# Git status

查看版本库的状态。可以得知哪些文件发生了变化，哪些文件还没有添加到git库中等等。 建议每次commit前都要通过该命令确认库状态.

一定要格外注意那些提示为 “Changed but not updated:”的文件。 这些文件都是与上次commit相比发生了变化，但是却没有通过git add标识的文件。

### Git status –s

# Git pull

直接拉最新的代码

不建议使用

# Git release

每一次的commit是一次快照，该命令会跟被修改文件每一次commit的快照进行合并文件，这比merge显示的更详细，速度比较慢。

# Git branch

查看本地分支

git branch –r 查看远程分支

# Git remote

查看远程分支

## 还原代码

例如 “git checkout app/model/user.rb” 就会将user.rb文件从上一个已提交的版本中更新回来，未提交的工作目录中的内容全部会被覆盖

### Git checkout .

这时候后面一个点表示所有修改未add的都退回去。

# 创建一个分支

## 1） git branch 分支名

虽然创建了分支，但是不会将当前工作分支切换到新创建的分支上，因此，还需要命令“git checkout 分支名” 来切换，

## 2） git checout –b 分支名

不但创建了分支，还将当前工作分支切换到了该分支上。

# 切换分支

切换到某个分支：git checkout 分支名   
切换到主分支：git checkout master

# 删除分支

## 删除本地分支

git branch –D 分支名   
注意： 删除后，发生在该分支的所有变化都无法恢复。强制删除此分支。

## 删除远程分支

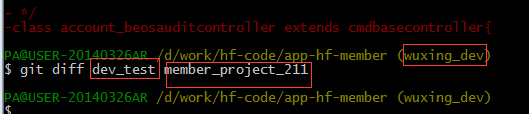
一不小心把本地的临时分支push到server上去了，想要删除。  
一开始用  
  
git branch -r -d origin/branch-name  
  
不成功，发现只是删除的本地对该远程分支的track，正确的方法应该是这样：  
  
git push origin :branch-name  
  
冒号前面的空格不能少，原理是把一个空分支push到server上，相当于删除该分支。

# 比较两个分支上的文件的区别

git diff master 分支名 （比较主分支和另一个分支的区别）

git diff master (比较master与自己所在分支区别)

可在一个分支上比较另外两个分支，如图



Git diff 表示当前版本对比

# 查看分支历史

git-show-branch （查看当前分支的提交注释及信息）   
       git-show-branch -all（查看所有分支的提交注释及信息）例如：

\* [dev] d2   
! [master] m2   
--   
\* [dev] d2   
\* [dev^] d1   
\* [dev~2] d0   
\*+ [master] m2   
在上述例子中， “--”之上的两行表示有两个分支dev和master， 且dev分支上最后一次提交的日志是“d2”,master分支上最后一次提交的日志是 “m2”。 “--”之下的几行表示了分支演化的历史，其中 dev表示发生在dev分支上的最后一次提交，dev^表示发生在dev分支上的倒数第二次提交。dev~2表示发生在dev分支上的倒数第三次提交。

# 查看当前分支的操作记录

git whatchanged

# 合并分支

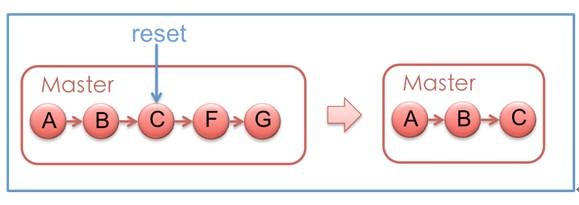
## 法一：

git merge “注释” 合并的目标分支 合并的来源分支   
如果合并有冲突，git会有提示。   
例如：git checkout master   （切换到master分支）   
       git merge HEAD dev~2 (合并master分支和dev~2分支)或者：git merge master dev~2

## 法二：

git pull 合并的目标分支 合并的来源分支   
例如: git checkout master （切换到master分支）   
       git pull . dev~2（合并当前分支和dev~2分支）

# Git reset

库的逆转与恢复除了用来进行一些废弃的研发代码的重置外，还有一个重要的作用。比如我们从远程clone了一个代码库，在本地开发后，准备提交回远程。但是本地代码库在开发时，有功能性的commit，也有出于备份目的的commit等等。总之，commit的日志中有大量无用log，我们并不想把这些 log在提交回远程时也提交到库中。 因此，就要用到git reset。   
       git reset的概念比较复杂。它的命令形式：git reset [--mixed | --soft | --hard] [<commit-ish>]   
命令的选项：   
       --mixed 这个是默认的选项。如git reset [--mixed] dev^(dev^的定义可以参见2.6.5)。它的作用仅是重置分支状态到dev1^, 但是却不改变任何工作文件的内容。即，从dev1^到dev1的所有文件变化都保留了，但是dev1^到dev1之间的所有commit日志都被清除了， 而且，发生变化的文件内容也没有通过git add标识，如果您要重新commit，还需要对变化的文件做一次git add。 这样，commit后，就得到了一份非常干净的提交记录。 （回退了index和仓库中的内容）   
       --soft相当于做了git reset –mixed，后，又对变化的文件做了git add。如果用了该选项， 就可以直接commit了。（回退了仓库中的内容）   
       --hard这个命令就会导致所有信息的回退， 包括文件内容。 一般只有在重置废弃代码时，才用它。 执行后，文件内容也无法恢复回来了。（回退了工作目录、index和仓库中的内容）   
例如：   
切换到使用的分支上；   
       git reset HEAD^ 回退第一个记录   
       git reset HEAD~2 回退第二个记录   
如果想把工作目录下的文件也回退，则使用git reset - - hard HEAD^ 回退第一个记录   
       git reset - - hard HEAD~2 回退第二个记录   
还可以使用如下方法：   
将当前的工作目录完全回滚到指定的版本号，假设如下图，我们有A-G五次提交的版本，其中C的版本号是 bbaf6fb5060b4875b18ff9ff637ce118256d6f20，我们执行了'git reset bbaf6fb5060b4875b18ff9ff637ce118256d6f20'那么结果就只剩下了A-C三个提交的版本   


# Git revert

还原某次对版本的修改，例如：git revert commit\_id （其中commit\_id为commit代码时生成的一个唯一表示的字符串）   
例如：（3.6中）git revert dfb02e6e4f2f7b573337763e5c0013802e392818 （执行此操作，则还原上一次commit的操作）

# Git show

显示对象的不同类型。

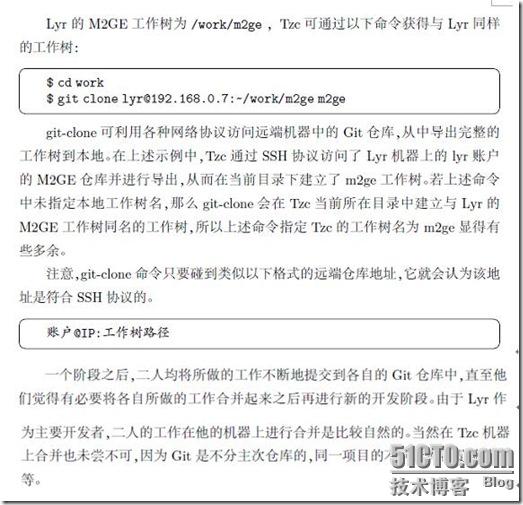
# Git tag

创建、列出、删除或者验证一个标签对象（使用GPG签名的）。   
可以将某个具体的版本打上一个标签，这样就不需要记忆复杂的版本号哈希值字符串了，例如你可以使用 “git tag revert\_version bbaf6fb5060b4875b18ff9ff637ce118256d6f20” 来标记这个被你还原的版本，那么以后你想查看该版本时，就可以使用 revert\_version标签名，而不是哈希值了。

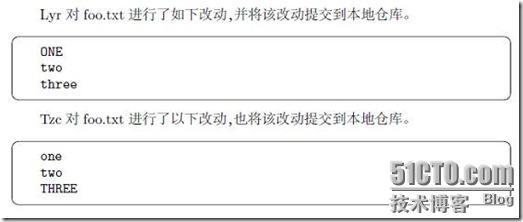
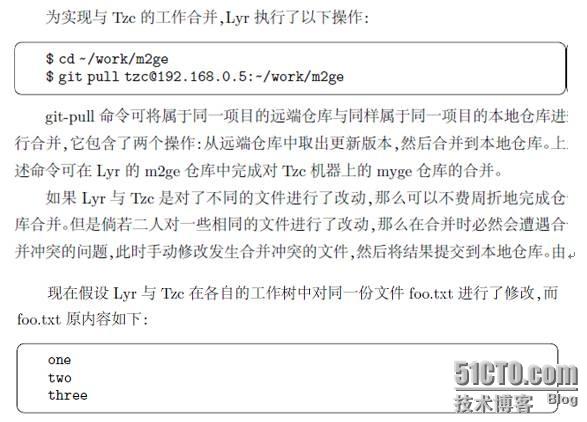
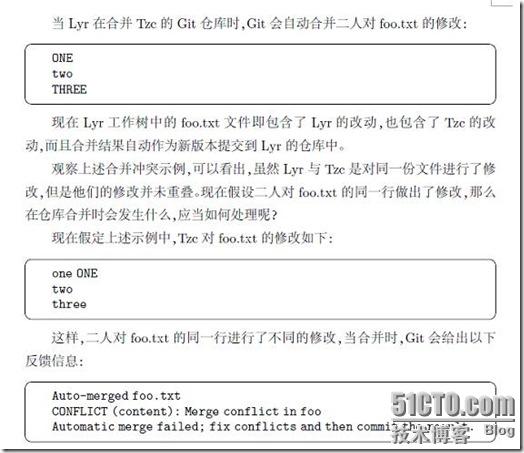
# Git服务器操作命令（与服务器交互）

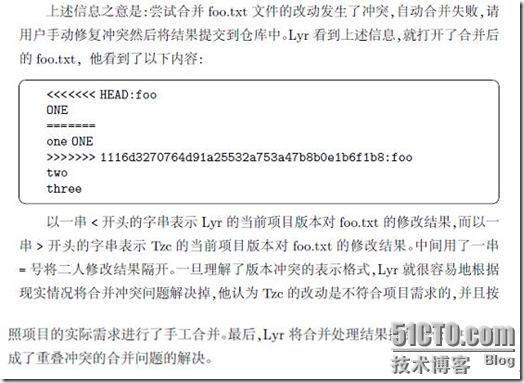
# Git clone

取出服务器的仓库的代码到本地建立的目录中（与服务器交互）   
通过git clone获取远端git库后，.git/config中的开发者信息不会被一起clone过来。仍然需要为本地库的.git/config文件添加开发者信息。此外，开发者还需要自己添加   . gitignore文件。   
通过git clone获取的远端git库，只包含了远端git库的当前工作分支。如果想获取其它分支信息，需要使用 “git branch –r” 来查看， 如果需要将远程的其它分支代码也获取过来，可以使用命令 “ git checkout -b 本地分支名 远程分支名”，其中，远程分支名为 “git branch –r” 所列出的分支名， 一般是诸如“origin/分支名”的样子。如果本地分支名已经存在， 则不需要“-b”参数。   
例如：



# Git pull

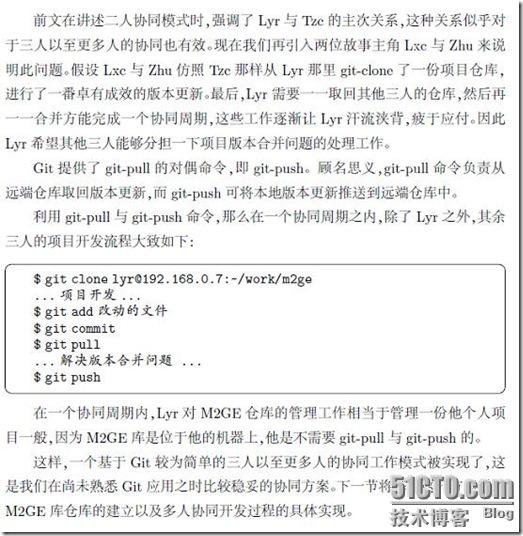
从服务器的仓库中获取代码，和本地代码合并。（与服务器交互，从服务器上下载最新代码，等同于： Git fetch + Git merge）   
从其它的版本库（既可以是远程的也可以是本地的）将代码更新到本地，例如：“git pull origin master ”就是将origin这个版本库的代码更新到本地的master主分支。   
       git pull可以从任意一个git库获取某个分支的内容。用法如下：   
**git pull [username@ipaddr](mailto:username@ipaddr" \t "\\):远端repository名远端分支名 本地分支名**。这条命令将从远端git库的远端分支名获取到本地git库的一个本地分支中。其中，如果不写本地分支名，则默认pull到本地当前分支。   
需要注意的是，git pull也可以用来合并分支。 和git merge的作用相同。 因此，如果你的本地分支已经有内容，则git pull会合并这些文件，如果有冲突会报警。   
例如：   




# Git push

git push origin <local\_branch\_name>:<remote\_branch\_name>

将本地commit的代码更新到远程版本库中，例如 “git push origin”就会将本地的代码更新到名为orgin的远程版本库中。   
       git push和git pull正好想反，是将本地某个分支的内容提交到远端某个分支上。用法： **git push**[**username@ipaddr**](mailto:username@ipaddr)**:远端repository名本地分支名 远端分支名**。这条命令将本地git库的一个本地分支push到远端git库的远端分支名中。   
需要格外注意的是，git push好像不会自动合并文件。因此，如果git push时，发生了冲突，就会被后push的文件内容强行覆盖，而且没有什么提示。 这在合作开发时是很危险的事情。   
例如：



# Git fetch

从服务器的仓库中下载代码。（与服务器交互，从服务器上下载最新代码）   
相当于从远程获取最新版本到本地，不会自动merge，比Git pull更安全些。   
使用此方法来获取服务器上的更新。   
例如：如果使用git checkout nov/eclair\_rocket （nov/eclair\_rocket为服务器上的分支名），则是获取上次使用git fetch命令时从服务器上下载的代码；如果先使用 git fetch ，再使用git checkout nov/eclair\_rocket，则是先从服务器上获取最新的更新信息，然后从服务器上下载最新的代码。

git fetch origin master:tmp  
git diff tmp   
git merge tmp  
git branch -d tmp  
      
    从远程获取最新的版本到本地的tmp分支上  
   之后再进行比较合并  
  
2. git pull：相当于是从远程获取最新版本并merge到本地  
     
git pull origin master  
  
  
    上述命令其实相当于git fetch 和 git merge  
    在实际使用中，git fetch更安全一些  
   因为在merge前，我们可以查看更新情况，然后再决定是否合并

# Git merge

每一次的commit是一次快照，该命令会跟被修改文件最新的快照进行合并文件。

把服务器上下载下来的代码和本地代码合并。或者进行分支合并。   
例如：当前在master分支上，若想将分支dev上的合并到master上，则git merge dev   
注意：git merge nov/eclair\_eocket （是将服务器git库的eclair\_eocket分支合并到本地分支上）   
       git rebase nov/eclair\_eocket （是将服务器git库的eclair\_eocket分支映射到本地的一个临时分支上，然后将本地分支上的变化合并到这个临时分支，然后再用这个临时分支初始化本地分支） （多了一个本地分支）

git merge branchA branchB, branchB 一般默认为当前branch，所以

git **merge** origin **master** //将origin **merge** 到 **master** 上

git **merge** origin/**master** //将origin上的**master**分支 **merge** 到当前 branch 上

一般进行merge操作时，最好先checkout到你希望进行merge操作的分支，也就是branchB上，然后再进行

git merge branchA //默认为当前branch，即branchB

因为你不能确保是否产生confict，所以先checkout到branchB上。  
此外，为了保险起见，你可以在merge前先建一个备份分支

**git** checkout -b branchB\_backup

这样即使你merge后出现不可控的情况，还可以check回来。

# Git checkout

## 切换到分支

1) 创建一个新分支，并切换到该分支上   
       Git checkout –b 新分支名   
2）切换到某个已经建立的本地分支local\_branch   
       Git checkout local\_branch   
（使用cat .git/HEAD后，显示refs:refs/heads/ local\_branch）   
3) 切换到服务器上的某个分支remote\_branch   
       Git checkout remote\_branch   
（远程分支remote\_branch可以通过 git branch –r 列出）   
4) 切换到某个commit id   
       Git checkout commit\_id   
（使用cat .git/HEAD后，显示commit\_id）

## 用已有分支初始化新分支

执行下面的命令，在切换到某个已经建立的local branch或者某个remote branch或者某个commit id 或者某个tag的同时，创建新分支new\_branch，并且挂到这个新分支上。   
1） 切换到某个已经建立的本地分支local\_branch，并且使用此分支初始化一个新分支new\_branch。   
       git checkout –b new\_branch local\_branch   
2) 切换到某个远程分支remote\_branch，并且用此分支初始化一个新分支new\_branch。   
       Git checkout –b new\_branch remote\_branch   
3) 切换到某个commit id，并建立新分支new\_branch   
       Git checkout –b new\_branch commit\_id   
4) 切换到某个tag，并建立新分支new\_branch   
       Git checkout –b new\_branch tag

# Git reflog 和Git reset配合还原提交

