**14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. 概述 3](#_Toc18399)

[2. 准备 3](#_Toc1377)

[3. 创建local仓库 4](#_Toc19891)

[4. 对某个仓库单独设置local 4](#_Toc17701)

[5. 往仓库里添加文件 4](#_Toc13511)

[6. commit和tree和blob的关系 5](#_Toc20074)

[7. 分离头指针 6](#_Toc18584)

[8. rebase的简单使用 6](#_Toc17083)

[8.1. 修改某次的commit信息 6](#_Toc8320)

[8.2. 连续多个commit合成一个 7](#_Toc20507)

[8.3. 间隔的多个commit合成一个 9](#_Toc7739)

[9. 比较差异diff 10](#_Toc13815)

[9.1. head和缓存区差异的比较 10](#_Toc23968)

[9.2. 工作区和缓存区差异的比较 10](#_Toc31356)

[9.3. 两个commit、分支、head间的差异 11](#_Toc1809)

[9.4. 针对性的比较某些文件 11](#_Toc7114)

[10. 恢复工作区和暂存区 12](#_Toc17001)

[10.1. 暂存区恢复成和Head一样的 12](#_Toc9118)

[10.2. 工作区恢复成和暂存区一样的 12](#_Toc13945)

[10.3. 消除最近几次提交 12](#_Toc31077)

[11. 突然来新的紧急bug，stash暂存当前的工作 12](#_Toc8533)

[11.1. stash的策略 12](#_Toc28123)

[12. .gitignore忽略 13](#_Toc14323)

[13. 远程仓库 14](#_Toc5279)

[13.1. 克隆本地仓库到一个地方 14](#_Toc12275)

[13.2. 查看、添加、删除远程仓库 14](#_Toc25440)

[13.3. 推送本地仓库的改动到远程 15](#_Toc11405)

[13.4. 拉取远程仓库的改动 15](#_Toc2967)

[13.5. ssh协议 15](#_Toc20466)

[13.6. github上仓库的操作实例 15](#_Toc27369)

[14. 极客时间--------------旧资料 16](#_Toc31903)

[15. 什么是Git和GitHub 16](#_Toc3995)

[15.1. Git ： 16](#_Toc28283)

[15.2. GitHub： 16](#_Toc31180)

[16. Git的常用命令图 16](#_Toc20920)

[17. 工作流程Flow 16](#_Toc18249)

[17.1. Git flow 17](#_Toc28101)

[17.1.1. 特点 17](#_Toc10041)

[17.1.2. 评价 17](#_Toc27864)

[17.2. Github flow 17](#_Toc7156)

[17.2.1. 流程 17](#_Toc19683)

[17.3. 评价 18](#_Toc1216)

[17.4. Gitlab flow 18](#_Toc11488)

[17.4.1. 上游优先 18](#_Toc30282)

[17.4.2. 持续发布 18](#_Toc480)

[17.4.3. 版本发布 19](#_Toc24595)

[18. GitHub中的watch、star、fork 19](#_Toc22854)

[18.1. watch 19](#_Toc11827)

[18.2. star 19](#_Toc14861)

[18.3. fork 20](#_Toc21006)

[18.4. 使用建议 20](#_Toc1293)

[18.5. 小细节 20](#_Toc12796)

[19. AS中初次使用Git 20](#_Toc29850)

[19.1. 安装Git 20](#_Toc16116)

[19.2. Android Studio配置Git 21](#_Toc7669)

[19.3. 创建本地仓库 22](#_Toc10420)

[19.4. 忽略文件 23](#_Toc11108)

[19.5. 添加文件 23](#_Toc11260)

[19.6. 如何clone项目 26](#_Toc31727)

[20. 项目中使用Git 28](#_Toc5078)

[20.1. Git flow 28](#_Toc32270)

[20.2. 特点 28](#_Toc3596)

[20.3. 新增功能1——显示Hello World 28](#_Toc1235)

[20.3.1. 创建dev、feature-1分支，并且我们需要转到feature-1分支上 28](#_Toc13353)

[20.3.2. 完成功能1 29](#_Toc4947)

[20.3.3. 将feature-1分支合并到dev分支上 32](#_Toc30148)

[20.3.4. 删除feature-1分支 33](#_Toc14421)

[20.4. 新增功能2——显示Hello Android 33](#_Toc13797)

[20.4.1. 迁出dev分支，并且新建分支feature-2之后迁出feature-2分支 33](#_Toc9370)

[20.4.2. 完成功能2 33](#_Toc15550)

[20.4.3. 将feature-2分支合并到dev分支上 34](#_Toc14390)

[20.5. 增加功能3——Hello Java 34](#_Toc26647)

[20.6. 删除功能3 35](#_Toc26571)

[20.6.1. Git revert 35](#_Toc6164)

[20.6.1.1. 复制哈希码 35](#_Toc8375)

[20.6.1.2. Terminal 输入 git revert 35](#_Toc838)

[20.6.1.3. 保存revert 36](#_Toc19298)

[20.6.2. Git reset 36](#_Toc13347)

[20.6.2.1. HEAD~2 36](#_Toc17747)

[20.6.2.2. 点击Reset 36](#_Toc17239)

[20.7. master分支上被修改了，使用Rebase 37](#_Toc14530)

[20.7.1. 同事添加新特性 37](#_Toc15759)

[20.7.1.1. 添加feature-1和feature-4 37](#_Toc9645)

[20.7.1.2. 演示图 38](#_Toc22702)

[20.7.2. Rebase 38](#_Toc20683)

[20.7.2.1. 点击菜单栏上的VCS-->Git-->Rebase，如下图所示： 39](#_Toc31624)

[20.7.2.2. 在Onto的下拉栏中选中dev 39](#_Toc11976)

[20.7.2.3. 如果有冲突 40](#_Toc29846)

[20.7.2.4. 演示图 41](#_Toc22190)

[20.8. 推送到远程仓库 42](#_Toc24395)

[20.8.1. 添加到GitHub 42](#_Toc7286)

[20.8.2. 添加到自定义url地址下（eg，gitLab） 42](#_Toc29923)

[20.9. 如何取消git关联 43](#_Toc15616)

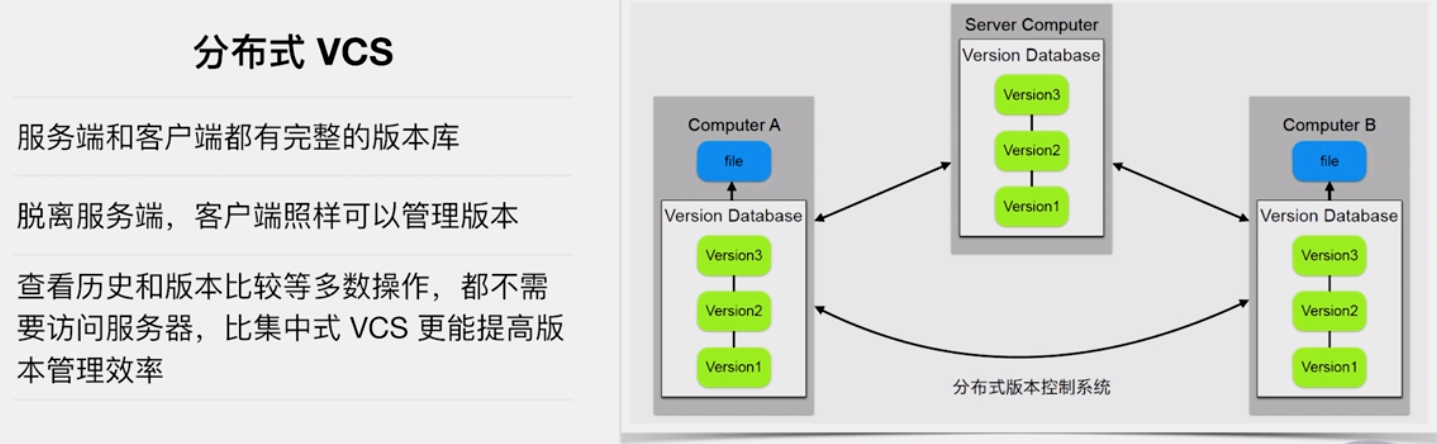
[20.10. 查看git远程仓库地址 43](#_Toc11499)

[21. 遇到了early EOF index-pack failed问题 43](#_Toc3859)

# 概述

版本控制系统VCS（version control system）的演变





# 基础用法

## 准备

官方文档地址：<https://git-scm.com/book/zh/v2>。

官方版本可以在 Git 官方网站下载。 打开 <http://git-scm.com/download/win>，下载会自动开始。要注意这是一个名为 Git for Windows的项目（也叫做 msysGit），和 Git 是分别独立的项目；更多信息请访问 http://msysgit.github.io/。可以直接安装 GitHub for Windows。该安装程序包含图形化和命令行版本的 Git。 它也能支持 Powershell，提供了稳定的凭证缓存和健全的 CRLF 设置。可以在 GitHub for Windows 网站下载，网址为 <http://windows.github.com>。





## 创建local仓库



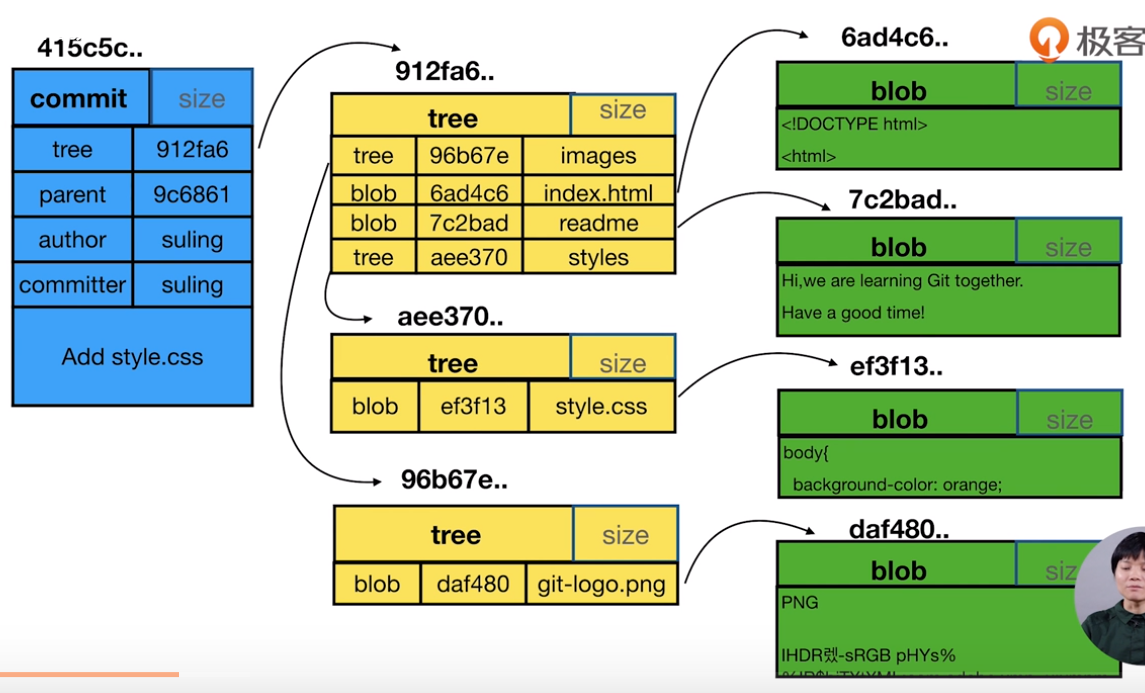
## 对某个仓库单独设置local

设置global后对某个仓库额外设置local属性，local属性优先级更高，比如说账号和邮箱

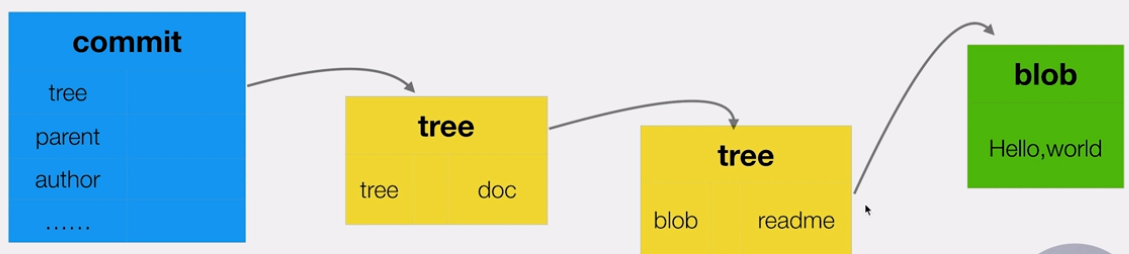
## 往仓库里添加文件



## commit和tree和blob的关系



* 一个commit中一定包含至少一个tree或者blob
* 文件夹是个tree
* 文件是个blob



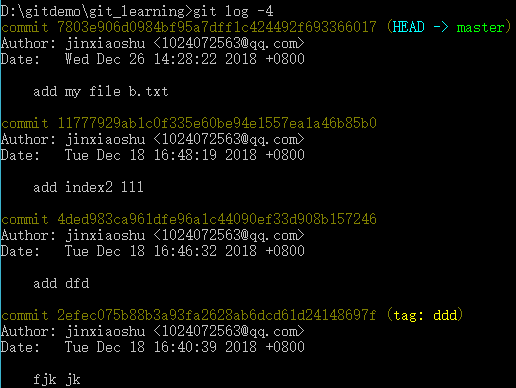
## 分离头指针

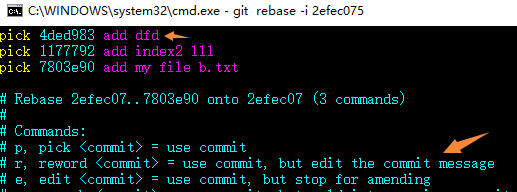
* 开发过程中突然checkout到某个commit，head指向某个具体的commit而不是指向分支，这种状态叫分离头指针
* 然后作了修改，commit，且一直不根据这个checkout的commit创建分支
* 突然又切换到其他分支上去了，这个checkout的commit将不被保留
* 所以如果想commit，请checkout后新建分支

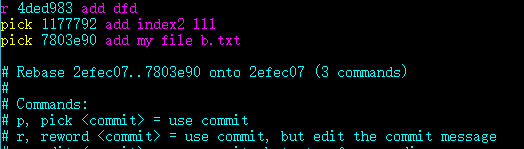
# rebase的简单使用

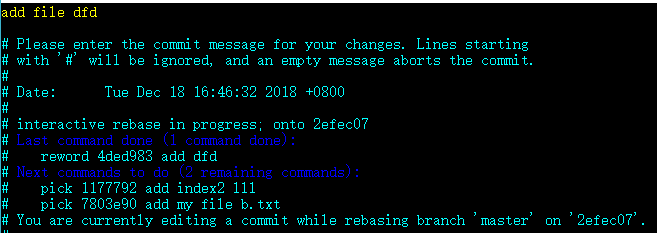
## 修改某次的commit信息

如下图，想将add dfd修改为 add file afd，输入git rebase -i 2efec075b8（父commit）

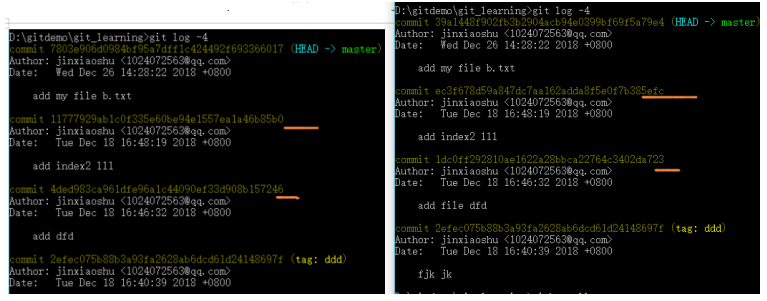






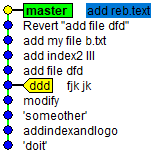
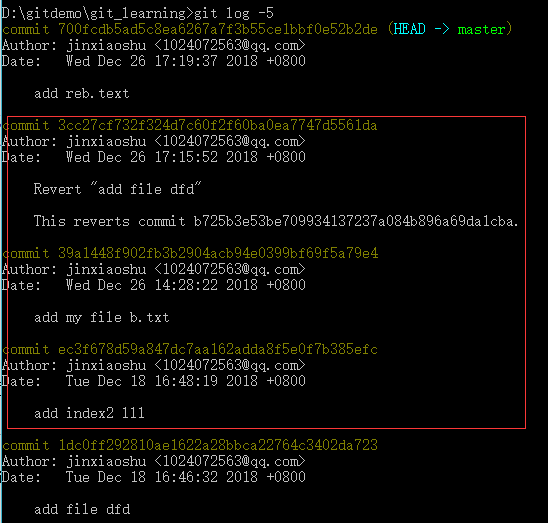


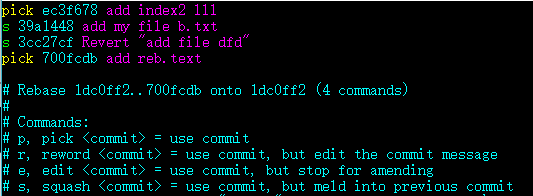
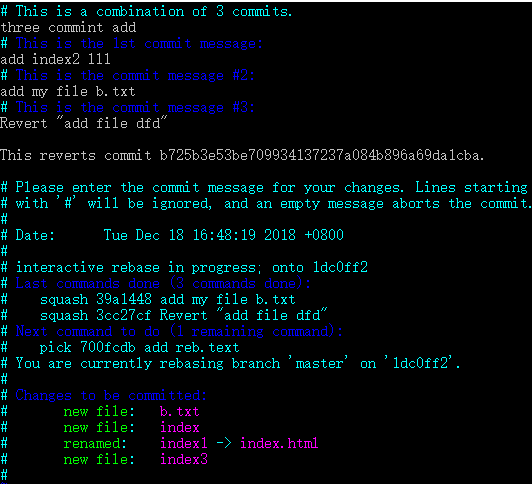
可以看到通过rebase后，相当于在父节点那里将当前分支和原始分支分开了，add dfd和add file dfd都存在，且add dfd网上的commit的id都变了

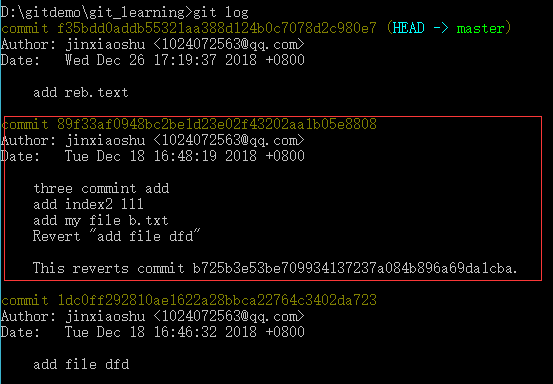
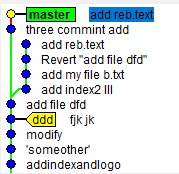


## 连续多个commit合成一个

如下图，想将三次commit合并到ec3上，git rebase -i 1dc0ff29（父commit）

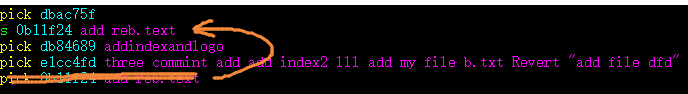
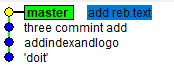
 



## 间隔的多个commit合成一个

如下图，想将ob11合并到dbac上，git rebase -i dbac（父commit,没有父的话需要在编辑中添加）

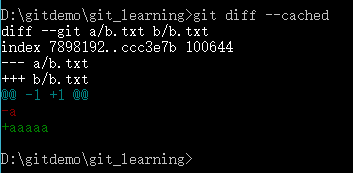




# 比较差异diff

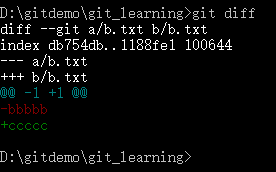
## head和缓存区差异的比较

add后的内容都在缓存区中，通过git diff --cached可以比较head和缓存区的差异



## 工作区和缓存区差异的比较

add前的内容都在工作区中，通过git diff 可以比较工作区和缓存区的差异（即使缓存区已经commit了，缓存区依旧保留上次add的内容），比较多个文件git diff -- index b.txt



## 两个commit、分支、head间的差异

比较两个commit之间的差异

git diff 01fb1 2eb9a8d43

比较最近一次提交和上上次提交

git diff HEAD HEAD~2

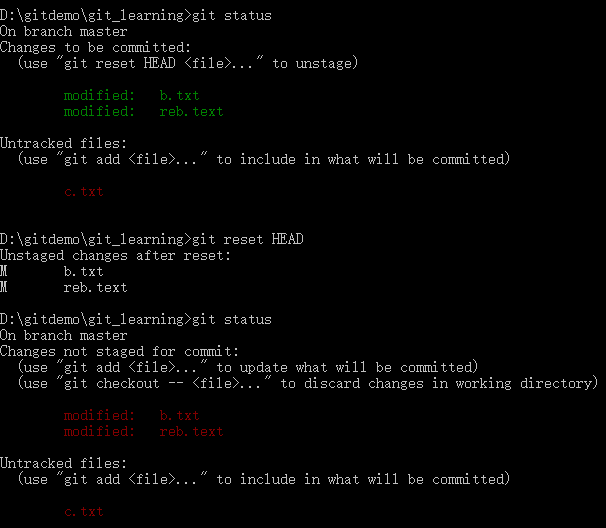
**比较两个分支**

git diff master temp

## 针对性的比较某些文件

在后边加上 -- xxx xxx





# 恢复工作区和暂存区

## 暂存区恢复成和Head一样的

git reset HEAD -- xxx xxx

## 工作区恢复成和暂存区一样的

git checkout -- xxx xxx

## 消除最近几次提交

# 突然来新的紧急bug，stash暂存当前的工作

stash当前修改 git stash

查看现有stash git stash list

移除stash git stash drop stash@{0}

重新应用缓存的stash git stash pop（pop会调用drop或者apply不会调用drop）

## stash的策略

默认情况下，git stash会缓存下列文件：

* 添加到暂存区的修改（staged changes）
* Git跟踪的但并未添加到暂存区的修改（unstaged changes）

但不会缓存一下文件：

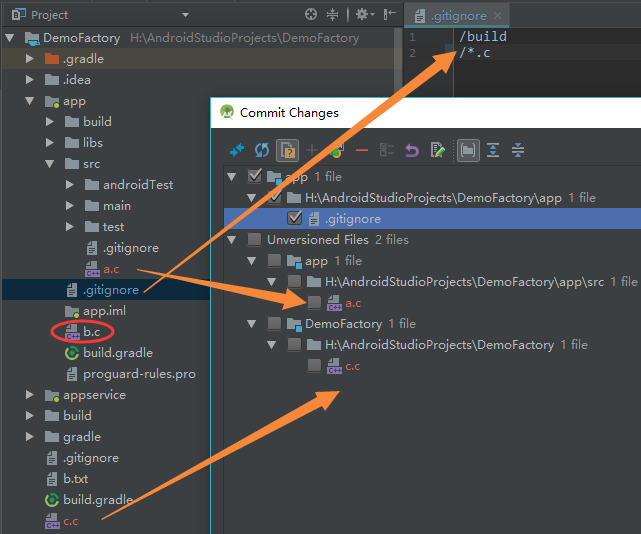
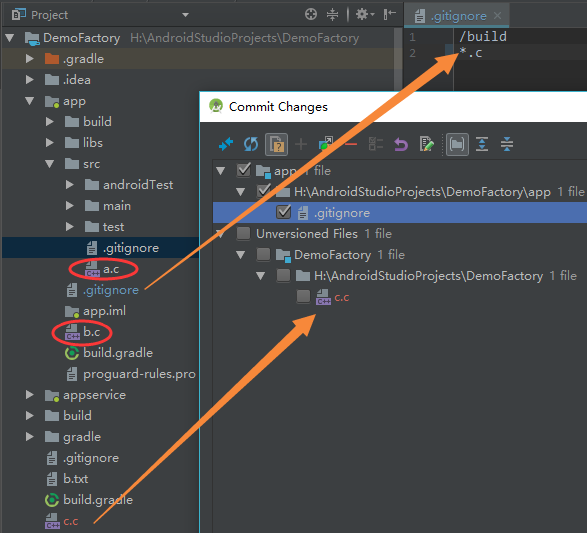
* 在工作目录中新的文件（untracked files）
* 被忽略的文件（ignored files）

git stash命令提供了参数用于缓存上面两种类型的文件。使用-u或者--include-untracked可以stash untracked文件。使用-a或者--all命令可以stash当前目录下的所有修改。

在stash后，工作区和缓存区会恢复成HEAD，在改完bug后，commit bug之后，使用 git stash pop（或apply）恢复到之前的工作状态

# .gitignore忽略

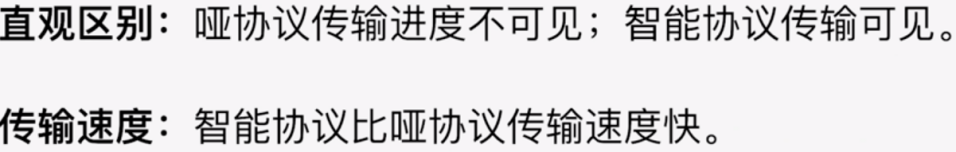
* 未被添加到缓存区的的文件可以被gitignore规则作用
* .gitignore的规则是以该文件.gitignore为根目录来定的（如左图，忽略app根目录下.c结尾的文件）
* .gitignore的规则最大作用范围是根目录以内（如右图，忽略app根目录以内.c结尾的文件）

a.txt 所有 a.txt 文件  
\*.txt 所有 .txt 结尾的 文件  
/\*.txt 根目录下 .txt 结尾的 文件  
  
dirt/ 所有 dirt/目录下 所有 文件  
\*\*/dirt 同上  
\*\*/dirt/\* 同上  
\*\*/dirt/a.txt 所有 dirt/一级目录下 a.txt 文件  
\*\*/dirt/\*\*/a.txt 所有 dirt/目录下 a.txt 文件  
  
/dirt 根目录下 dirt/目录下 所有 文件  
/dirt/ 同上  
dirt/\* 同上  
/dirt/\* 同上  
  
/dirt/a.txt 根目录下 dirt/一级目录下 a.txt 文件  
dirt/a.txt 同上  
dirt/\*\*/a.txt 根目录下 dirt/目录下 a.txt 文件  
  
/dirt/\*.txt 根目录下 dirt/一级目录下 .txt 结尾的 文件  
dirt/\*.txt 同上 (不忽略 dirt/dirt2/a2.txt 和 dirt0/dirt/a2.txt)  
  
dirt/\*\*/dirt2 根目录下 dirt/目录下 dirt2目录下 所有 文件  
  
只需要管理 根目录下 dirt/一级目录下 a.txt文件：  
/dirt/\* (不能写为/dirt/，否则父目录被前面的规则排除掉了)  
!/dirt/a.txt  
  
只需要管理 所有 dirt/一级目录下 所有 a.txt文件：  
\*\*/dirt/\*  
!\*\*/dirt/a.txt（不能写为!/dirt/a.txt，那是管理根目录下的a.txt）  
（使用!\*\*/dirt/\*\*/a.txt也不能管理dirt/dirt2/a.txt，因为dirt2父目录被忽略了）

# 远程仓库



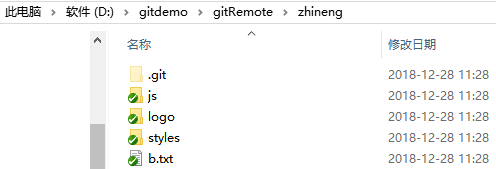


## 克隆本地仓库到一个地方

git clone file://D:\gitdemo\git\_learning/.git zhineng（自动设置本地仓库为克隆仓库的origin）

git clone --bare file://D:\gitdemo\git\_learning/.git zhineng（）

* --bare 表示不带工作区的裸仓库，且没有git信息，不能提交
* file:// 表示智能协议
* zhineng 表示新建仓库的文件夹名

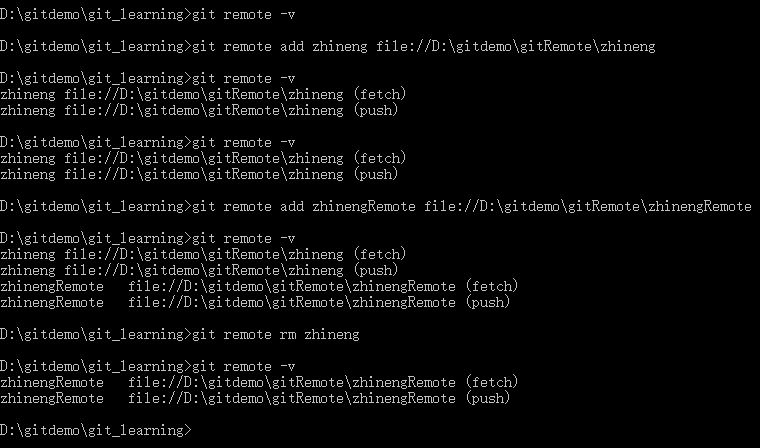


## 查看、添加、删除远程仓库

查看 git remote -v

添加 git remote add zhineng [file://D:\gitdemo\gitRemote\zhineng](file://D:\\gitdemo\\gitRemote\\zhineng)

删除 git remote rm zhineng



## 推送本地仓库的改动到远程

git push zhinengRemote master

git push --set-upstream zhinengRemote master（第一次需要使用--set-upstream来关联）

git push zhinengRemote（想把本分支的推到远程，可以不用写本分支的名称）

git push zhinengRemote --all推送所有分支到远程仓库

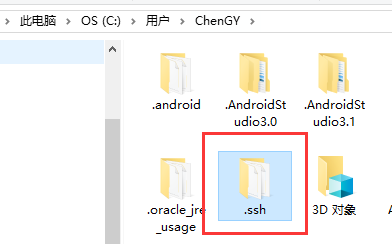
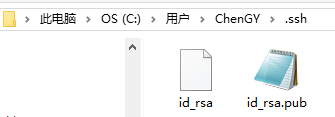
## 拉取远程仓库的改动

git fetch gitssh 下载远程仓库的所有变动，还没和本地进行关联

git pull gitssh [branch] 拉取远程仓库的变化，并与本地分支合并，默认是当前分支

## ssh协议

首先检查用户主目录下是否有.ssh文件夹，并且.ssh文件夹中有没有id\_rsa和id\_rsa.pub这两个文件，如果没有，则使用ssh-keygen -t rsa -b 4096 -C ["your\_email@example.com"](mailto:\"your_email@example.com\")，根据邮箱名创建

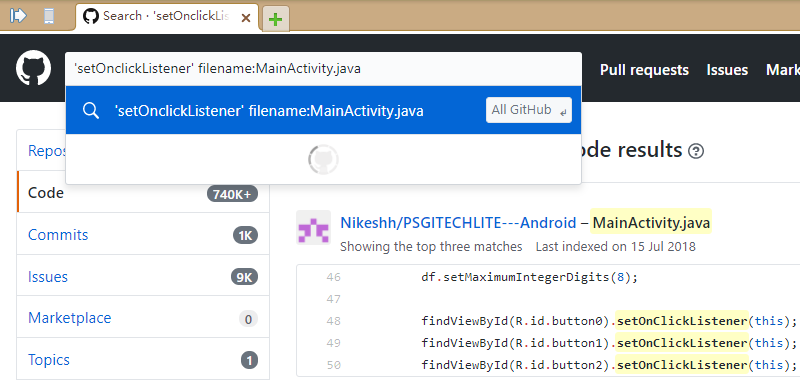
 

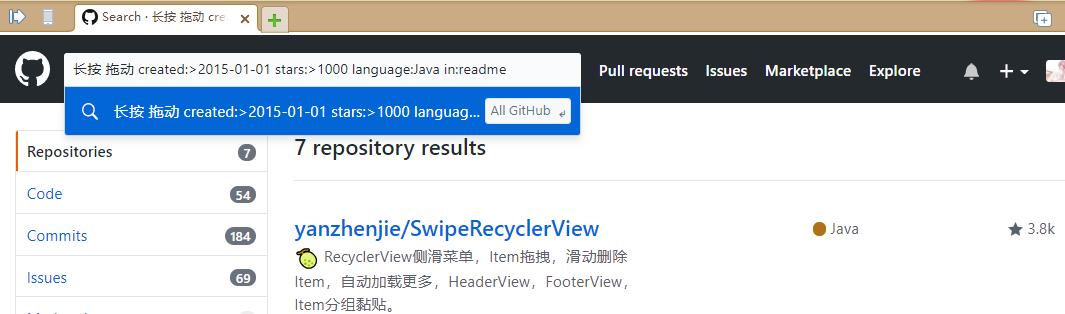
## github上仓库的操作实例

1. 创建好github上的仓库My GitDemoSSH并选择协议MIT，将创建一个LICENSE文件
2. 使用git remote add mygitssh git@github.com:15102391937/DemoGitSSH.git将本地的仓库推拉地址设为github
3. 使用git fetch mygitssh master将github上仓库的master分支的所有内容拉倒本地，创建一个新的分支retmotes/mygitssh/master,该分支没有和本地的master分支关联上
4. 使用git merge mygitssh/master，报错refusing to merge unrelated histories，因为这两个分支没有关联
5. 使用git merge --allow-unrelated-histories mygitssh/master将没有关联的两个分支关联并进行合并
6. 使用git push mygitssh master将本地仓库推送到github上

# GitHub高级搜索技巧

|  |  |
| --- | --- |
| 日期筛选 | created:>2015-01-01 |
| 星星筛选 | stars:>1000 |
| 语言筛选 | language:Java |
| 关键字在某个文件中 | in:readme |
| 某个文件含有某代码片段 | 'setOnclickListener' filename:MainActivity.java |





# 极客时间--------------旧资料

# 什么是Git和GitHub

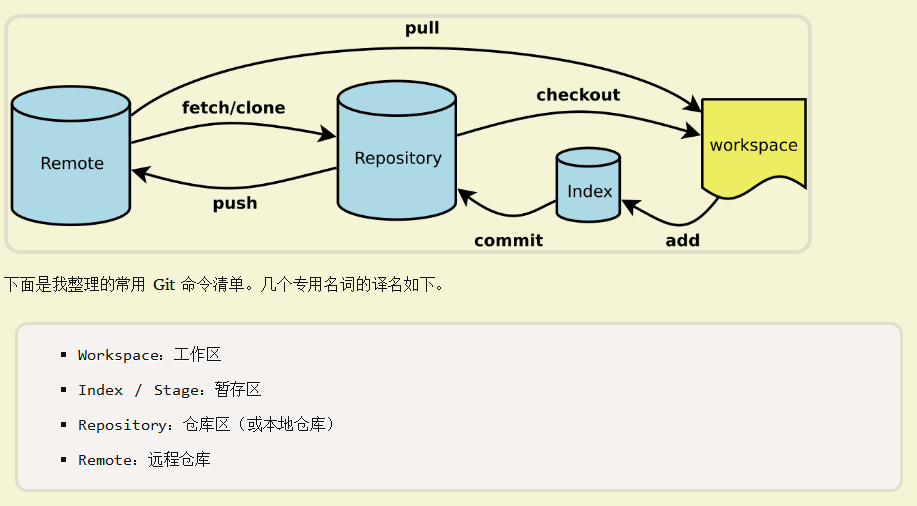
## Git ：

分布式版本控制系统，最初用在Linux上，可以和SVN、CVS等作为同一个概念理解，目前世界上最先进的分布式版本控制系统（没有之一）。

## GitHub：

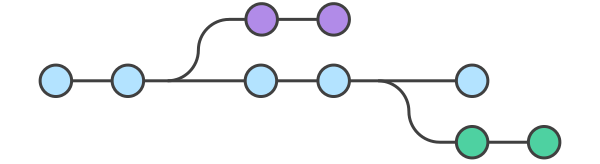
基于Git的项目托管平台，简单的说就是一个网站，通过注册账号后，可以使用Git服务，它提供了可以在上面创建资源仓库的Web页面，通过shell命令或者图形用户界面可以和远程项目同步更新，对项目进行管理。还是不太清楚？就是一个项目由多个人一起干，开发中对于项目里的文件，每个人都有可能拿同一文件进行来编辑，为了协同开发，就有了像Git（版本控制）这样的东西。而GitHub呢，就相当于项目放置的平台罢了，不过它里面有很多不同的开源项目，往往是很多人协同开发的， 借助Git来管理，相对于Git本地仓库来说，它是一个远程仓库。

# Git的常用命令图



# 工作流程Flow

Git 作为一个源码管理系统，不可避免涉及到多人协作。协作必须有一个规范的工作流程，让大家有效地合作，使得项目井井有条地发展下去。"工作流程"在英语里，叫做"workflow"或者"flow"，原意是水流，比喻项目像水流那样，顺畅、自然地向前流动，不会发生冲击、对撞、甚至漩涡。



三种广泛使用的工作流程：

* Git flow
* Github flow
* Gitlab flow

这三种工作流程，有一个共同点：都采用"功能驱动式开发"（Feature-driven development，简称FDD）。它指的是，需求是开发的起点，先有需求再有功能分支（feature branch）或者补丁分支（hotfix branch）。完成开发后，该分支就合并到主分支，然后被删除。

## Git flow

最早诞生、并得到广泛采用的一种工作流程，就是Git flow 。

### 特点

它最主要的特点有两个。

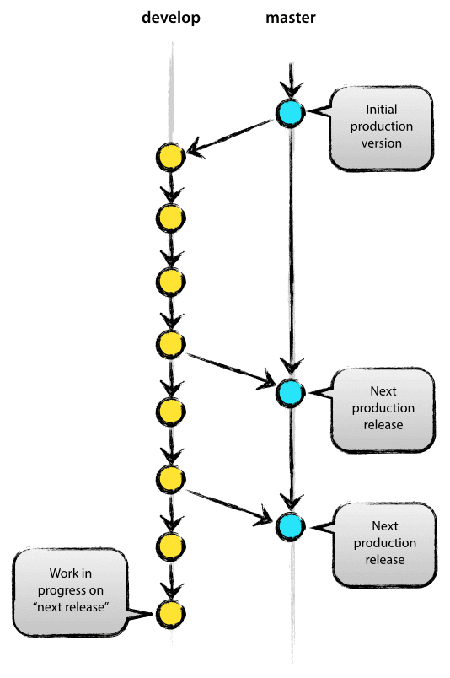
1. 项目存在两个长期分支。

* 主分支master（用于存放对外发布的版本，任何时候在这个分支拿到的，都是稳定的分布版）
* 开发分支develop（用于日常开发，存放最新的开发版）

1. 项目存在三种短期分支。

* 功能分支（feature branch）
* 补丁分支（hotfix branch）
* 预发分支（release branch）

一旦完成开发，它们就会被合并进develop或master，然后被删除。



### 评价

Git flow的优点是清晰可控，缺点是相对复杂，需要同时维护两个长期分支。大多数工具都将master当作默认分支，可是开发是在develop分支进行的，这导致经常要切换分支，非常烦人。

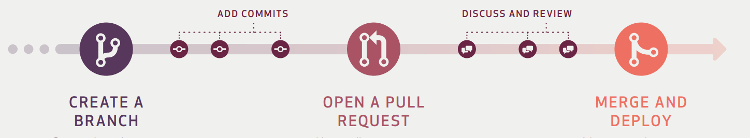
更大问题在于，这个模式是基于"版本发布"的，目标是一段时间以后产出一个新版本。但是，很多网站项目是"持续发布"，代码一有变动，就部署一次。这时，master分支和develop分支的差别不大，没必要维护两个长期分支。

## Github flow

Github flow 是Git flow的简化版，专门配合"持续发布"。它是 Github.com 使用的工作流程。

### 流程

它只有一个长期分支，就是master，因此用起来非常简单。官方推荐的流程如下。



1. 根据需求，从master拉出新分支，不区分功能分支或补丁分支。
2. 新分支开发完成后，或者需要讨论的时候，就向master发起一个pull request（简称PR）。
3. Pull Request既是一个通知，让别人注意到你的请求，又是一种对话机制，大家一起评审和讨论你的代码。对话过程中，你还可以不断提交代码。
4. 你的Pull Request被接受，合并进master，重新部署后，原来你拉出来的那个分支就被删除。（先部署再合并也可。）

## 评价

Github flow 的最大优点就是简单，对于"持续发布"的产品，可以说是最合适的流程。问题在于它的假设：master分支的更新与产品的发布是一致的。也就是说，master分支的最新代码，默认就是当前的线上代码。

可是，有些时候并非如此，代码合并进入master分支，并不代表它就能立刻发布。比如，苹果商店的APP提交审核以后，等一段时间才能上架。这时，如果还有新的代码提交，master分支就会与刚发布的版本不一致。另一个例子是，有些公司有发布窗口，只有指定时间才能发布，这也会导致线上版本落后于master分支。

上面这种情况，只有master一个主分支就不够用了。通常，你不得不在master分支以外，另外新建一个production分支跟踪线上版本。

## Gitlab flow

Gitlab flow 是 Git flow 与 Github flow 的综合。它吸取了两者的优点，既有适应不同开发环境的弹性，又有单一主分支的简单和便利。它是 Gitlab.com 推荐的做法。

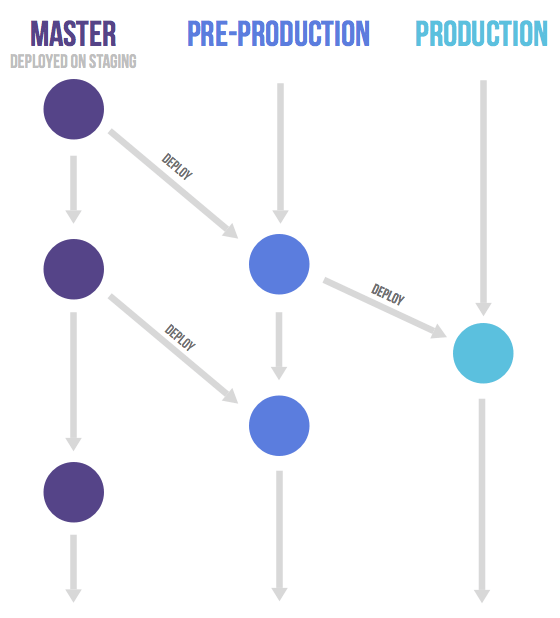
### 上游优先

Gitlab flow 的最大原则叫做"上游优先"（upsteam first），即只存在一个主分支master，它是所有其他分支的"上游"。只有上游分支采纳的代码变化，才能应用到其他分支。Chromium项目就是一个例子，它明确规定，上游分支依次为：

* Linus Torvalds的分支
* 子系统（比如netdev）的分支
* 设备厂商（比如三星）的分支

Gitlab flow 分成两种情况，适应不同的开发流程。

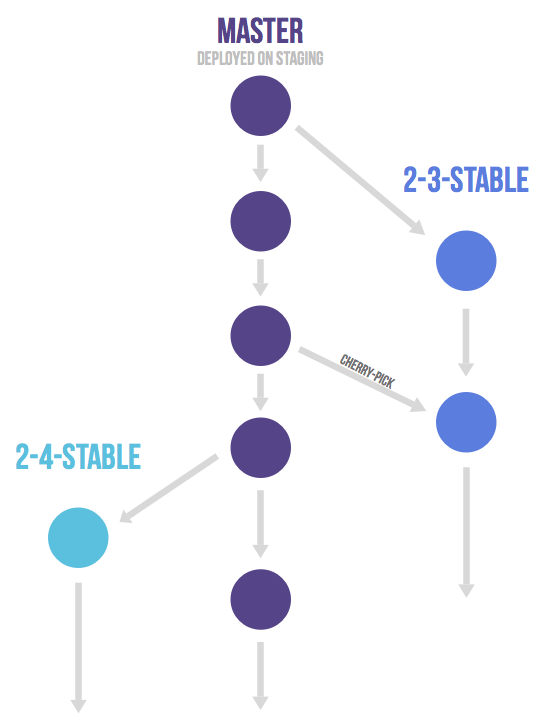
### 持续发布



对于"持续发布"的项目，它建议在master分支以外，再建立不同的环境分支。比如，"开发环境"的分支是master，"预发环境"的分支是pre-production，"生产环境"的分支是production。

开发分支是预发分支的"上游"，预发分支又是生产分支的"上游"。代码的变化，必须由"上游"向"下游"发展。比如，生产环境出现了bug，这时就要新建一个功能分支，先把它合并到master，确认没有问题，再cherry-pick到pre-production，这一步也没有问题，才进入production。只有紧急情况，才允许跳过上游，直接合并到下游分支。

### 版本发布

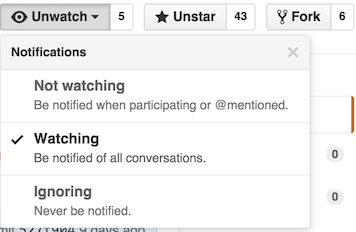


对于"版本发布"的项目，建议的做法是每一个稳定版本，都要从master分支拉出一个分支，比如2-3-stable、2-4-stable等等。以后，只有修补bug，才允许将代码合并到这些分支，并且此时要更新小版本号。

# GitHub中的watch、star、fork

## watch

watch 翻译过来可以称之为观察，点击 watch 可以看到如下的列表。



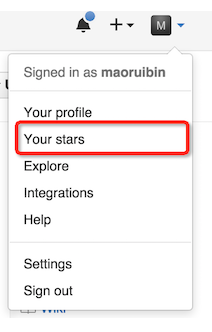
对于别人的项目，默认自己都处于 Not watching 的状态，当你选择 Watching，表示你以后会关注这个项目的所有动态，这个项目以后只要发生变动，如被别人提交了 pull request、被别人发起了issue等等情况，你都会在自己的个人通知中心，收到一条通知消息，如果你设置了个人邮箱，那么你的邮箱也可能收到相应的邮件。

如下，我 watching 了开源项目android-cn/android-discuss，那么以后任何人只要在这个项目下提交了 issue 或者在 issue 下面有任何留言，我的通知中心就会通知我。如果你配置了邮箱，你还可能会因此不断的收到通知邮件。如果你不想接受这个项目的所有通知，那么点击 Not Watching 即可。

## star

star 翻译过来是星，但这个翻译没任何具体意义，这里解释为`关注`或者`点赞`更合适，当你点击 star，表示你喜欢这个项目或者通俗点，可以把他理解成朋友圈的点赞吧，表示对这个项目的支持。

不过相比朋友圈的点赞，github 里面会有一个列表，专门收集了你所有 star 过的项目，点击 github 个人头像，可以看到 your stars 的条目，点击就可以查看你 star 过的所有项目了，如下图所示



不过，star 列表很容易出现这样的问题：如果你平时比较爱逛 Github， star 的项目过多后怎么快速查找自己想要的项目怎么办？

这时，如果 github 可以提供一个分类功能该多好，就像微信公众号文章的收藏，你在点击微信菜单中的收藏后，微信会提示你对收藏的文章设置 tag。这样设置的好处是，以后在微信收藏列表中查找项目时，可以根设置的 tag 归类查找，这样查找效率会提高不少。但是不知道 github 的产品经理是怎么想的，github 本身没有这个功能。

## fork

当选择 fork，相当于你自己有了一份原项目的拷贝，当然这个拷贝只是针对当时的项目文件，如果后续原项目文件发生改变，你必须通过其他的方式去同步。

一般来说，我们不需要使用 fork 这个功能，除非有一些项目，可能存在 bug 或者可以继续优化的地方，你想帮助原项目作者去完善这个项目或者单纯的想在原来项目基础上己维护一个属于自己项目（比如我 fork 的 AndroidWeekly 客户端，那么你可以 fork 一份项目下来，然后自己对这个项目进行修改完善，当你觉得项目没问题了，你就可以尝试发起 pull request 给原项目作者了。然后就静静等待他的 merge 邮件通知了。

我看到很多人错误的在使用 fork。很多人把 fork 当成了收藏一样的功能，包括一开始使用 github 的我，每次看到一个好的项目就先 fork，因为这样，就可以我的 repository(仓库)列表下查看 fork 的项目了。其实你完全可以使用 star 来达到这个目的。

## 使用建议

1、对于一些可能会经常发生变化的会不定期更新的好项目 多使用 watch.

2、喜欢一个项目就 star 它吧

3、修改开源项目就使用 fork，这样你就可以在原项目的基础上，对项目进行修改提交，现在你是这个项目的主人啦~

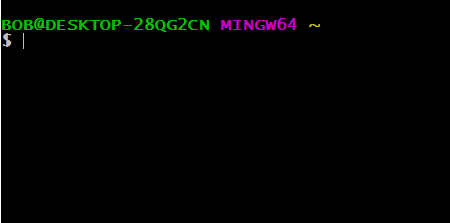
## 小细节

有些时候，你看到一个项目的 star 数有很多，你就想知道到底都有那些人 star 了这个项目，或者 fork 了这个项目，只要点击 star 傍边的数字，就可以查看有哪些人 star 了这个项目。是不是有点意思，现在你就可以去试试，watch、fork上面的数字都是可以点击的，道理一样。

# AS中初次使用Git

## 安装Git

安装完成之后，在开始菜单中找到Git-->Git Bash，如果出现类似下面的对话框的话就证明安装成功了。



安装成功之后，需要配置一下你的信息，直接在上面的命令行中输入以下命令：

$ git config --global user.name "Your Name"

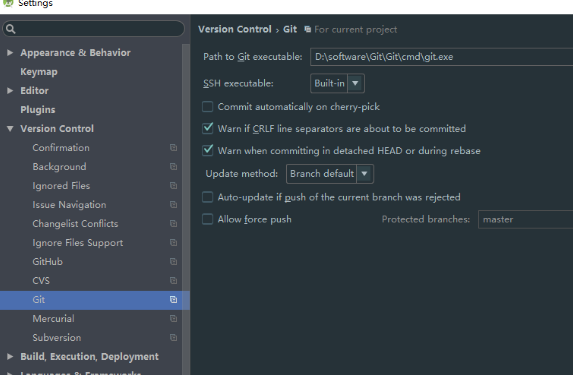
$ git config --global user.email "email@example.com"

注意：

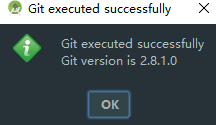
1. 将上方的Your Name换成你自己的名字，随意起都行。
2. 将上方的email@example.com换成你自己的邮箱。
3. 讲一下git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

## Android Studio配置Git

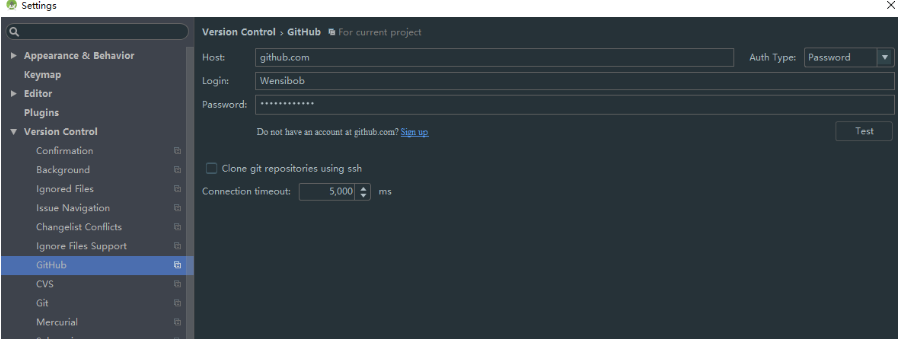
打开Settings，接着点击Version Control --> Git在Path To Git Executable上输入Git的存放位置，如下图：



之后点击旁边的Test按钮，如果出现如下的成功提示，说明Git配置成功。

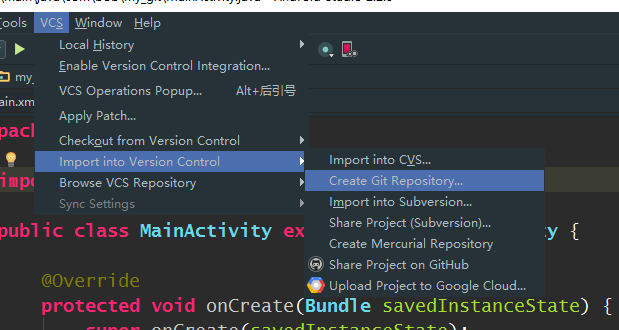


接着按照下图配置你的Github账号密码，输入完成之后点击Test按钮，如果账号密码都正确的话就会提示成功，那么就可以继续往下走了：

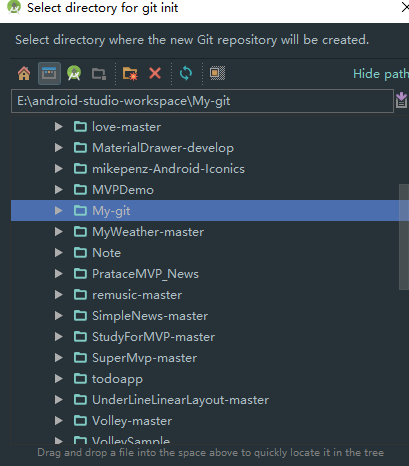


## 创建本地仓库

不管你是刚刚新建的项目，亦或是已经写好的项目，进行的操作都是一样的。这里我以新建一个名为MyGit的项目为例子，对于一个已经新建好的项目，找到菜单栏上的VCS，按照下图进行操作：



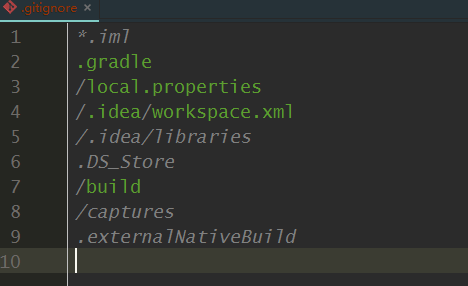
选择项目的根目录为git初始化的目录：



初始化之后你会发现原本文件的文件名都是**白色**的，现在变成了**棕色**，这表示文件已经被git跟踪了，但是并没有添加到仓库中：

## 忽略文件

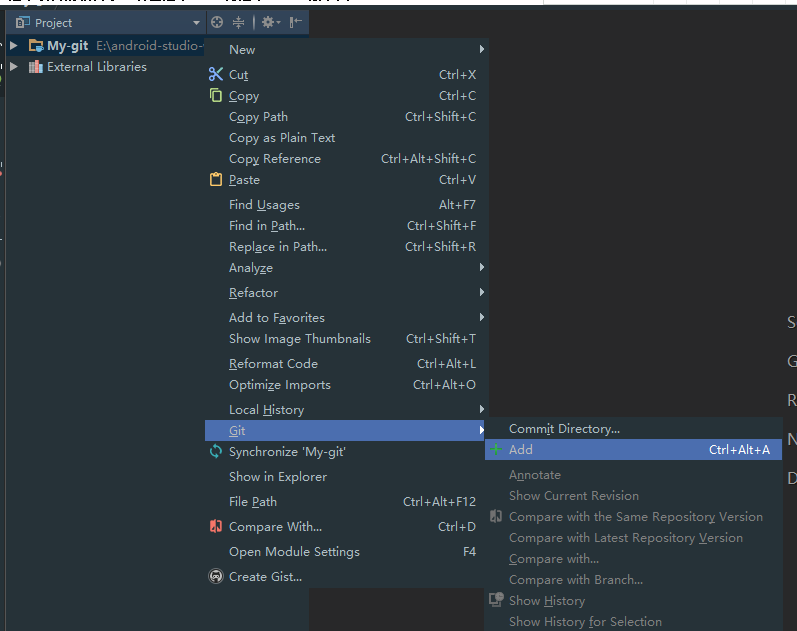
我们知道，在git初始化一个仓库的时候会自动生成一个.gitignore文件，这个文件用来忽略那些不用加入到仓库的文件，在我们这个工程中总共生成了两个.gitignore文件，分别是在项目根目录下，以及在app文件夹下。我们可以对这些文件进行编辑，表示我们需要忽略哪些文件，但是一般情况下，我们选择默认就好，除非你有需要就进行适当的修改：



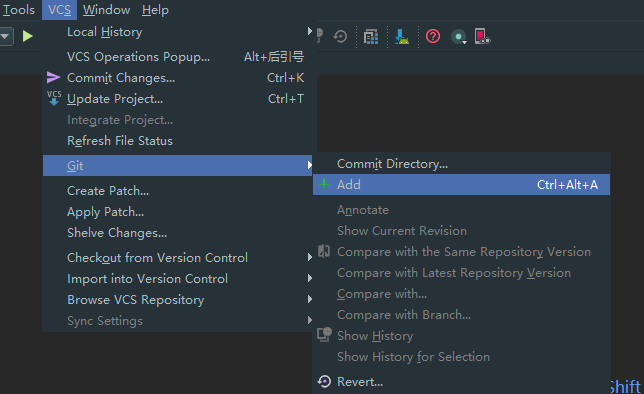
## 添加文件

添加文件就如git命令中的git add，在Android Studio中add的方式有四种，但是都是大同小异，让我来一一举例：

1.选中项目的根目录，右键选中Git，再选中Add，如下图：

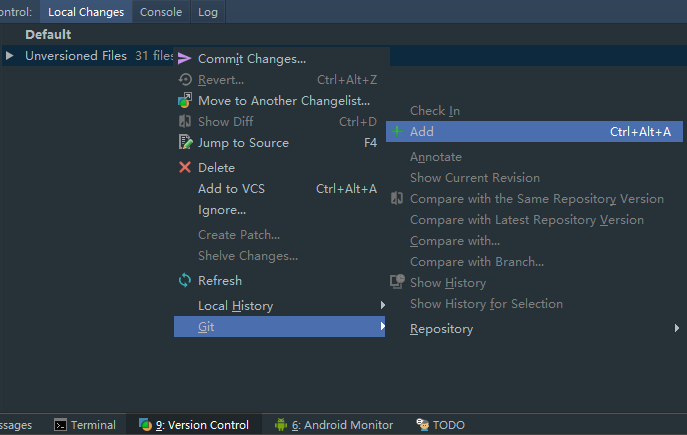
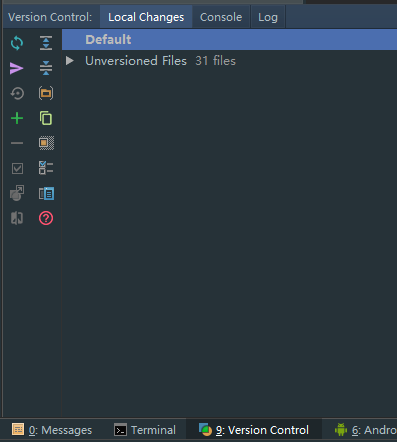


1. 选中项目根目录，点击菜单栏中的VCS菜单，选中Git-->Add，如下图：

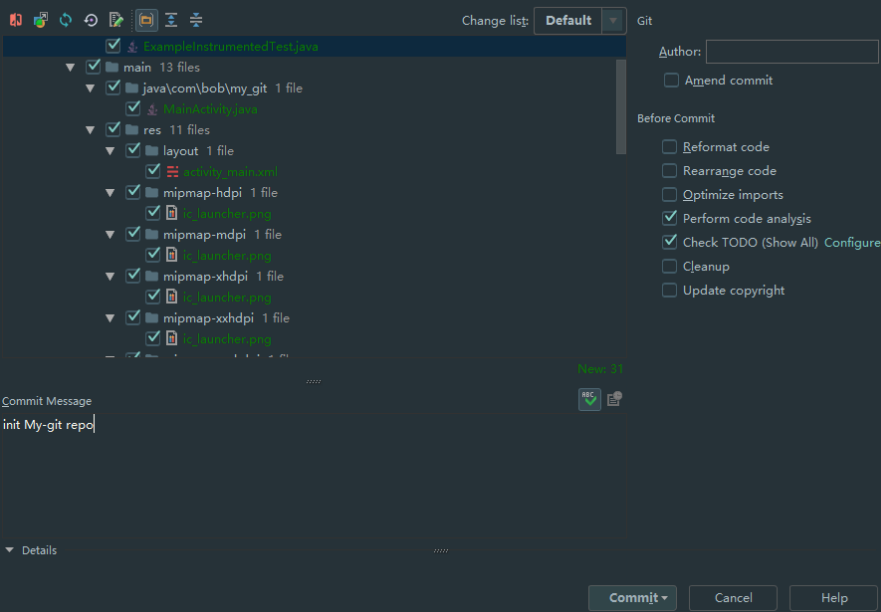


3.使用快捷键Alt+9，或者点击屏幕下方的Version Control工具按钮

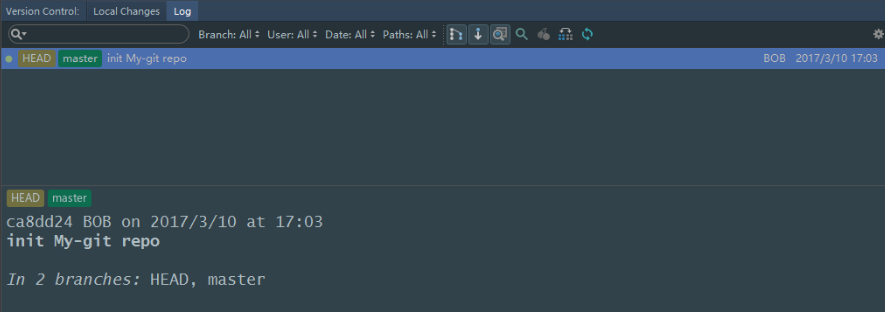
打开版本控制的窗口，可以看到应该是如下图的样子，该窗口有两个下拉栏，一个是Default，用来记录已经添加的文件，另一个是Unversioned files，用来记录已经被跟踪但是未添加的文件。右键Unversioned files，选中Git-->Add，如下图：



add之后文件名变成了**绿色**，这是代表已经添加进仓库为文件，接下来就可以commit文件了，使用快捷键Ctrl+K或者选中工程根目录右键Git-->Commit Directory可以调出commit窗口，如下所示，在其中选择你想要提交的文件，填写提交的信息，在Author文本框中可以填写提交此次提交的操作者名字，如果不填写的话，就会默认是之前配置Github账号的用户名。可以看到，提交之后文件名重新变回了熟悉的**白色**。

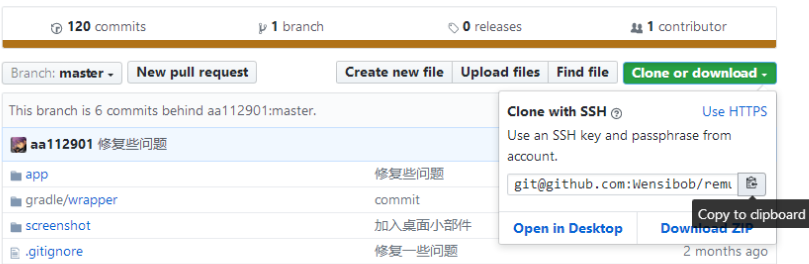


提交成功之后，你可以使用快捷键Alt+9，或者点击屏幕下方的Version Control工具按钮，切换到Log菜单查看Log日志，如下图所示：

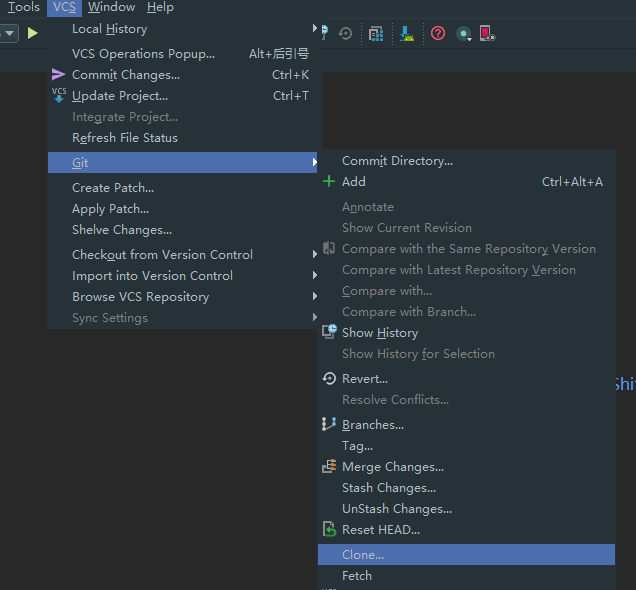


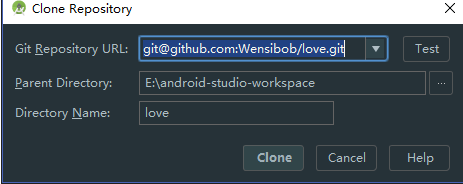
## 如何clone项目

使用Git clone项目到本地中是很简单的，在Android Studio中也是如此，首先找到你喜欢的项目，fork到你自己的仓库之后，点击Clone or Download按钮，复制地址，如下图所示：



接下来回到Android Studio，按照下图的操作可以打开clone的对话框，在地址栏中粘贴刚才复制的地址，点击Test按钮，测试是否可以通过，如果成功，那么久可以点击Clone导入项目了。



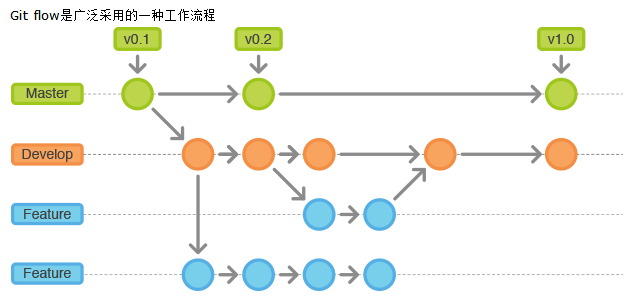


# 项目中使用Git

回到我们最开始讲的MyGit项目，首先我们需要模仿一下在日常开发中使用Git的情形，例如我们经常会在分支上进行工作，所以熟练地掌握分支工作的流程以

## Git flow

以下举得例子将会围绕Git的一种工作模式，即：Git Flow



## 特点

它最主要的特点有两个。

1. 项目存在两个长期分支。

* 主分支master（用于存放对外发布的版本，任何时候在这个分支拿到的，都是稳定的分布版）
* 开发分支develop（用于日常开发，存放最新的开发版）

1. 项目存在三种短期分支。

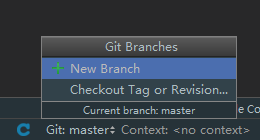
* 功能分支（feature branch）
* 补丁分支（hotfix branch）
* 预发分支（release branch）

## 新增功能1——显示Hello World

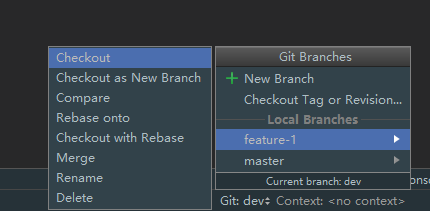
正如我们在介绍Git flow介绍的，master分支只是用于产品的发布，在平时的开发中是不会使用它的，而只会使用dev分支，但是如果我们有了新的功能，一般是会在dev分支中在创建一条该功能的分支，所以我们应该这样做。

### 创建dev、feature-1分支，并且我们需要转到feature-1分支上

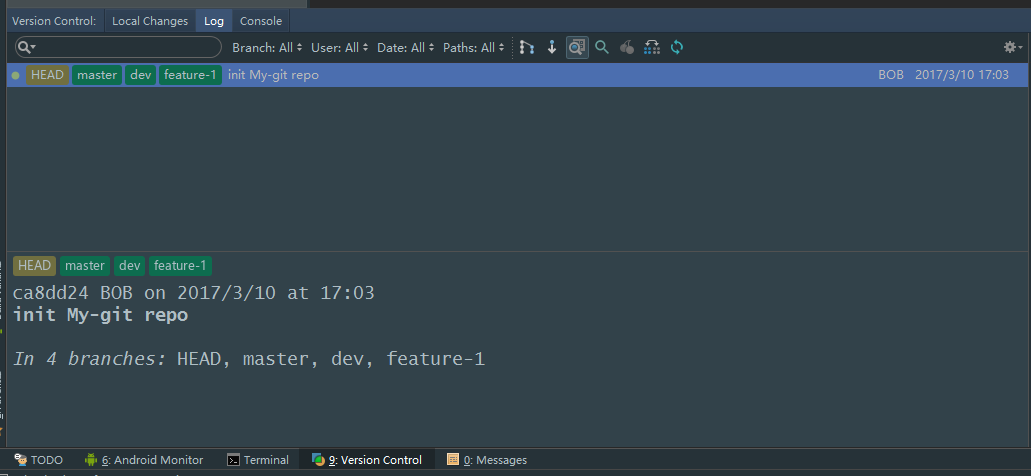
在Android Studio中，我们可以很方便的管理分支，在主界面的右下角，点击Git可以出现当前的分支，默认为master，我们选中New Branch，如下图所示：



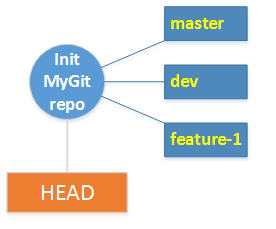
在弹出的对话框中我们输入feature-1，点击OK，这样我们不仅新建了feature-1分支，并且正处于该分支中，接下来按照同样的方法创建dev分支，如果不出意外的话，我们现在应该是处于dev分支上，但是因为我们现在要开发功能1，所以必须转换到feature-1分支上，按照下图的操作，我们能够回到feature-1分支上。



接下来打开Log，我们应该能够看到如下的情景：



可以看到我们现在有三个分支：master、dev、feature-1，但是AS提示我们应该有四条分支，其实HEAD就是当前活跃分支的游标。形象的记忆就是：你现在在哪儿，HEAD 就指向哪儿，所以 Git 才知道你在那儿！不过 HEAD 并非只能指向分支的最顶端，实际上它可以指向任何一个节点，它就是 Git 内部用来追踪当前位置的标记。我们可以使用下面的图来演示当前分支的情况：



### 完成功能1

/\*\*

\* feature-1

\* display HelloWorld

\* @param view

\*/

public void feature1(View view) {

if (btn\_feature\_1.getText().toString().equals("功能1")) {

btn\_feature\_1.setText(R.string.feature\_1\_dis);

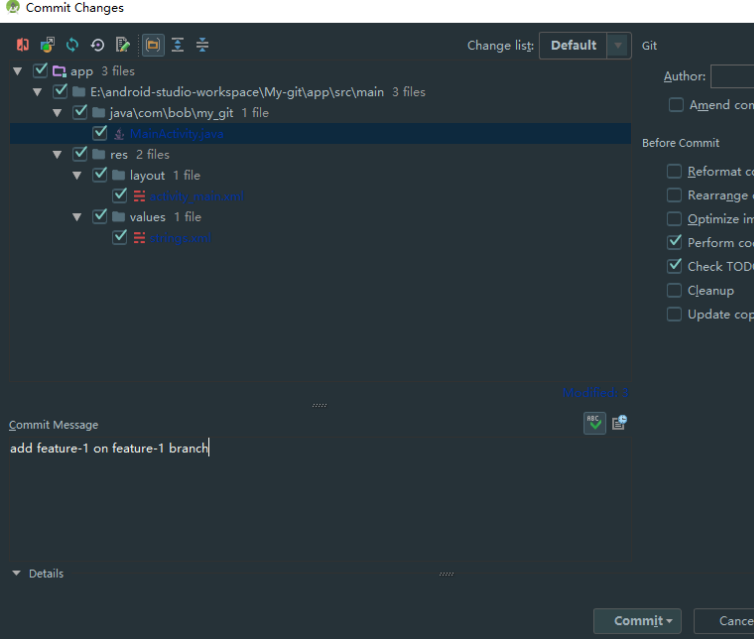
} else {

btn\_feature\_1.setText(R.string.feature\_1);

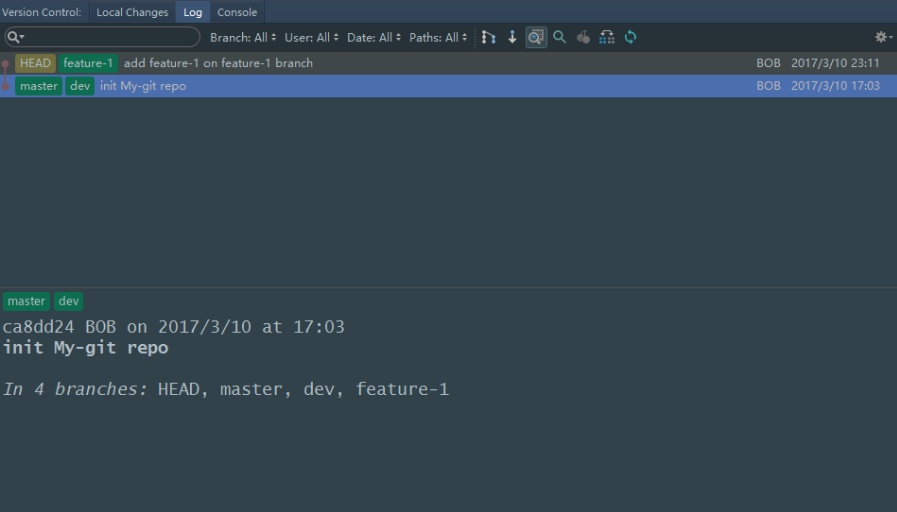
}

}

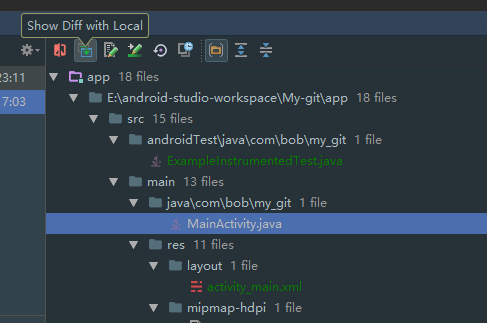
好了，功能1编写完成，那么就提交吧！老方法：Ctrl+K进行提交，按照下图填写提交信息，每一次的提交信息最好能够详细并且格式规范，这样以后再查看Log的时候就会比较方便。



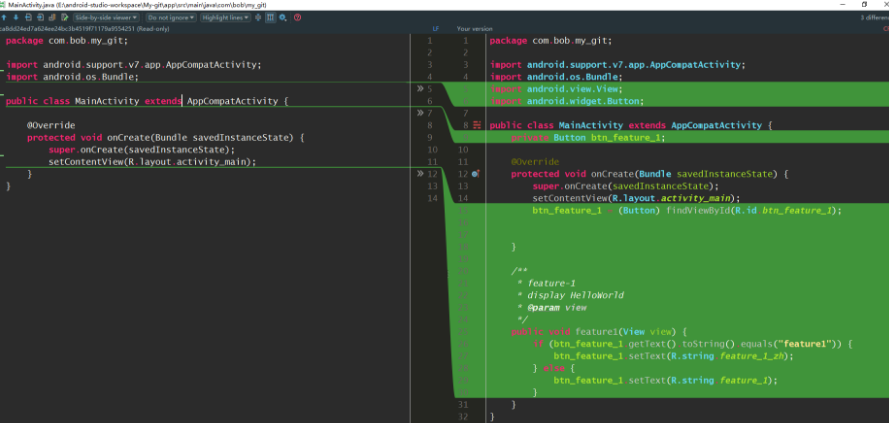
点击Commit按钮提交完毕之后，可以看到现在的log图变成了下图：



我们可以打开Log图的右侧，他列出了目前正在被Git跟踪的所有文件，我们选中MainActivity.java，点击上边的第二个按钮Show Diff(显示差异)：

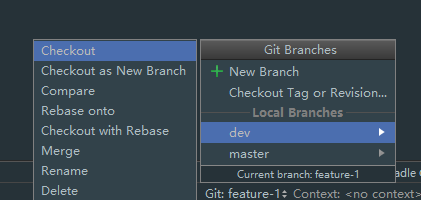


在弹出的界面中可以显示该文件的当前版本和master分支中的该文件的差异，我们可以使用Esc快捷键退出该界面，如下图所示：

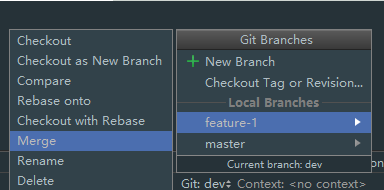


### 将feature-1分支合并到dev分支上

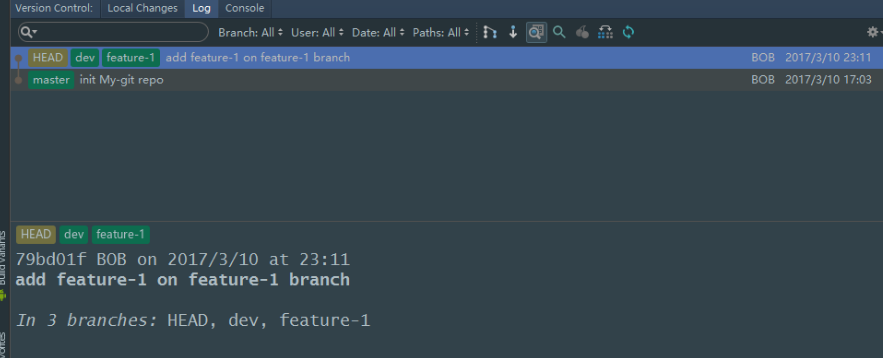
如下图所示，选中dev分支并选择checkout：



你会发现现在代码回到了最开始的状态，现在选中feature-1分支并选中merge，准备将feature-1分支合并到dev分支上：

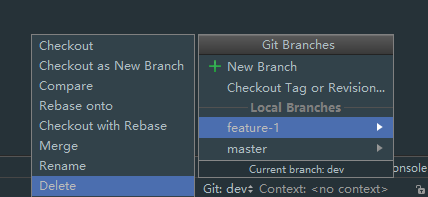


再来重新看一下Log图，他长下面这样，可以看到现在feature-1分支已经与dev分支合并，并且现在他们是处于同一状态的：



### 删除feature-1分支

最后的最后，我们需要删除已经完成任务的feature-1分支，以免分支过多管理混乱。



## 新增功能2——显示Hello Android

同样的道理，我们可以按照如下的步骤这样做：

### 迁出dev分支，并且新建分支feature-2之后迁出feature-2分支

具体的步骤我就不再演示了，跟上面的是一样的。

### 完成功能2

/\*\*

\* feature-2

\* display HelloAndroid

\* @param view

\*/

public void feature2(View view) {

if (btn\_feature\_2.getText().toString().equals("功能2")) {

btn\_feature\_2.setText(R.string.feature\_2\_dis);

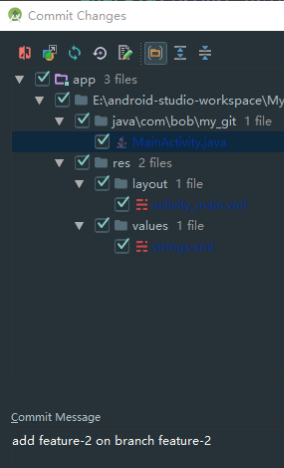
} else {

btn\_feature\_2.setText(R.string.feature\_2);

}

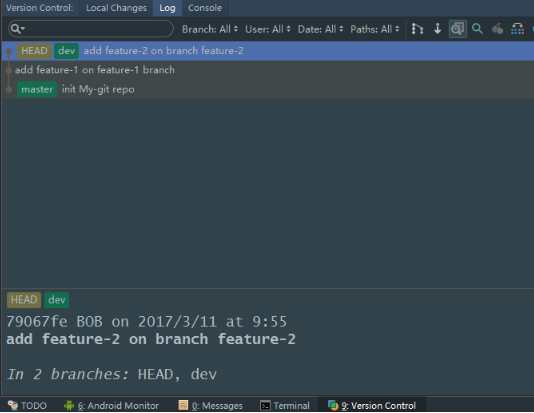
}

使用快捷键Ctrl+K提交feature-2的修改，并填写提交信息。



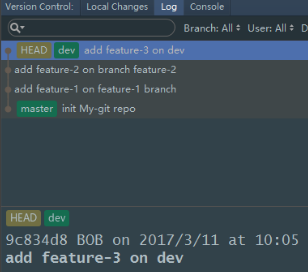
### 将feature-2分支合并到dev分支上

切换回dev分支并且合并feature-2分支之后删除feature-2分支。如果不出错的话，你的Log应该是下面这样的：



## 增加功能3——Hello Java

同样的道理，这里我就不再一一演示，如果按照上面的步骤做的话，最后的Log应该是这样的：

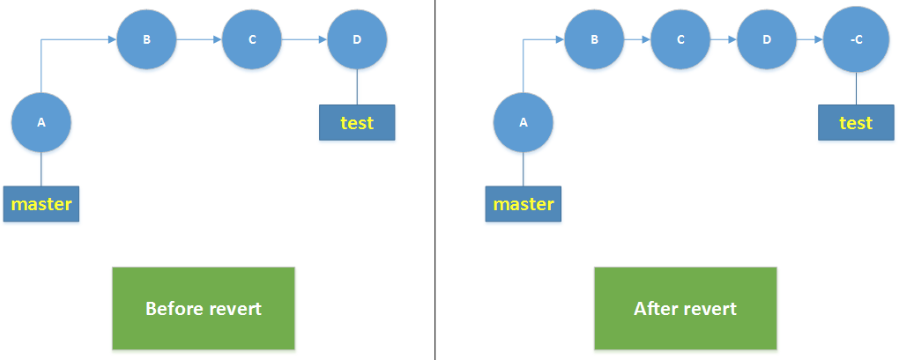


## 删除功能3

重点来了，老板说了，新开发的功能3不喜欢，需要删除功能3，咋办呢？这里就需要讲到Git的回退了，在Android Studio中提供了两种回退的方式：Git revert以及Git reset。

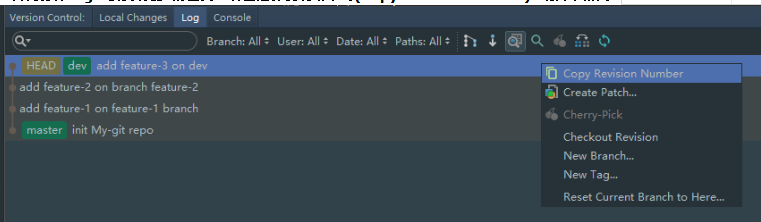
### Git revert

Git revert可以将版本回退到上一步，但是会新增一个提交，他的流程就像下面这幅图一样：



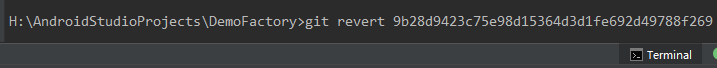
#### 复制哈希码

首先打开Log，找到功能3的提交，右键选择复制哈希码(Copy Revision Number)，如下图所示：



#### Terminal 输入 git revert

打开Android Studio的终端Terminal，他就在Version Control的旁边，之后输入以下命令按回车键：

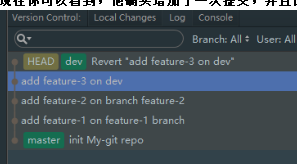


那一段很复杂的数字字母就是我们刚才复制的哈希码

#### 保存revert

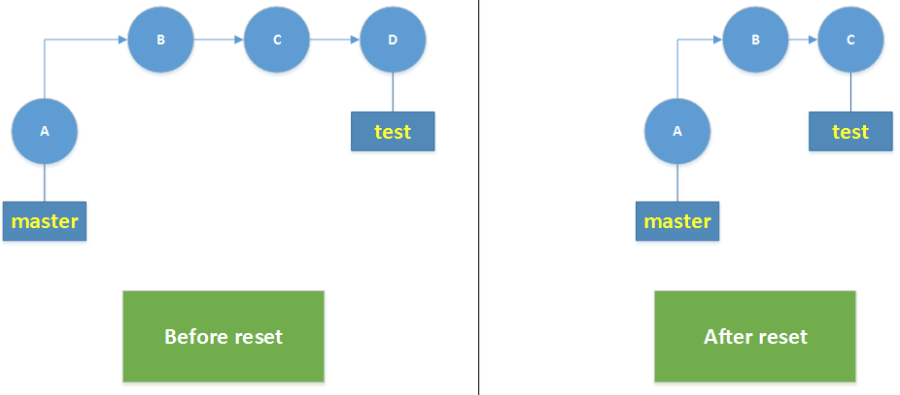
之后终端会列出此次提交的具体信息，如果确认要回退，请输入q保存此次操作并且退出会话。这时你的界面已经回到上次的版本，然后再次ctrl + k 提交，输入提交信息 revert “add frature-3 on dev”

现在你可以看到，他确实增加了一次提交，并且回到了上一版的内容，Log应该是这样的：



### Git reset

相比之下，Git reset就要干脆的多，与Git revert的功能一样，它也可以将代码恢复到上一个版本，但是不会新增一次提交，他的流程如下：



因为我们需要删除功能3，并且让Log看起来并没有revert的这一次提交，所以我们应该在dev分支上后退两步，确实是这样的对吧！

注意：该功能是接着上面的revert来的

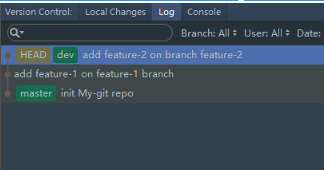


#### HEAD~2

点击菜单栏上的VCS-->Git-->Reset HEAD，打开对话框，在To Commit文本框中输入HEAD~2，就像下图这样：

#### 点击Reset

点击Reset按钮之后，你可以发现Log变了，变回原来那个熟悉的画面了：



## master分支上被修改了，使用Rebase

### 同事添加新特性

#### 添加feature-1和feature-4

突然你发现你的同事在master分支上提交了两次，分别是增加了功能1和功能4，但是其中的功能1很显然HelloWorld被写成了WorldHello，例如这样的：

/\*\*

\* feature-1

\* display WorldHello

\* @param view

\*/

public void feature1(View view) {

btn\_feature\_1.setText(R.string.feature\_1\_dis);

}

/\*\*

\* feature-4

\* display Hello

\* @param view

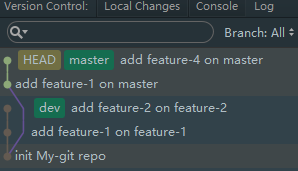
\*/

public void feature4(View view) {

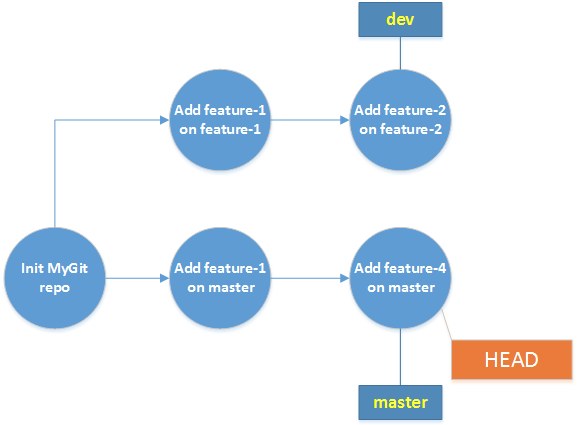
btn\_feature\_4.setText(R.string.feature\_4\_dis);

}

提交更改，之后Log应该是这样的：



#### 演示图

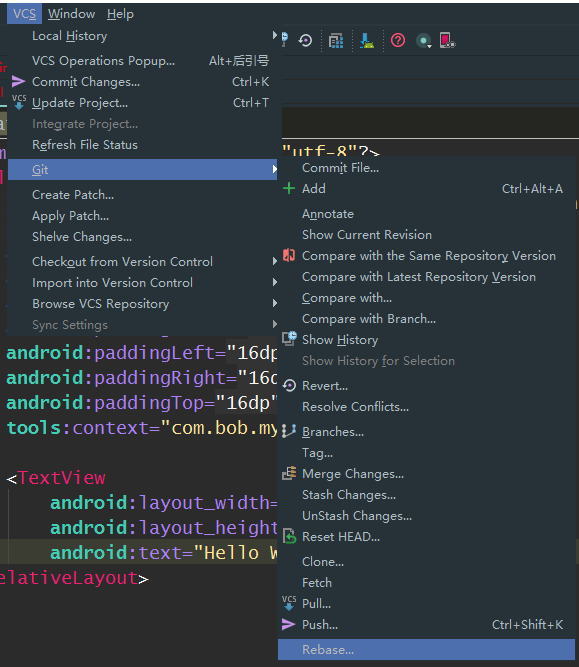


### Rebase

老板说了，master分支只要功能4不需要功能1，而dev分支上的功能1、2全部都要合并到master分支上。那么这个时候我们就可以使用rebase了。

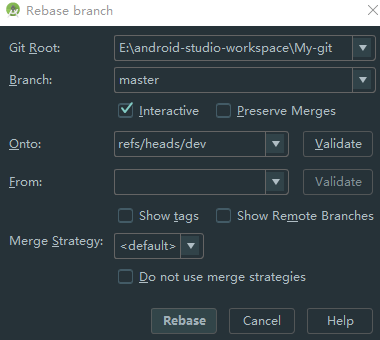
git rebase用于把一个分支的修改合并到当前分支。现在我们切换到master分支，将dev上的做修改加入到master中，所以我们选择rebase，在Android Studio中提供了功能十分强大的rebase。

#### 点击菜单栏上的VCS-->Git-->Rebase，如下图所示：

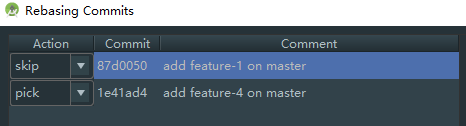


#### 在Onto的下拉栏中选中dev

在弹出的对话框中，我们在Onto的下拉栏中选中dev分支，表示我们需要将master分支rebase到该分支下。

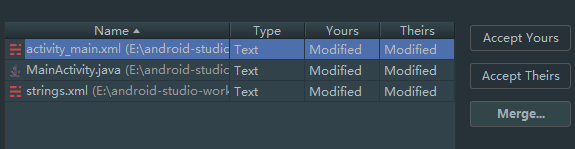


点击Rebase之后，你会发现Android Studio弹出对话框，显示master分支的两次提交，需要我们做出选择，因为我们不需要master分支上的功能1但是需要功能4，所以在功能1的提交上我们选择skip(跳过这个提交)，在提交4上选择pick(挑选此次这个提交)，点击 Start Rebasing

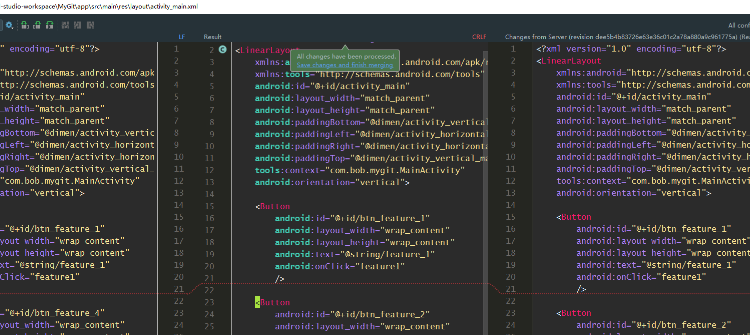


#### 如果有冲突

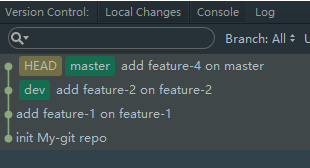
如果有冲突，可以看到又有对话框弹出，此次是让我们对每个文件进行挑选，如下图所示：



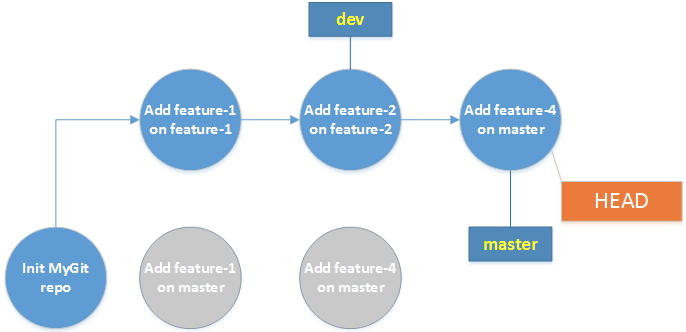
对于每一个文件，你可以选择接受你的那一部分，或者接受另一只分支上的内容，又或者你可以点击Merge对文件进行筛选。我们点击Merge按钮，可以看到有三个文件呈现在屏幕上。其中中间的文件是最后的结果，左边的为当前分支master分支，右边的为dev分支，你还可以发现在修改过的每一行中都存在一个X >>符号，点击X表示不需要这一行的修改，点击>>表示接受这一行的修改，我们甚至还可以像在编辑器中那样复制、粘贴、编辑内容，我们最终作出的选择如下图所示，之后可以点击Apply进行保存，如果你不想保存，那就点击Abort终止此次修改：



对于剩下的两个文件也做相同的处理，之后我们可以看到master分支已经有了dev分支的功能1和功能2和自身的功能4，并且去掉了自己之前的功能1，可以看一下Log，如下所示：



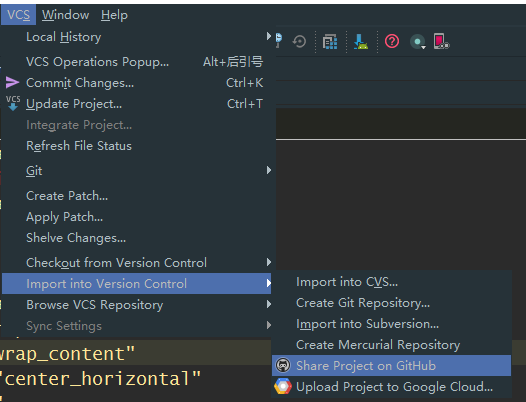
#### 演示图



## 推送到远程仓库

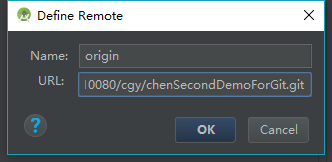
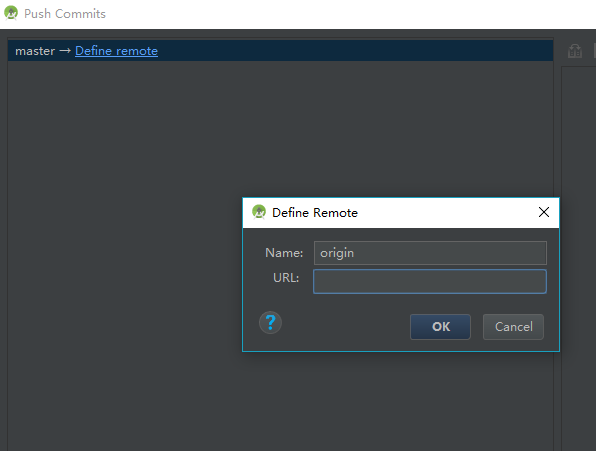
### 添加到GitHub

推送很简单，你可以导航到菜单栏VCS-->Import Into Version Control -->Share Project on Github，之后你就可以在你的Github上面看到这一次的推送了，如下图所示：



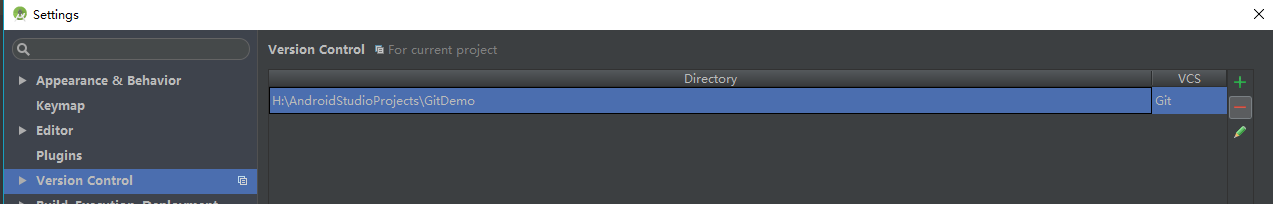
### 添加到自定义url地址下（eg，gitLab）

直接操作推送，由于还没有远端，点击Definer emote会弹出下面的对话框，输入url地址即可



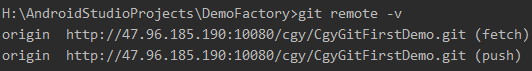
## 如何取消git关联

点击红色减号按钮即可，如果需要把git目录一同删去，需要去git文件夹所在目录，手动删除



## 查看git远程仓库地址

Terminal中输入git remote -v



# 坑

## early EOF index-pack failed

使用命令git config --add core.compression -1（或9）。compression 是压缩的意思，从 clone 的终端输出就知道，服务器会压缩目标文件，然后传输到客户端，客户端再解压。取值为 [-1, 9]，-1 以 zlib 为默认压缩库，0 表示不进行压缩，1..9 是压缩速度与最终获得文件大小的不同程度的权衡，数字越大，压缩越慢，当然得到的文件会越小

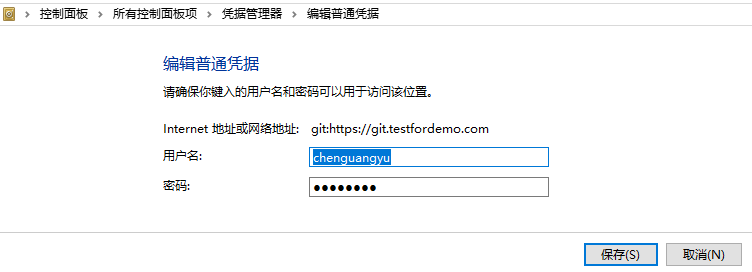
## SSL certificate problem

解决方法其实就是“直接不管ssl证书的事儿”，用git配置命令：git config --global http.sslVerify false

## 私有git输入账号密码错误

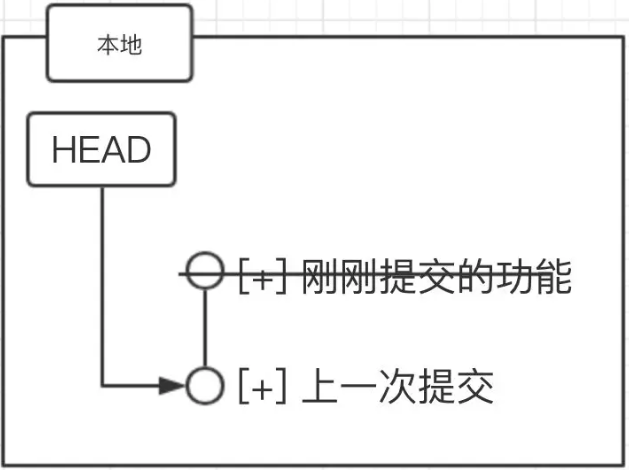
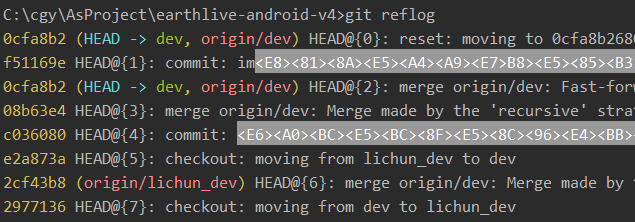
拉取项目时把git账户输入错误了，而Android studio还傻傻的记住错误的账户密码，重新打开也不会提示重新输入!尝试过设置不记住密码，然后并不起作用。解决办法：打开电脑控制面板——用户账户——管理Windows凭据——(找到git相关的)点击编辑





# 没有提交到远程就使用reset

使用git reset --hard commit—id命令，将工作区的提交穿越到你指定的commit里，这个时候你会发现git log根本没有记录这之后的提交记录，不要太紧张，Git提供了一个命令git reflog用来记录你的每一次改变目录树的命令，使用好他就可以很方便的恢复你的提交

我们的主线往往是一根直线，多一个分支相当于多一个分叉，无数分支纵横交错就像一颗树状的结构，所以我们称之为目录树。commit rebase reset merge这些都是改变目录树的操作。

git log 可以看到所有分支的历史提交，但看不到已经被删除的commit 记录和 reset rebase merge 的操作。而git reflog可以看到，最前面的黄色字符串就是之前的版本号了。比如用git reset --hard f51169e回退到被reset前的版本