Git课件

目录

[第一章 Git简介 2](#_Toc492889836)

[1.1 git简介 2](#_Toc492889837)

[1.2 git的诞生 2](#_Toc492889838)

[1.3 分布式和集中式 3](#_Toc492889839)

[第二章 Git工作流程 3](#_Toc492889840)

[2.1 工作区 4](#_Toc492889841)

[2.2 暂存区 4](#_Toc492889842)

[2.3 本地仓库 4](#_Toc492889843)

[2.4 远程仓库 4](#_Toc492889844)

[2.5 总结 5](#_Toc492889845)

[第三章 Git常用命令 6](#_Toc492889846)

[3.1 HEAD 6](#_Toc492889847)

[3.2 add 7](#_Toc492889848)

[3.3. commit 7](#_Toc492889849)

[3.4 branch 8](#_Toc492889850)

[3.5 merge 9](#_Toc492889851)

[3.6 rebase 10](#_Toc492889852)

[3.7 reset 11](#_Toc492889853)

[3.8 revert 12](#_Toc492889854)

[3.9 push 13](#_Toc492889855)

[3.10 其他命令 13](#_Toc492889856)

# Git简介

## git简介

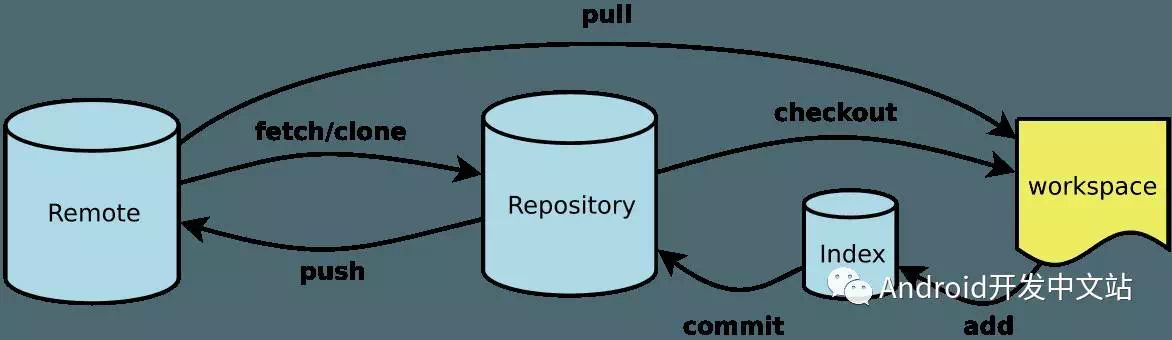
git是目前世界上最先进的分布式版本控制系统

## git的诞生

很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等

## 分布式和集中式

# Git工作流程



以上包括一些简单的命令，但是我们先别关心这些，我们先了解一些专有名词

Workspace：工作区

Index/Stage：暂存区

Repository：仓库区（本地仓库）

Remote：远程仓库

## 2.1 工作区

程序员进行开发改动的地方，是你当前看到的 ，也是最新的。

通常我们开发就是拷贝远程仓库中的一个分支，基于该分支进行开发。在开发过程中就是对工作区的操作

## 2.2 暂存区

Git目录下的index文件，暂存区会记录git add 添加文件的相关信息（文件名，大小，timestamp），不保存文件实体。可以通过git status查看暂存区状态。暂存区标记了你当前工作区中，哪些内容被git管理

当你完成某一个需求或者功能后需要提交到远程仓库，那么第一步就是通过git add先提交到暂存区，被git管理

## 2.3 本地仓库

保存了对象被提交过的各个版本，比起工作区和暂存区的内容，他会更旧一点。

Git commit后同步index的目录树到本地仓库，方便从下一步通过git push同步本地仓库与远程仓库的同步

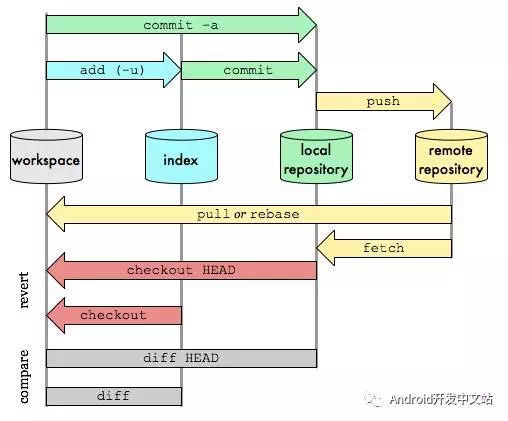
## 2.4 远程仓库

远程仓库的内容可能被分布在多个地点的处于协作关系的本地仓库修改，因此他可能与本地仓库同步，也有可能不同步。但是他的内容是最新的

## 2.5 总结

1. 任何对象都是在工作区中诞生和被修改的
2. 任何修改都是从进入index去才开始被版本控制
3. 只有把修改提交到本地仓库，该修改才能在仓库中留下痕迹
4. 与协作者分享本地的修改，可以把他们push到远程仓库来共享

下面这张图更加直接阐述了四个区域之间的管理

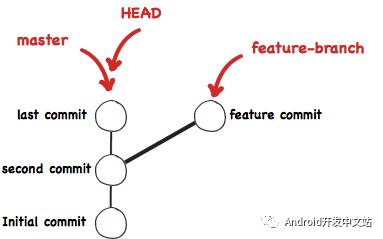


# Git常用命令



下面我们来解释一下这些 常用命令

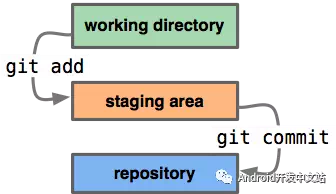
## 3.1 HEAD



在掌握具体命令之前，先理解一下HEAD

HEAD，他始终只想当前所在分支的最新的提交点。你所处的分支发生变化了，或者产生了新的提交点，HEAD就会发生改变

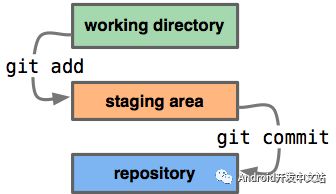
## 3.2 add



Add相关的命令比较简单，主要实现将工作区修改的内容提交到缓存区，交给git管理

|  |  |
| --- | --- |
| Git add . | 添加当前目录的所有文件到缓存区 |
| Git add <dir> | 添加制定目录到暂存区，包括子目录 |
| Git add <file1> | 添加制定文件到缓存区 |

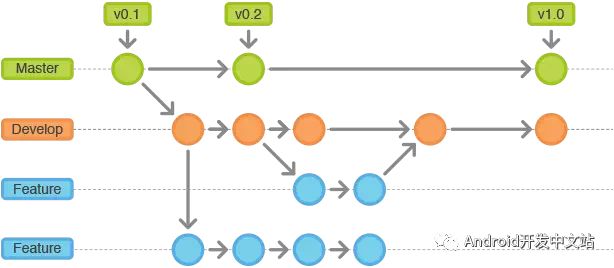
## 3.3. commit



Commit相关的命令也很简单，主要实现将暂存区的内容提交到北地仓库，病史当前 分支的HEAD向后移动一个提交点

|  |  |
| --- | --- |
| Git commit -m <message> | 提交暂存区到本地仓库，message代码说明了信息 |
| Git commit <file1> -m <message> | 提交缓存区的制定文件到远程仓库 |
| Git commit -amend -m <message> | 使用一次新的commit，替代上一次的提交 |

## 3.4 branch

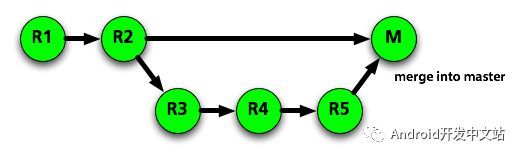


涉及到协作，自然会涉及到分支，关于分支大概有展示分支，切换分支，创建分支，删除分支

|  |  |
| --- | --- |
| Git branch | 列出所有本地分支 |
| Git branch -r | 列出所有远程分支 |
| Git branch -a | 列出所有本地分支和远程分支 |
| Git branch <branch-name> | 创建一个分支，但是依然停留在当前分值 |
| Git checkout -b <branch-name> | 创建一个分支，并且切换到该分支上 |
| Git branch -track <branch> <remote-branch> | 创建一个分支，与指定的远程分支建立追踪关系 |
| Git checkout <branch-name> | 切换到制定分支，并且更新工作区 |
| Git branch -d <branch-name> | 删除分支 |
| Git push origin -delete <branch-name> | 删除远程分支 |

关于分支的操作虽然比较多，但是也都是比较好理解的

## 3.5 merge



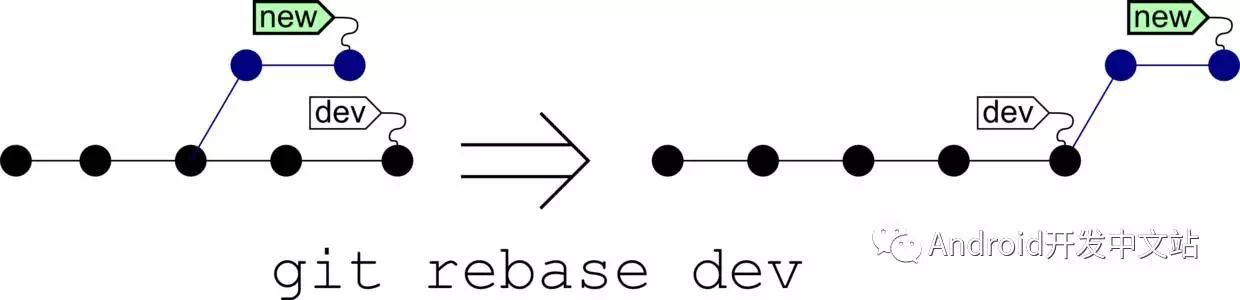
Merge命令吧不同的分支合并起来，如上图所示，在实际开发中，我们可能master分支中切出一个分支，然后进行开发完成需求，中间经过R3，R4，R5的commit记录，最后开发完成需要融合如master中，这边用到了merge

|  |  |
| --- | --- |
| Git fetch <remote> | Merge之前先拉一下远程仓库的最新代码 |
| Git merge <branch> | 合并指定分支到当前分支 |

一般在merge之后，会出现conflict，需要针对冲突情况，手动解除冲突，主要是因为两个用户修改了同一个文件的同一块区域，如下图所示



## 3.6 rebase

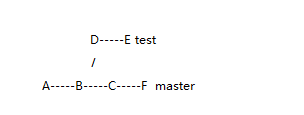


Rebase又称衍合，是合并的另一种选择

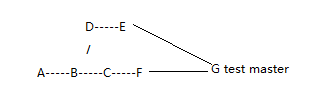
在开始阶段，我们处于new分支上，执行git rebase dev，那么new分支上新的commit都在master分支上重演一遍，最后checkout切换回到new分支。这一点与merge是一样的，合并前后所处的分支并没有改变。Git rebase dev ，通俗的解释就是new分支想站在dev的肩膀上继续下去，rebase也需要手动解决冲突

Rebase和merge的区别：

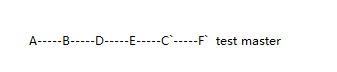
现在我们有两个分支master和test，提交如下



在master执行git merge test ，然后会得到如下的结果，



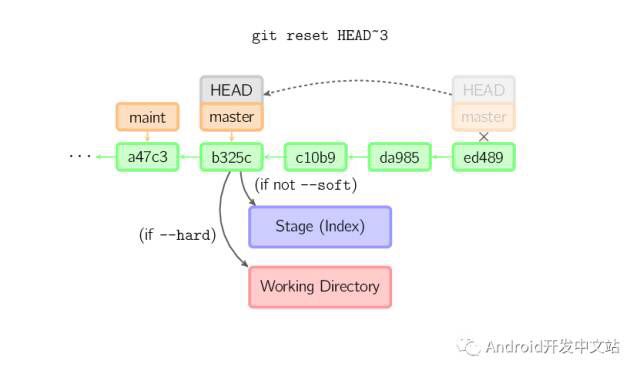
在master执行git rebase test ，然后得到如下的结果



可以看到，merge操作会生成新的节点，之前的提交分开显示。二rebase操作不会生成新的节点，是将两个分支融合成一个线性的提交。

如何你想要一个干净的，没有merge commit的线性历史树，那么你应该选择git rebase，如果你想保留完成的历史记录，并且想要避免重写commit history的风险，那么久使用git merge

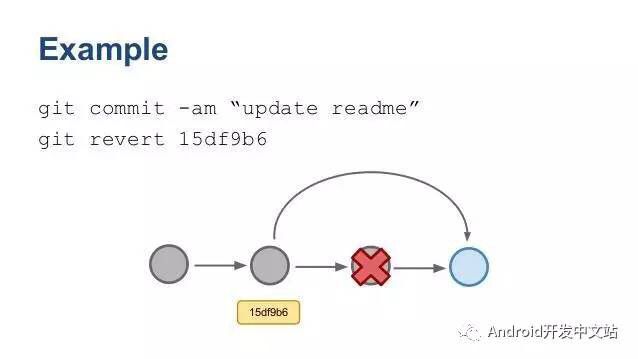
## 3.7 reset



Reset命令吧当前分值指向另一个位置，并且相应的变动工作区和暂存区

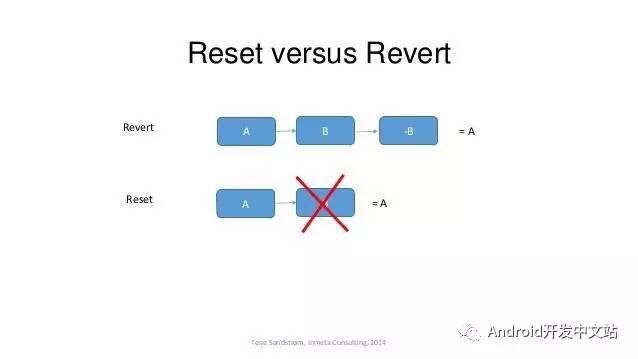
|  |  |
| --- | --- |
| Git reset -soft <commit> | 只改变提交点，暂存区和工作目录的内容不变 |
| Git reset -mixed <commit> | 改变提交点，同时改变暂存区的内容 |
| Git reset -hard <commit> | 暂存区，工作区的内容都会被修改到与提交点完全一致的状态 |
| Git reset -hard head | 让工作区回到上一次提交时候的状态 |

## 3.8 revert



Git revert用一个新提交来消除一个历史提交所做的任何修改

Revert和reset的区别



Git revert是用一次新的commit来回滚到之前的commit，git reset是直接删除制定的commit

在回滚这一个操作上，效果看似差不多。但是在日后继续merge以前的老版本是有区别。因为git revert是用一次逆向的commit，中和了之前的提交，因此日后合并老的branch时，导致这部分改变不会再次出现，减少冲突。但是git reset是把某些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，产生很多冲突。

Git reset是把HEAD向后移动了一下，而git revert是HEAD继续向前进，只是新的commit的内容和要revert的内容刚好相反，能够抵消要被revert的内容

## 3.9 push

上传本地仓库分支到远程仓库分支，实现同步

|  |  |
| --- | --- |
| Git push <remote> <branch> | 上传本地指定分支到远程仓库 |
| Git push <remote> -force | 强行推送当前分支到远程仓库，即使有冲突 |
| Git push <remote> -all | 推送所有分支到远程仓库 |

## 3.10 其他命令

|  |  |
| --- | --- |
| Git status | 显示有变更的文件 |
| Git log | 显示当前分支的版本历史 |
| Git diff | 显示暂存区和工作区的差异 |
| Git diff HEAD | 显示工作区与当前分支最新commit之间的差异 |
| Git cherry-pick <commit> | 选择一个commit，合并当前分支 |