GIT基本概念和用法总结   
guibin.beijing@gmail.com   
  
在日常使用GIT过程中，经常会出错，比如无意间丢失了未提交的数据，回退版本时丢失了工作目录，等等。经过思考发现，所有这些错误都是因为对GIT中一些基本的概念模糊而导致，因为对一些基本概念不清晰，导致对GIT每一条命令将会产生的结果不符合预期。下面我就梳理以下我经常碰到的问题相关的基本概念。   
  
1. Working Directory（工作目录）   
Git的工作目录是保存当前正在工作的文件所在的目录，和working tree是相同的意思。在这个目录中的文件可能会在切换branch时被GIT删除或者替换。这个目录是个临时目录，临时存储你从GIT库中取出的文件，这些文件一直会被保存，直到下次提交。   
  
2. GIT Directory（GIT库目录）   
项目的所有历史提交都被保存在了GIT库目录中，只要你不作回滚操作，它应该不会丢失。   
  
3. GIT Index（Git索引）   
Git index 可以看作是工作目录和Git库目录之间的暂存区，和staging area是相同的意思。可以使用Git index构建一组你准备一起提交的改变。Git Index和Git Staging area是同一个意思，都是指已经被add的但尚未commit的那些内容所在的区域。最简单的查看目前什么内容在index中的方法是使用**git status**命令。 

* 命令中”Changes to be committed“中所列的内容是在Index中的内容，commit之后进入Git Directory。
* 命令中“Changed but not updated”中所列的内容是在Working Directory中的内容，add之后将进入Index。
* 命令中“Untracked files”中所列的内容是尚未被Git跟踪的内容，add之后进入Index。

哪些操作能够改变git index中的内容？   
A). **git add <path>...**会将working directory中的内容添加进入git index。   
B). **git reset HEAD <path>...**会将git index中path内容删除，重新放回working directory中。   
  
4. git diff   
git diff可以比较working tree同index之间，index和git directory之间，working tree和git directory之间，git directory中不同commit之间的差异， 

* **git diff [<path>...]**：这个命令最常用，在每次add进入index前会运行这个命令，查看即将add进入index时所做的内容修改，即working directory和index的差异。
* **git diff --cached [<path>...]**：这个命令初学者不太常用，却非常有用，它表示查看已经add进入index但是尚未commit的内容同最后一次commit时的内容的差异。即index和git directory的差异。
* **git diff --cached [<commit>] [<path>...]**：这个命令初学者用的更少，也非常有用，它表示查看已经add进入index但是尚未commit的内容同指定的<commit>之间的差异，和上面一条很相似，差别仅仅<commit>，即index和git directory中指定版本的差异。
* **git diff <commit> [<path>...]**：这个命令用来查看工作目录和指定<commit>的commit之间的差别，如果要和Git directory中最新版比较差别，则<commit>=HEAD。如果要和某一个branch比较差别，<commit>=分支名字
* **git diff <commit> <commit> [<path>...]**：这个命令用来比较git directory中任意两个<commit>之间的差别，如果想比较任意一个<commit>和最新版的差别，把其中一个<commit>换成HEAD即可。

5. 如何merge不同的分支   
在git中，在执行任何命令时你一定要清楚，你在哪？对谁执行这个命令？   
比如在创建新的branch时，执行命令：**git branch 1.0-beta**，这个命令是说在当前branch上，以当前branch为基准，创建一个新的branch，名叫1.0-beta。   
在比如，当merge不同的branch时：

**引用**

git checkout 1.0-beta   
git merge master

首先切换到1.0-beta branch上，然后将主干（master）上的代码合并到当前1.0-beta分支上。   
merge完后，可能会由冲突，按照git的提示，编辑标识为"CONFLICT (content)"的文件，解决冲突后再次将冲突的文件add，commit后，merge完毕。   
  
6. git reset 

* 在一般使用中，如果发现错误的将不想staging的文件add进入index之后，想回退取消，则可以使用命令：**git reset HEAD <file>...**，同时git add完毕之后，git也会做相应的提示，比如：

**引用**

# Changes to be committed:   
#   (**use "git reset HEAD <file>..." to unstage**)   
#   
# new file:   Test.scala

* **git reset [--hard|soft|mixed|merge|keep] [<commit>或HEAD]**：将当前的分支重设（reset）到指定的<commit>或者HEAD（默认，如果不显示指定commit，默认是HEAD，即最新的一次提交），并且根据[mode]有可能更新index和working directory。mode的取值可以是hard、soft、mixed、merged、keep。下面来详细说明每种模式的意义和效果。

A). **--hard**：重设（reset） index和working directory，自从<commit>以来在working directory中的任何改变都被丢弃，并把HEAD指向<commit>。   
B). **--soft**：index和working directory中的内容不作任何改变，仅仅把HEAD指向<commit>。这个模式的效果是，执行完毕后，自从<commit>以来的所有改变都会显示在git status的**"Changes to be committed"**中。   
C). **--mixed**：仅reset index，但是不reset working directory。这个模式是默认模式，即当不显示告知git reset模式时，会使用mixed模式。这个模式的效果是，working directory中文件的修改都会被保留，不会丢弃，但是也不会被标记成"Changes to be committed"，但是会打出什么还未被更新的报告。报告如下：

**引用**

Unstaged changes after reset:   
M Test.Scala   
M test.txt

D). --merge和--keep用的不多，在下面的例子中说明。

下面列出一些git reset的典型的应用场景：   
A) 回滚add操纵

**引用**

$ edit                                     (1)   
$ git add frotz.c filfre.c   
$ mailx                                    (2)   
$ **git reset**                                (3)   
$ git pull git://info.example.com/ nitfol  (4)

(1) 编辑文件frotz.c, filfre.c，做了些更改，并把更改添加到了index   
(2) 查看邮件，发现某人要你pull，有一些改变需要你merge下来   
(3) 然而，你已经把index搞乱了，因为index同HEAD commit不匹配了，但是你知道，即将pull的东西不会影响已经修改的frotz.c和filfre.c，因此你可以revert这两个文件的改变。revert后，那些改变应该依旧在working directory中，因此执行**git reset**。   
(4) 然后，执行了pull之后，自动merge，frotz.c和filfre.c这些改变依然在working directory中。   
  
B) 回滚最近一次commit

**引用**

$ git commit ...   
$ **git reset --soft HEAD^**      (1)   
$ edit                        (2)   
$ git commit -a -c ORIG\_HEAD  (3)

(1) 当提交了之后，你又发现代码没有提交完整，或者你想重新编辑一下提交的comment，执行**git reset --soft HEAD^**，让working tree还跟reset之前一样，不作任何改变。   
**HEAD^**指向HEAD之前最近的一次commit。   
(2) 对working tree下的文件做修改   
(3) 然后使用reset之前那次commit的注释、作者、日期等信息重新提交。注意，当执行git reset命令时，git会把老的HEAD拷贝到文件.git/ORIG\_HEAD中，在命令中可以使用ORIG\_HEAD引用这个commit。commit 命令中 **-a** 参数的意思是告诉git，自动把所有修改的和删除的文件都放进stage area，未被git跟踪的新建的文件不受影响。commit命令中**-c <commit>** 或者 **-C <commit>**意思是拿已经提交的commit对象中的信息（作者，提交者，注释，时间戳等）提交，那么这条commit命令的意思就非常清晰了，把所有更改的文件加入stage area，并使用上次的提交信息重新提交。   
  
C) 回滚最近几次commit，并把这几次commit放到叫做topic的branch上去。

**引用**

$ git branch topic/wip     (1)   
$ **git reset --hard HEAD~3**  (2)   
$ git checkout topic/wip   (3)

(1) 你已经提交了一些commit，但是此时发现这些commit还不够成熟，不能进入master分支，但你希望在新的branch上润色这些commit改动。因此执行了**git branch**命令在当前的HEAD上建立了新的叫做 topic/wip的分支。   
(2) 然后回滚master branch上的最近三次提交。**HEAD~3**指向当前HEAD-3个commit的commit，**git reset --hard HEAD~3**即删除最近的三个commit（删除HEAD, HEAD^, HEAD~2），将HEAD指向HEAD~3。   
  
D) 永久删除最后几个commit

**引用**

$ git commit ...   
$ **git reset --hard HEAD~3**   (1)

(1) 最后三个commit（即HEAD, HEAD^和HEAD~2）提交有问题，你想永久删除这三个commit。   
  
E) 回滚merge和pull操作

**引用**

$ git pull                         (1)   
Auto-merging nitfol   
CONFLICT (content): Merge conflict in nitfol   
Automatic merge failed; fix conflicts and then commit the result.   
$ **git reset --hard**                 (2)   
$ git pull . topic/branch          (3)   
Updating from 41223... to 13134...   
Fast-forward   
$ git reset --hard ORIG\_HEAD       (4)

(1) 从origin拉下来一些更新，但是产生了很多冲突，你暂时没有这么多时间去解决这些冲突，因此你决定稍候有空的时候再重新pull。   
(2) 由于pull操作产生了冲突，因此所有pull下来的改变尚未提交，仍然再stage area中，这种情况下**git reset --hard** 与 **git reset --hard HEAD**意思相同，即都是清除index和working tree中被搞乱的东西。   
(3) 将topic/branch合并到当前的branch，这次没有产生冲突，并且合并后的更改自动提交。   
(4) 但是此时你又发现将topic/branch合并过来为时尚早，因此决定退滚merge，执行**git reset --hard ORIG\_HEAD**回滚刚才的pull/merge操作。说明：前面讲过，执行git reset时，git会把reset之前的HEAD放入.git/ORIG\_HEAD文件中，命令行中使用ORIG\_HEAD引用这个commit。同样的，执行pull和merge操作时，git都会把执行操作前的HEAD放入ORIG\_HEAD中，以防回滚操作。   
  
F) 在被污染的working tree中回滚merge或者pull

**引用**

$ git pull                         (1)   
Auto-merging nitfol   
Merge made by recursive.   
nitfol                |   20 +++++----   
...   
$ **git reset --merge ORIG\_HEAD**      (2)

(1) 即便你已经在本地更改了一些你的working tree，你也可安全的git pull，前提是你知道将要pull的内容不会覆盖你的working tree中的内容。   
(2) git pull完后，你发现这次pull下来的修改不满意，想要回滚到pull之前的状态，从前面的介绍知道，我们可以执行git reset --hard ORIG\_HEAD，但是这个命令有个副作用就是清空你的working tree，即丢弃你的本地未add的那些改变。为了避免丢弃working tree中的内容，可以使用**git reset --merge ORIG\_HEAD**，注意其中的--hard 换成了 --merge，这样就可以避免在回滚时清除working tree。   
  
G) 被中断的工作流程   
在实际开发中经常出现这样的情形：你正在开发一个大的feature，此时来了一个紧急的bug需要修复，但是目前在working tree中的内容还没有成型，还不足以commit，但是你又必须切换的另外的branch去fix bug。请看下面的例子

**引用**

$ git checkout feature ;# you were working in "feature" branch and   
$ work work work       ;# got interrupted   
$ git commit -a -m "snapshot WIP"                 (1)   
$ git checkout master   
$ fix fix fix   
$ git commit ;# commit with real log   
$ git checkout feature   
$ **git reset --soft HEAD^** ;# go back to WIP state  (2)   
$ **git reset**                                       (3)

(1) 这次属于临时提交，因此随便添加一个临时注释即可。   
(2) 这次reset删除了WIP commit，并且把working tree设置成提交WIP快照之前的状态。   
(3) 此时，在index中依然遗留着“snapshot WIP”提交时所做的uncommit changes，**git reset**将会清理index成为尚未提交"snapshot WIP"时的状态便于接下来继续工作。   
  
(H) Reset单独的一个文件   
假设你已经添加了一个文件进入index，但是而后又不打算把这个文件提交，此时可以使用git reset把这个文件从index中去除。

**引用**

$ **git reset -- frotz.c**                      (1)   
$ git commit -m "Commit files in index"     (2)   
$ git add frotz.c                           (3)

(1) 把文件frotz.c从index中去除，   
(2) 把index中的文件提交   
(3) 再次把frotz.c加入index   
  
(I) 保留working tree并丢弃一些之前的commit   
假设你正在编辑一些文件，并且已经提交，接着继续工作，但是现在你发现当前在working tree中的内容应该属于另一个branch，与这之前的commit没有什么关系。此时，你可以开启一个新的branch，并且保留着working tree中的内容。

**引用**

$ **git tag start**   
$ git checkout -b branch1   
$ edit   
$ git commit ...                            (1)   
$ edit   
$ git checkout -b branch2                   (2)   
$ **git reset --keep start**                    (3)

(1) 这次是把在branch1中的改变提交了。   
(2) 此时发现，之前的提交不属于这个branch，此时你新建了branch2，并切换到了branch2上。   
(3) 此时你可以用**reset --keep**把在start之后的commit清除掉，但是保持working tree不变。   
  
7. git revert   
git revert用于回滚一些commit。对于一个或者多个已经存在的commit，去除由这些commit引入的改变，并且用一个新的commit来记录这个回滚操作。这个命令要求working tree必须是干净的。   
git revert和git reset的功能很相似，但是有区别，具体如下。   
**git revert**用于用一个commit来记录并回滚早前的commit，经常是一些错误的提交。如果你想干脆扔掉working tree中的东西，可以使用**git reset --hard**   
比如   
A) **git revert HEAD~3**：丢弃最近的三个commit，把状态恢复到最近的第四个commit，并且提交一个新的commit来记录这次改变。   
B) **git revert -n master~5..master~2**：丢弃从最近的第五个commit（包含）到第二个（不包含）,但是不自动生成commit，这个revert仅仅修改working tree和index。   
  
8. git revert 和 git reset的区别   
1. git revert是用一次新的commit来回滚之前的commit，git reset是直接删除指定的commit。   
2. 在回滚这一操作上看，效果差不多。但是在日后继续merge以前的老版本时有区别。因为git revert是用一次逆向的commit“中和”之前的提交，因此日后合并老的branch时，导致这部分改变不会再次出现，但是git reset是之间把某些commit在某个branch上删除，因而和老的branch再次merge时，这些被回滚的commit应该还会被引入。   
3. git reset 是把HEAD向后移动了一下，而git revert是HEAD继续前进，只是新的commit的内容和要revert的内容正好相反，能够抵消要被revert的内容。   
  
9. 如何删除远程分支   
删除远程分支就是将本地的空分支push到远程即可。

**引用**

#查看远程分支   
$ **git ls-remote** *idc*   
Password:   
fa7dc3cd254c6fff683e20722284565b92d869ff HEAD   
14a62709ecadd11a266d234d19955f4679fa95ab refs/heads/cpp-1.0   
34b38625bce0aa4d4a4e266e20bba3e0ccd1b97e refs/heads/cpp-1.0.RC1   
3f40a21f20f51aaa74e2a6954b64d82506cd4adf refs/heads/cpp-1.1   
2f795085d57b6784a6358d97dbd0d1227891b01a refs/heads/distri   
  
#删除远程叫做diftri的分支   
$ **git push idc :distri**   
Password:   
To xxx@192.168.4.40:Project.git   
- [deleted]         distri   
  
#确认远程分支被删除   
$ **git ls-remote** *idc*   
Password:   
fa7dc3cd254c6fff683e20722284565b92d869ff HEAD   
14a62709ecadd11a266d234d19955f4679fa95ab refs/heads/cpp-1.0   
34b38625bce0aa4d4a4e266e20bba3e0ccd1b97e refs/heads/cpp-1.0.RC1   
3f40a21f20f51aaa74e2a6954b64d82506cd4adf refs/heads/cpp-1.1

9. 如何删除本地分支   
使用git branch命令就可以删除本地分支，比如

**引用**

**git branch -d** *toBeDelBranch*

10. 如何clone（克隆）远程仓库中的指定分支，而非默认的master分支   
在git clone 命令中使用-b参数指定分支名字即可，比如将远端aiotrade.git上的levelIISZ-1.1分支克隆下来：

**引用**

**git clone -b** levelIISZ-1.1 username@192.168.4.40:aiotrade.git