刚刚开始使用不久，经常会在Merge时出现没有change-id的情况，在配置gerrit使用经常不能提交，使得自己很很困扰。最近有次熬夜加班，在代码完成后由于多人提交次数很平凡，出现了提交不上去的情况，当时那个慌啊。还好有给力的同事帮忙处理，突然觉得有必要好好学学Git了，不能仅仅满足于图形化工具的使用。

* 工作区、版本库、缓存区的关系与区别，如下图：



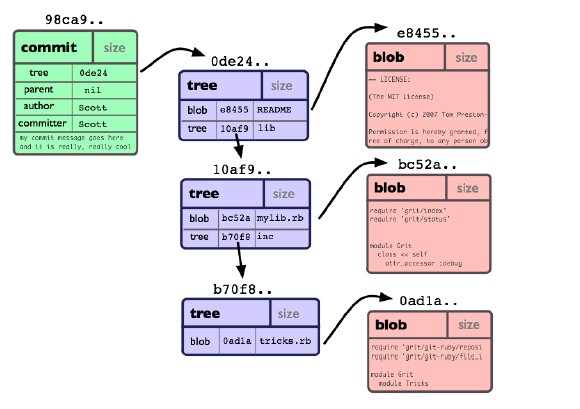
工作区：左侧的工作区就是我们日常编辑的部分

暂存区：使用git add XXX后的部分

版本库：使用commit后的部分

HEAD：当前版本指针

* Git中的tree，blob,commit,tag的概念，

[](http://blog.chinaunix.net/link.php?url=http://blog.chinaunix.net%2Fattachment%2F201108%2F24%2F25524263_1314185213KZ1F.jpg)

Blob: 就是一块内存区域，其中内容可以是文本，源码或者图片

Tree: 很想文件系统中的目录，可以指向blob或者其他的树，就像目录可以包含文件和子目录一样

Commit: 包含提交者的信息（姓名，Email），指向当前提交下所属的tree的指针，指向之前提交（父提交）的指针

Tag: 包含指向任意commit的指针，便于记录和使用指定的tree，而不用使用哈希值。

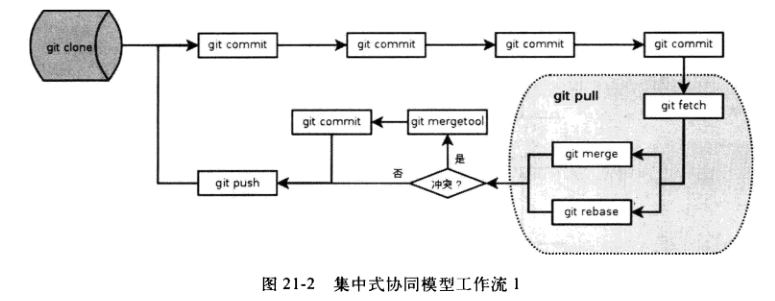
|  |  |
| --- | --- |
| Git命令 | 解释 |
| Git status | 查看当前状态 |
| Git diff |  |
| Git diff head |  |
| Git diff --cached/--staged |  |
| Git checkout | 注意检出时会修改指针、暂存区、工作区，因此要注意该操作可能会丢失修改 |
| Git checkout - | 切换到上一个分支 |
| Git checkout <branch> | 切换到指定branch |
| Git checkout –b new\_branch | 检出并创建分支 |
| Git branch |  |
| Git branch –r |  |
| Git commit –m ‘XXX’ | 提交并添加评论，需要注意的是提交什么的哈希码，是通过SHA1算法获得的160bit哈希值，在分布式版本控制系统中需要使用SHA1来唯一标识，而不是顺序ID。 |
| Git commit –am ‘XXX’ |  |
| Git commit --allow-empty | 允许空白提交 |
| Git commit --amend | 对刚刚的提交进行修补，不会产生新的提交。有时，在merge操作后，在gerrit出现不能提交的情况，可以通过该命令，为merge commit产生一条changeID。其原理其实是组合了两条git命令，如下所示:  Git reset --soft HEAD^  Git commit –e –F .git/COMMIT\_EDITMSG |
| Git add -i | 进入交互式界面添加/修改/删除文件到暂存区 |
| Git add some\_file |  |
| Git rebase |  |
| Git rebase –i | 合并提交 |
| Git clean –nd | 查看那些文件和目录会被删除 |
| Git clean –fd | 强制删除多余的目录和文件 |
| Git reset | 省略commit标识号，相当于使用了HEAD的指向作为提交ID。 |
| Git reset --hard some\_commit\_hashcode | 包含3个步骤：a.替换引用的指向；b.替换暂存区；c.替换工作区 |
| Git reset --soft <commit> | 包含1个步骤：a.替换引用的指向； |
| Git reset --mixed <commit>，默认方式 | 包含2个步骤：a.替换引用的指向；b.替换暂存区； |
| Git reset --soft HEAD^ | 工作区和暂存区不变，但引用向前回退一次 |
| Git stash | 保存当前进度，会分别对暂存区和工作区的状态进行保存，完整版本为git stash save “XXX” |
| Git stash list | 查看保存的进度列表 |
| ~~Git stash pop~~ | 从最近保存的进度进行恢复，并将该进度清除， |
| Git stash apply | 恢复进度，并且不删除 |
| Git stash clear | 删除所有存储的进度 |
| Git stash branch branch\_name | 基于进度创建分支 |
| Git log | 查看日志 |
| Git log -i | 查看指定条目数日志 |
| Git log --graph | 以图形化的方式查看日志，在理解分支信息时很有用 |
| Git log –graph –pretty=oneline | 只查看单行的commitID |
| Git reflog |  |
| Git config –global user.name | Git相关环境变量配置 |
| Git config –global alias.ci “commit -s” | 设置git操作别名，例如用ci表示类似SVN中的check in操作 |
| Vim file\_name |  |
| Git grep | 工作区文件内容搜索 |
| Git clone XXX | 拉代码，可以通过https和ssh等方式 |
| ls | 显示当前路径下的目录和文件 |
| Git rm/mv | 在Git中删除/移动文件 |
| Gitk --all | 最早的图形化工具 |
| Git blame | 追溯文件的修改记录（前提是bug已经定位到代码上） |
| Git bisect | 二分查找， |
| Git describe |  |
| Git tag | 查看里程碑列表 |
| Git tag tagname | 轻量级里程碑，在.git/refs/tags中。其他还是重量级里程碑等内容，包括数字签名。 |
| Git cherry-pick <branch>[^^] | 拣选分支 |
| Git revert HEAD | 将HEAD提交反向再提交一次。由于修改历史操作只能是针对自己的版本库，而无法去修改别人的版本库，这时就可能需要使用revert去修正一个错误的历史提交。 |
| Git init |  |
| Git push | 注意要禁止非快进式推送，理解不深 |
| Git fsck | 查看到版本库中包含的没有被任何引用关联的松散对象（danglin开头）,可能是commit也可能是blob |
| Git prune | 清理临时文件（未被关联的松散对象） |
| Git gc | 垃圾回收，清理版本库 |
| Git fetch |  |
| Git merge |  |
| Git merge --no-ff | None fast-forward，不快速向前 |
| Git pull | 拉去本地分支关联的远程分支的内容，并merge |

Android项目包含近200个Git版本库，因而google公司开发了repo(对git的封装)和gerrit两个工具进行版本库管理，其中gerrit是一种特别的集中式协同模型，通过SSH协议管理Git版本库，并实现一个Web界面的评审工作流。其中，Git不能直接推送到分支，而是推送到特殊的引用refs/for/<branch-name>。

其中困扰我良久的change-id其实不是git中的概念，而是gerrit中的概念，它通过hooks的方式(其实就是面向切面的拦截器，在C语言中一般叫做hooks钩子，位于.git\hooks\文件夹中)为该次提交添加一个change-id，然后就可以被gerrit管理起来了。

**Gerrit**审核服务器最初其实是为Android项目开发。Redmine是一款实现需求管理和缺陷跟踪的项目管理软件，可以和Git版本库实现整合，git的提交可以之间关闭redmine上的Bug,同时Git的提交还可以反映出项目成员的工作进度。**Redmine**中的用户（项目成员）用一个ID做标识，而Git的提交者则用一个包含用户名和邮件地址的字符串，需要一个关联配置。

传统SVN集中式系统模型图如下所示：



Tip:

主要供自己工作参考，若有疏漏，望见谅。参考蒋鑫大师的《GIT权威指南》和[大塚弘记](http://www.dangdang.com/author/%B4%F3%89V%BA%EB%BC%C7_1)的GitHub入门与实践，前者非常全面，后者简单有效。