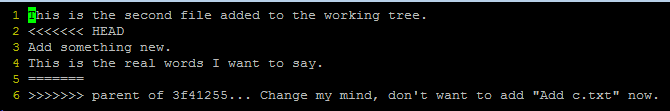
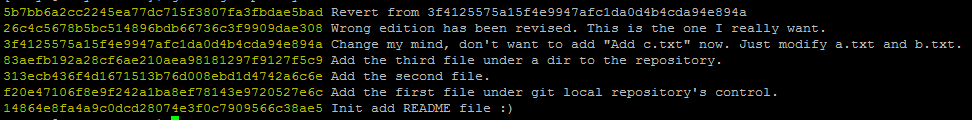
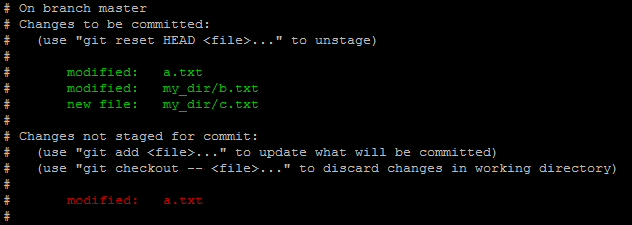
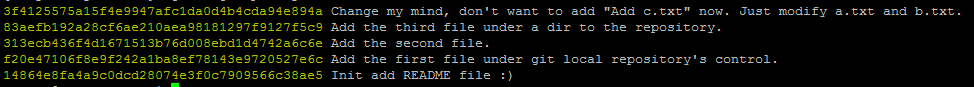
我working tree下的原始文件信息如下  
  
 我修改了a.txt和my\_dir/b.txt，并将将他们加入了index区域，当前运行git status得到如下输出

  
 这里再执行git reset . 将当前目录及子目录内的所有修改移出index区域，再次运行git status命令

  
 到这一步之后，就用上面提到git checkout就可以解决问题了。

commit之后的回滚

这种情形是git本地回滚里面最复杂，也是最容易让人迷糊的了，因为针对不同的情况，方法比较多，所以不是很好记。

* 修改最后一次commit的记录：很多时候先要回滚仅仅是因为自己对最后一 次的commit的漏掉（注意，这里说的漏掉不仅仅是你少提交了文件的修改，也包括你多提交了一下你不想要提交的东西）了一些东西，想要回滚这次 commit之后再重新commit。如果是这样的话，没有必要真的非要先回滚再重新commit。只要在在自己已经满意了自己所有的修改之后，直接执行 git commit --amend，就可以开启上次提交的“补救”提交模式，然后把你对上次所有漏掉的东西加上去就好了。下面看个例子：我进行了一次错误的提交，修改的内容 如下：  
    
   目前commit 记录如下：  
    
   现在我想补救这次commit，相当于取消这次新加入的文件b.txt、取消对a.txt第三行的修改，然后加入我真正想要的修改：在my\_dir下增加一个c.txt，并且修改a.txt的第三行为另外一句话。  
    
     
    
   再次通过git log查看commit记录  
    
   请 注意比较最新的一次commit的修改，其实已经被修改为另一个SHA1的值了。这里请注意，从某种意义上说（实际上这种替换在reflog中很容易追踪 到痕迹，只是在所有的commit逆向引用链条中，我们已经找不到之前的那个fad4...），这种操作已经做到了无痕修改最后一次提交。这和SVN的逆 向merge是本质不同的。
* 回滚中间的某次提交（当然也包括最后一次）：比如我想要回滚上图中倒数第二次提交，就是HEAD^那次，我们先通过git show HEAD^看看那次提交都干了啥？  
    
   然后再通过git revert HEAD^ 来回滚这次操作，然后我们得到了下面的提示：  
    
   杯具，冲突了。。。其实，只要你熟悉任何一种VCS工具，想想这个场景，其实也是挺正常的。那就git status看看哪些个文件冲突了吧。  
    
   其实你只要仔细看看上面的说明信息，应该已经知道该怎么解决这个冲突了。明显，a.txt是冲突发生的文件：  
    
   打 开这个文件，可以看到标准的冲突标识文件。这里正是之前我们采用补救式提交方式修改的那句话。至于冲突怎么解决很容易，看你究竟想要啥了，自己去编辑，去 掉冲突范围标识符号，保存文件即可。然后按照git正常的流程再次提交，编辑提交的信息即可。再次提交之后的log信息如下：  
    
   上 面我们基本上演示了一个标准的revert场景（包括了冲突解决），从这个过程可以看出，git revert和SVN的逆向merge几乎如出一辙，就是将你需要回滚的那次commit所做的所有操作，反向操作一次，然后重新做一一个单独的 commit对象进行提交。这个过程是否发生冲突，就取决于你的修改了。请注意，这个过程你虽然回滚了你不想要的修改内容，但是你没法抹掉那次 commit在history中的信息，请注意上图的第三行，他依旧坚挺的躺在那里。这个也是git revert的特点。当然，git revert实际上也提供-n（--no-commit）参数，用来表示仅将revert的修改体现在当前的working tree，不自动进行提交。但是如果你真的想回滚那些修改的话，再次commit这个环节是逃不掉的。
* 回滚最后的N次提交（永远从commit的history中抹掉这些记录）： 这种场景就轮到git reset登场了。git reset的帮助文档写的非常清楚，在回滚commit的场景中，他的作用就是将当前的HEAD reset到你指定的那个分支。但这个过程中最值得注意的就是你使用的参数，最常用的主要是--soft（个人推荐使用这个，他不会修改你目前index 或者working tree中所做的任何修改）/--mixed（你在reset时不加任何参数时的默认行为，会默默把你在index中的修改给灭了！）/--hard（这 个是我绝的最危险的参数，会把你index和working tree中的所有修改毁灭的毛都不剩，使用之前请三思，这确实是你要的行为！）这三种。因为我推荐使用--soft参数，下面主要演示回滚到 3f412...那次的记录（git reset --soft HEAD~2）：  
    
   从上面可以看出来，你的index区域忽然多了很多未提交的修改，这些就是回滚回来的记录，要怎么处理他们，就看你的了。这时我再来看看log的记录信息：  
    
   最 新的提交已经变成我们希望的那次了。其实从git reset的解释中，我们就可以看出，git reset是一个“斩断”式的回滚操作，因为你把当前的HEAD指针直接移动到了你需要回滚到的那次记录。而git本身的commit链条是逆向回溯的， 所以你在提交历史里面再也找不到当前HEAD指向的commit之后的记录了。（不过如果你是git文艺青年的话，你当然知道，想找到那些表面上找不到的 commit，通过reflog也是易如反掌）。

好了，到这里，常用的git回滚操作和场景都介绍完了。希望对不熟悉git的TX能有所帮助。