重点

1. *版本回滚——git reset*

reset和revert的区别：reset会删除某commit-id之后所有的commit

https://blog.csdn.net/weixin_44802825/article/details/104814984

原理

working tree ----add----> 暂存区 ----commit----> 本地仓库 ----push----> 远程仓库

reset加不加hard的区别

```
reset --soft    //仅仅将HEAD指向新版本号

reset    //将HEAD指向新版本号，且更改暂存区（reset --mixed是默认情况，--mixed可省略）

reset --hard    //将HEAD指向新版本号，且暂存区和工作区一起更改
```

> https://blog.csdn.net/albertsh/article/details/106448035
>
> HEAD——指向当前分支的最新的commit
>
> HEAD^和HEAD~
>
> ^x: 尖头符号，形似箭头，表示要朝那个方向，始终是走一步，x 表示第几个岔路口，代表方向盘
> ~y: 波浪符号，表示要在该方向上走 y 步，始终沿着该方向，代表油门
>
> 当前节点的祖宗节点如下：
>
> 自己: HEAD, HEAD^0 或 HEAD~0
> 父亲: HEAD^, HEAD~
> 母亲: HEAD^2
> 爷爷: HEAD^~, HEAD~2, HEAD^^
> 奶奶: HEAD^^2, HEAD~^2
> 姥爷: HEAD^2~, HEAD^2^
> 姥姥: HEAD^2^2

G-D-B-A可以认为是主干，其他都是merge进来的其他分支节点。

HEAD~默认是HEAD^

已add未commit

git reset HEAD就是回退到当前版本——用本地仓库还原暂存区

add以后我们发现工作区中添加了错误的内容并且add了，此时我们只是做了add 操作，就是将修改了内容添加到了暂存区，**还没有执行commit，所以还没有生成版本号，当前的版本号对应的内容**，还是你add之前的内容，所以我们只需要将代码回退到当前版本就行。

其实到这里，暂存区的修改就撤销了。工作区中内容可用下面的方法撤销，但是没什么必要，直接修改就行了。除非，**你需要一步将Workspace中某些文件还原。**

（执行**git reset HEAD**后，***用git status查看，发现已经退到未add的状态（Workspace中有未track的文件）。***这表明暂存区已经被还原到add之前的状态。但是Workspace中有未track的文件恰恰表明，Workspace中的错误文件还在。要想**回退Workspace中的文件的错误修改**的话：`git checkout \<filename>`）

这样，先撤销add到暂存区，再撤销对Workspace某个文件的修改，最后完全撤销对文件的错误修改。

已commit

已经commit了，还没有push,push的内容我们先不管，push这个命令其实和提交没关系，他只是推送到远程了，如果push了，也就是我们**回退了之后，再重新push一下而已**，所以请不要纠结push这个操作。他和提交版本其实没有关系的。

已经commit了，说明已经生成了最新的版本号了，此时我们想回退，则肯定是回退到之前的一个版本，版本号用git log查看。 git为我们提供了一个更简单的回退上一个版本的方法 **git reset HEAD~**,此命令专门用于回退到上一个版本。若你的错误已经经过好几次commit，回退到上一个版本无法解决时，就查看版本号，用git reset 版本号回退。

回退后，就进入了（1）中 的状态，再按照（1）中描述进行回退。

关于git reset --hard——用本地仓库还原暂存区+工作区

用来撤销已add未commit。

该指令和git reset HEAD+ it checkout -- to discard changes in working directory两步相同，一步到位直接将暂存区和Workspace都回退到本地仓库中的当前版本。

日常使用情景

- 未commit，未生成新的版本号

  用暂存区回滚working tree

  ```
  git checkout .
  ```

- 未commit

  用本地仓库**回滚暂存区**，将暂存区和HEAD保持一致

  ```
  git reset HEAD
  ```

- 未commit

用本地仓库回滚**暂存区和WorkSpace**，将工作区、暂存区和HEAD保持一致

```
git reset --hard HEAD
```

- 已commit，生成了新的版本号

  回滚本地仓库到上一版本号

```
git reset HEAD~1        //HEAD指向上个commit，
git reset 版本号
```

- 已commit

  回滚本地仓库到上一版本的同时，将工作区和暂存区也于新的HEAD保持一致

```
git reset --hard HEAD~1
git reset --hard 版本号
```

---

2 远程仓库回滚

当代码已经Push到远程仓库后，发现push的代码有问题，怎么回滚到之前的版本呢?

---

3. *分支操作*

分支操作的各种情形下的处理方法，最好的办法是明确提交分支的提交路径（commit快照），脑子里要有几条平行的分支的提交记录。

本地电脑上不但有本地的分支HEAD，也会记录远程仓库的指针origin/master和origin/dev，push时和远程仓库同步

https://www.cnblogs.com/andydao/p/6808431.html

（1）新建分支

commit后才会新建分支master，这之后才可以新建其他分支

```
git branch daily/0.0.0    //新建日常开发分支daily/0.0.0
```

```
$ git branch

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git branch daily/0.0.0
fatal: Not a valid object name: 'master'.

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git add .
warning: LF will be replaced by CRLF in .gitignore.
The file will have its original line endings in your working directory

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   "\345\255\246\344\271\240git.md"


Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git branch

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git branch daily/0.0.0
fatal: Not a valid object name: 'master'.

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git commit -m "first commit "
[master (root-commit) 7d1147b] first commit
 2 files changed, 2 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 "\345\255\246\344\271\240git.md"

Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (master)
$ git branch
* master
```

(2) 重命名分支

```
git branch -m <oldbranch> <newbranch>    //重命名分支
git branch -M <oldbranch> <newbranch>    //强制重命名
```

(3) git branch -d——删除分支

- 删除本地分支daily/1.0.0

```
git branch -d daily/1.0.0
```

- 强制删除本地分支

```
git branch -D daily/1.0.0
```

- 删除远程分支**(慎用)**

```
git push origin --delete daily/1.0.0
```

（4）git branch -a——查看分支列表

```
git branch    //当前项目分支列表
```

```
git branch -a    //查看本地仓库和远程仓库上所有分支列表

git branch -r    //查看远程仓库所有分支列表
git branch -r -d origin/branch-name        //查看并删除远程仓库上分支branch-name
git branch -D    //分支未提交到本地版本库前强制删除分支
bit branch -vv    //查看本地仓库分支列表，带有各分支的最后提交id、各本地分支与远程分支的关联情况。ahead标识本地仓库超前远程仓库几个commit
```

（5）git checkout——切换分支

切换分支，HEAD会改变指向

```
git checkout daily/1.0.0
```

创建的同时**切换**`daily/0.0.1`分支

```
git checkout -b daily/0.0.1
```

创建dev分支的同时，**关联分支**origin/dev

```
git checkout -b dev origin/dev    //origin/dev是拉取到本地的远端仓库
```

（6）git stash——暂存

- 原理

  stash就是一个栈，把add后再在working tree发生的修改进行暂存，并把工作区恢复到修改之前的状态

  只有未add的文件才能stash

  可以跨分支恢复

- 操作

  > - 暂存当前工作进度，将工作区和暂存区恢复到修改之前
  >
  > ```
  > git stash save "message"
  > ```
  >
  > - 显示暂存列表，编号`stash@{num}`越小代表保存进度的时间越近（显示栈内）
  >
  > ```
  > git stash list
  > ```
  >
  > - 恢复到工作区
  >
  > ```
  > git stash pop stash@{num}    //暂存从栈中弹出。因此只能恢复工作区一次
  > git stash pop //等价于git stash pop stash@{0}
  > git stash apply stash@{num}    //暂存不从栈中弹出，可多次恢复工作区
  > ```

- 删除栈中某一暂存、清空栈

```
git stash drop stash@{num}    //删除stash@{num}
git stash clear    //清空栈
```

- 应用场景

  - 忘记切换到自己的分支，在不该修改的分支上修改了工作区

```
//暂存上次add后的修改
git stash save "message"
//切换到自己的分支
git checkout myBranch
//释放暂存
git stash pop
```

  - 远程分支改动未pull就在本地修改，且发生了冲突，pull的话需要解决冲突

    为了避免合并解决冲突的情况，在本地修改还未add时，可以先暂存本地修改，pull后，再把修改放出来到working tree

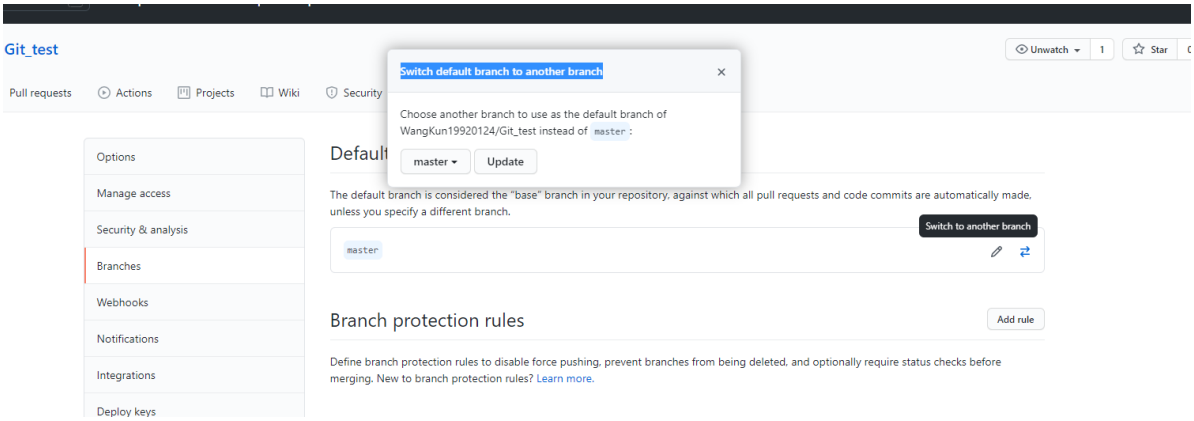
```
//暂存未add修改
git stash save "message"
//pull
git fetch origin master
git merge origin/master
//释放暂存
git stash pop
```

  - git merge时有未commit的文件，先stash，再merge，否则一旦merge出问题，用merge --abort无法恢复到merge之前的状态

```
//工作区有未commit或未add的文件
git stash save "temp for merge"
//merge
git merge
//fix confliction
//stash pop
stash pop
```

(7) default分支

修改default分支



> Settings----->Branches------->右侧箭头

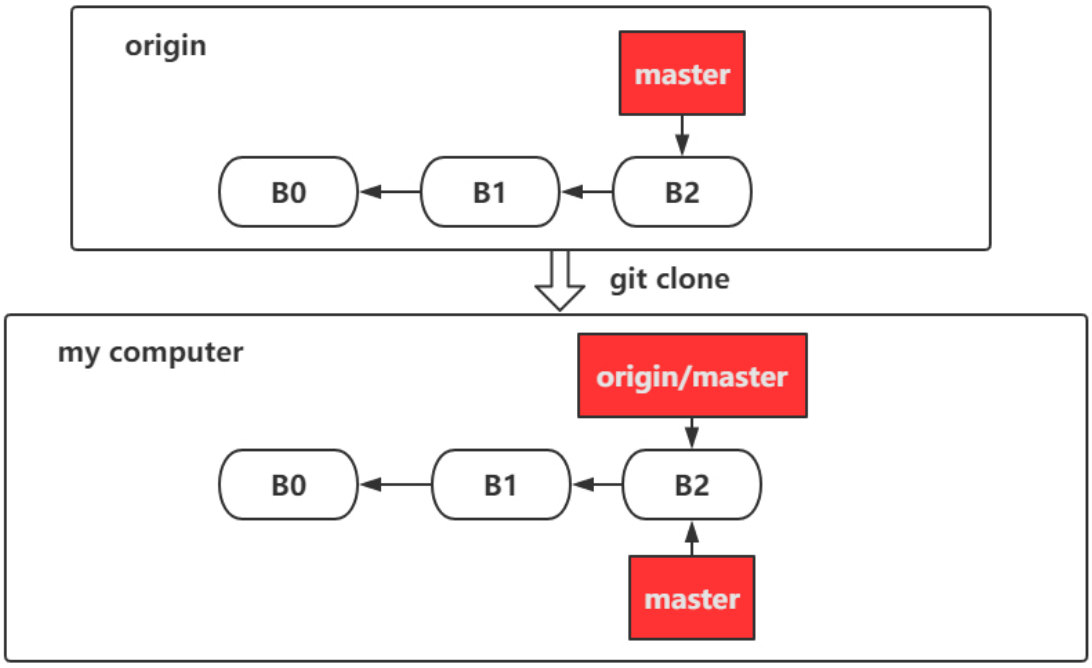
（8）分支的日常使用情景

---

4. 冲突操作

（1）git clone

git clone拉取远程仓库代码，自动创建远端仓库指针origin/master指向远端仓库的最新commit，同时创建本地指针master同origin/master指向同一commit

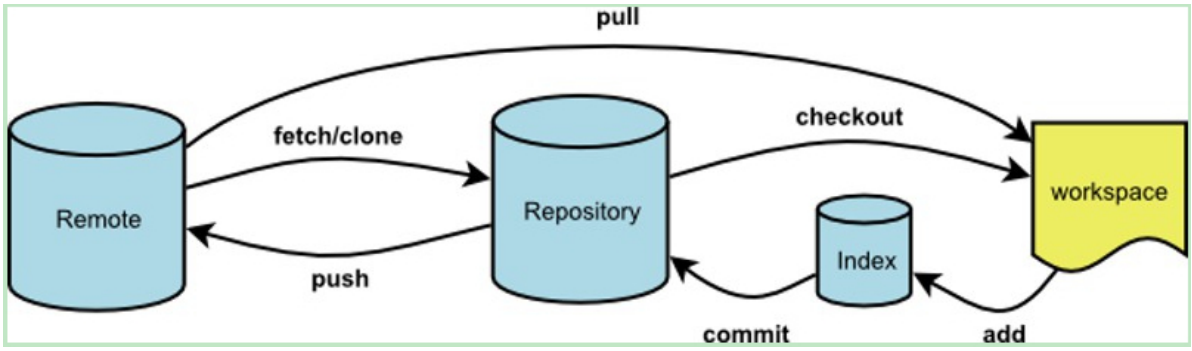


（2）git fetch

https://blog.csdn.net/qq_42780289/article/details/98049574

https://www.jianshu.com/p/d07f5a8f604d

fetch仅到本地仓库，不到暂存区和working tree



git fetch [remote-name] [branch-name]

- 指定远程仓库，默认是master分支

  ```
  git fetch [remote-name]
  ```

- 取回特定分支

  ```
  git fetch [remote-name] [branch-name]
  ```

  比如，取回origin仓库的master分支

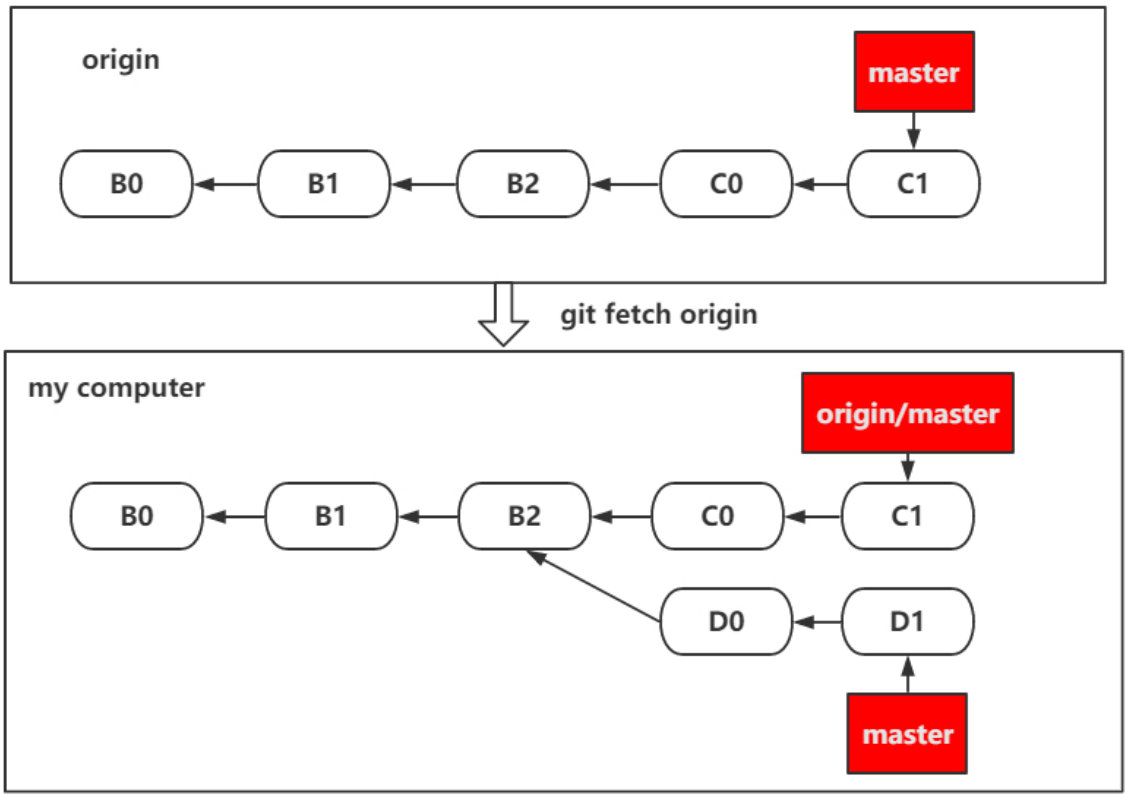
  ```
  git fetch origin master
  ```

- 取回的代码在本地主机上会建立指针`远程主机名/分支名`指向拉取的分支的最新commit。如：fetch master分之后更新：origin/master
- git fetch命令通常用来查看其他人的分支的开发进程，因为它**取回的代码对你本地的开发代码没有影响**

  对于拉取的代码分支，查看后，若不需要，可用`git branch -d`删除

git fetch origin

从远程仓库origin抓取origin/master分支的代码和提交，origin/master指向远端仓库的最新的commit，和本地代码master形成分支

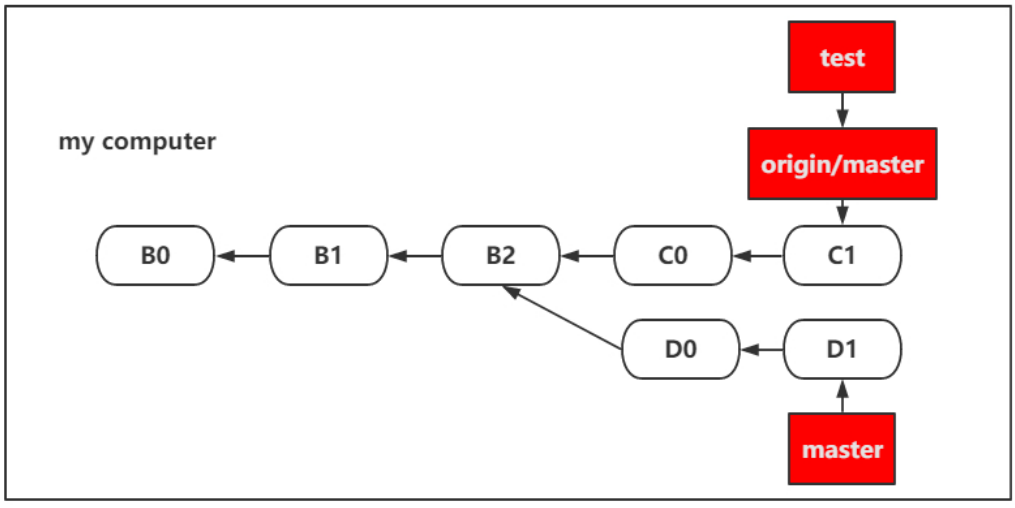


- 拉取了origin/master后，此时由于C0和D0，origin/master和master从B2开始产生了冲突。

  此时，1. 可以在origin/master的基础上工作。2. 可以将origin/master合并到master分支

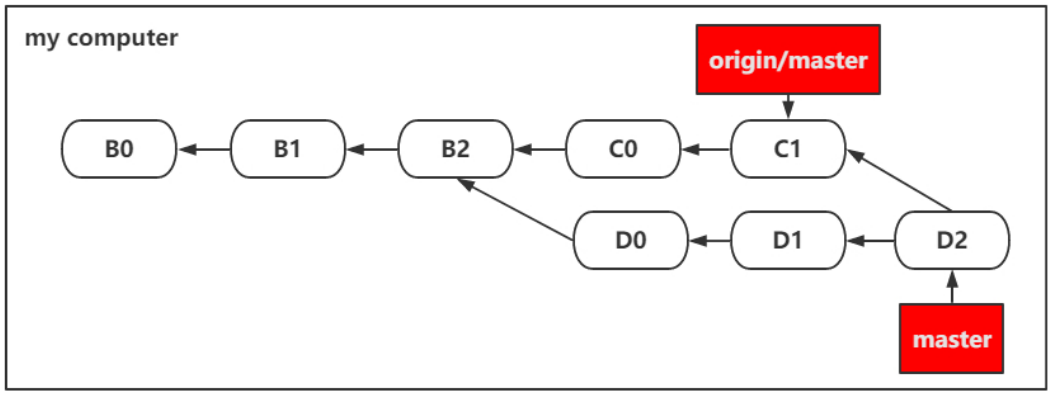
  > 1. 在origin/master基础上工作，而不是在master基础上工作的话。建立本地分支test关联远程分支origin/master

  ```
  git checkout -b test origin/master
  ```

  > 2. 将origin/master分支的内容和master分支合并

  ```
  //直接将origin/master合并到master分支
  git checkout master
  git merge origin/master --no-ff
  ```



（3）git merge

merge开始时最好不要有未commit的文件，否则merge --abort时可能会发生无法回到merge之前的状态。若有，用stash暂存，待merge结束确认无问题后再stash pop

图解说明，豁然开朗

https://www.jianshu.com/p/58a166f24c81

https://zhuanlan.zhihu.com/p/269043827

```
git merge [待合并分支]    //将待合并分支合并到当前分支
```

merge撤销

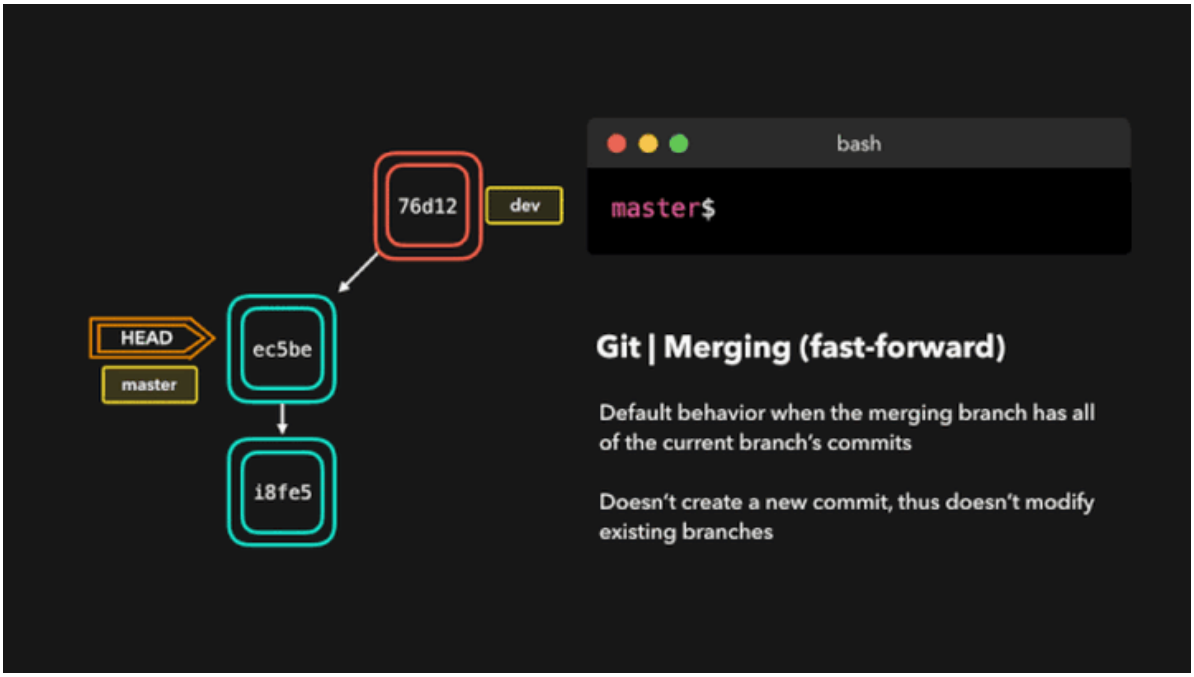
```
git merge --abort
```

```
git reset --hard HEAD~
```

fast-forward模式

在当前分支相比于我们要合并的分支没有额外的提交（commit）时，可以执行 fast-forward 合并。Git很懒，首先会尝试执行最简单的选项：fast-forward！这类合并不会创建新的提交，而是会将我们正在合并的分支上的提交直接合并到当前分支。

```
git merge --ff
```

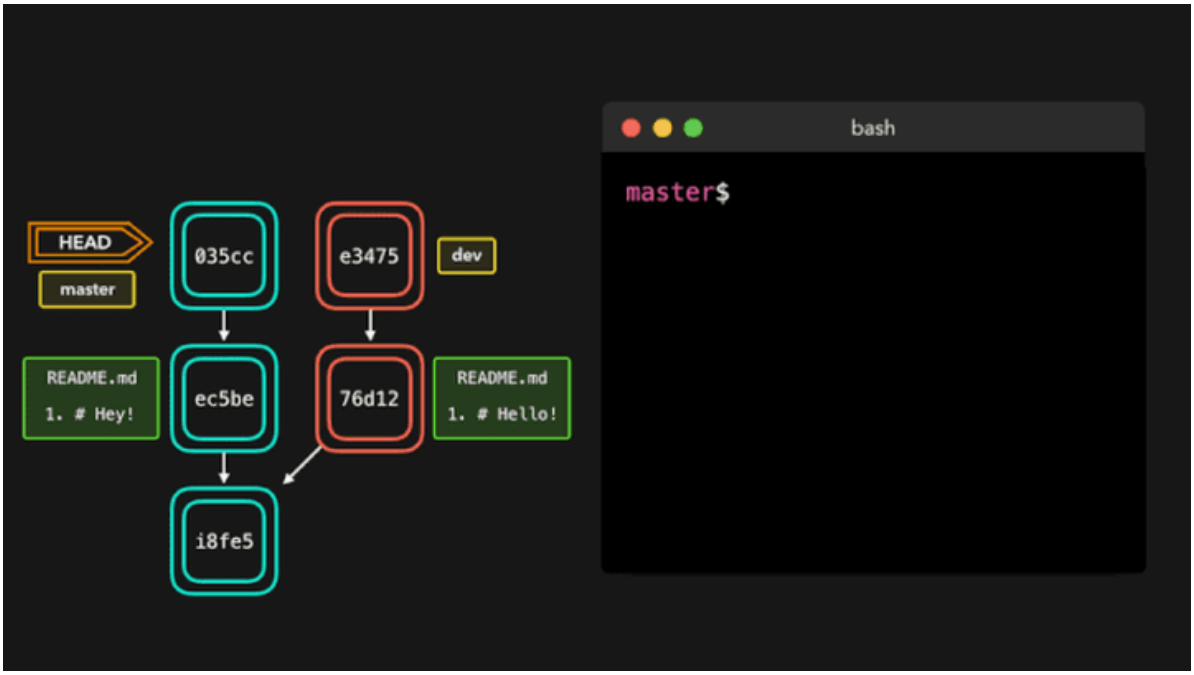


no-fast-forward模式

如果我们在当前分支上提交我们想要合并的分支不具备的改变，那么 git 将会执行 no-fast-forward 合并。

使用 no-fast-forward 合并时，Git 会在当前活动分支上创建新的 merging commit。这个提交的父提交（parent commit）即指向这个活动分支（HEAD^1），也指向我们想要合并的分支（HEAD^2）。

```
git merge --no-ff
```



(4) rebase

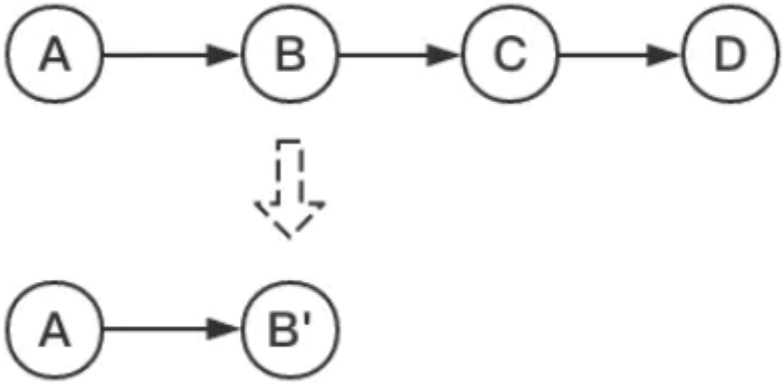
> https://zhuanlan.zhihu.com/p/145037478
>
> https://www.jianshu.com/p/4a8f4af4e803
>
> https://blog.csdn.net/nrsc272420199/article/details/85555911

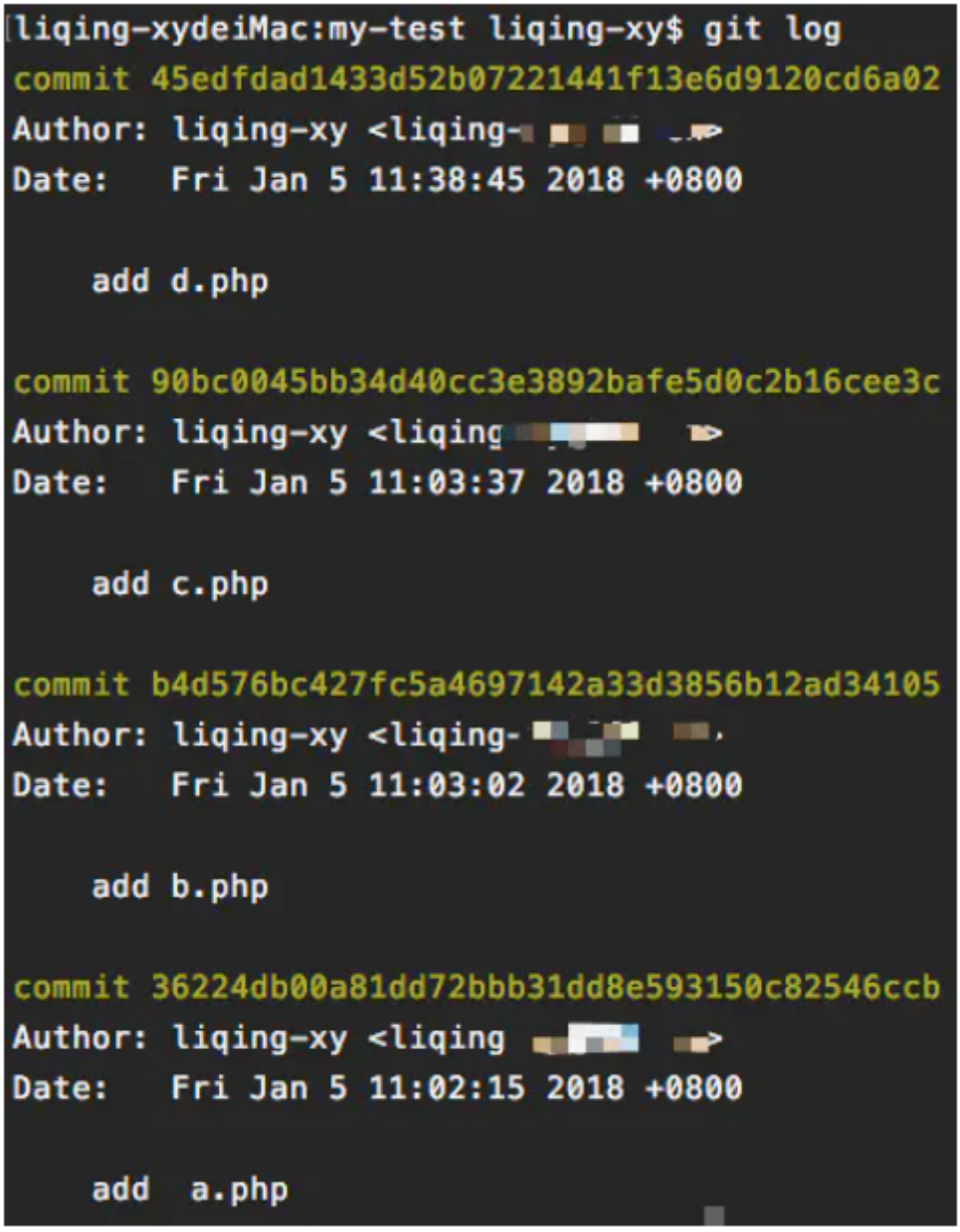
不要通过rebase对任何已经提交到公共仓库中的commit进行修改，只针对自己的开发分支进行。

rebase主要有两个用法：

合并多个commit为一个完整commit

> ***这种用法有一个重要问题：rebase后可以push吗？？***

> 当我们在本地仓库中提交了多次，在我们把本地提交push到公共仓库中之前，为了让提交记录更简洁明了，我们希望把如下分支B、C、D三个提交记录合并为一个完整的提交B'，然后再push到公共仓库。
>
> 
>
>
> 现在我们在测试分支上添加了四次提交，我们的目标是把最后三个提交合并为一个提交：



这里我们使用命令:

```
git rebase -i  [startpoint]  [endpoint]
```

其中`-i`的意思是`--interactive`，即弹出交互式的界面让用户编辑完成合并操作，`[startpoint]` `[endpoint]`则指定了一个编辑区间，如果不指定`[endpoint]`，则该区间的终点默认是当前分支`HEAD`所指向的`commit`(注：该区间指定的是一个前开后闭的区间)。

[startpoint]指定为所有要合并的commit-id的前一个commit-id

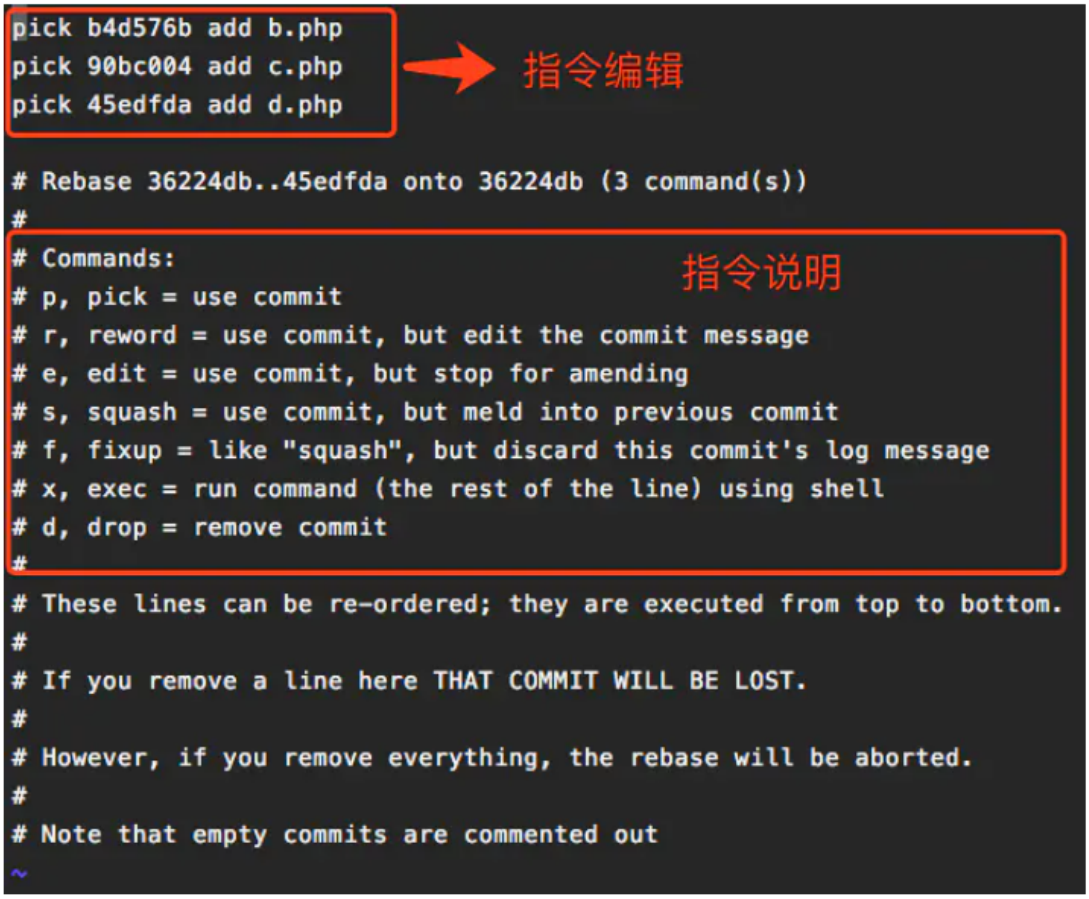
在查看到了log日志后，我们运行以下命令：

```
git rebase -i 36224db
```

或:

```
git rebase -i HEAD~3 //将包含HEAD在内的n个commit合并的话，就是HEAD~n
```

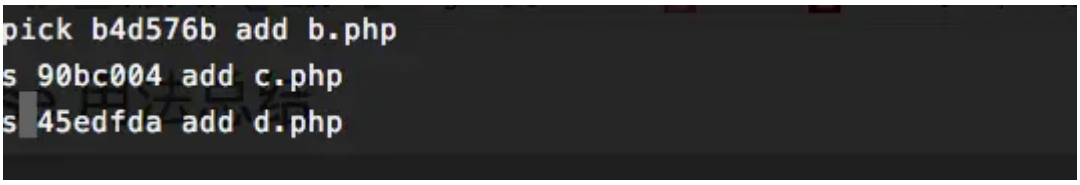
然后我们会看到如下界面:



上面未被注释的部分列出的是我们本次rebase操作包含的所有提交，下面注释部分是git为我们提供的命令说明。每一个commit id 前面的`pick`表示指令类型，git 为我们提供了以下几个命令:

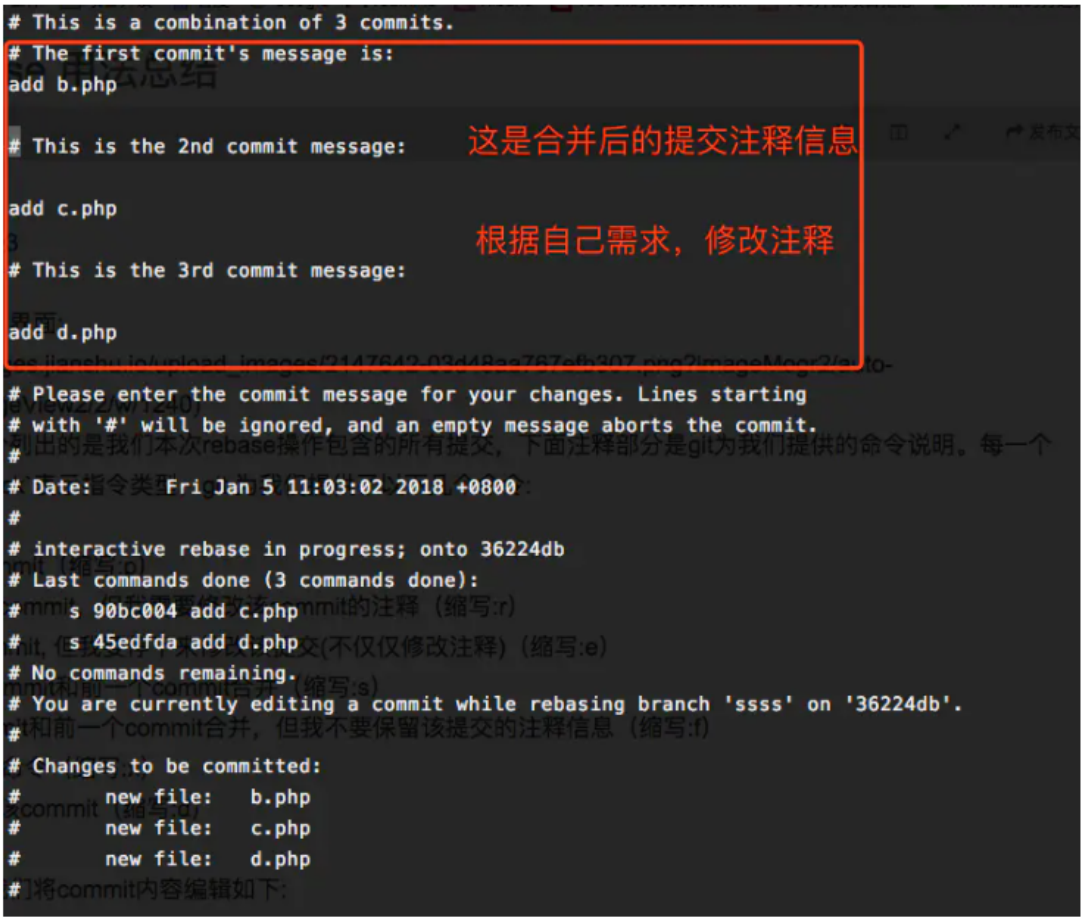
> - pick：保留该commit（缩写:p）
> - reword：保留该commit，但我需要修改该commit的注释（缩写:r）
> - edit：保留该commit, 但我要停下来修改该提交(不仅仅修改注释)（缩写:e）
> - squash：将该commit和前一个commit合并（缩写:s）
> - fixup：将该commit和前一个commit合并，但我不要保留该提交的注释信息（缩写:f）
> - exec：执行shell命令（缩写:x）
> - drop：我要丢弃该commit（缩写:d）

根据我们的需求，我们将commit内容编辑如下:

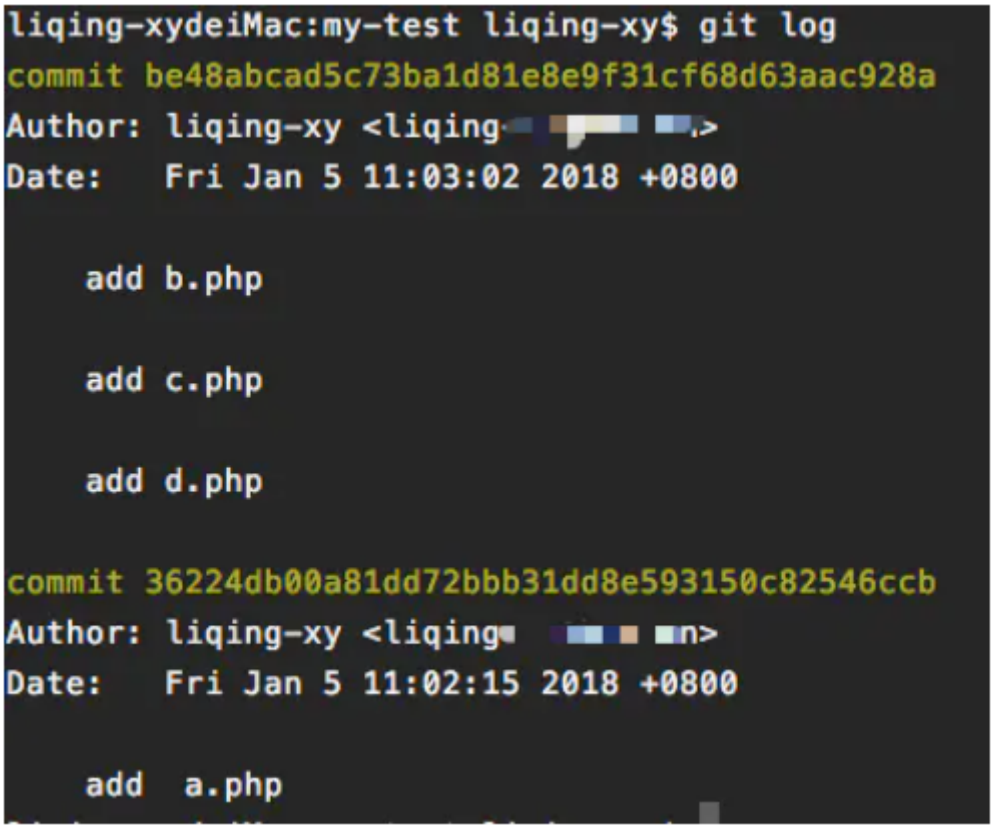


然后是注释修改界面:

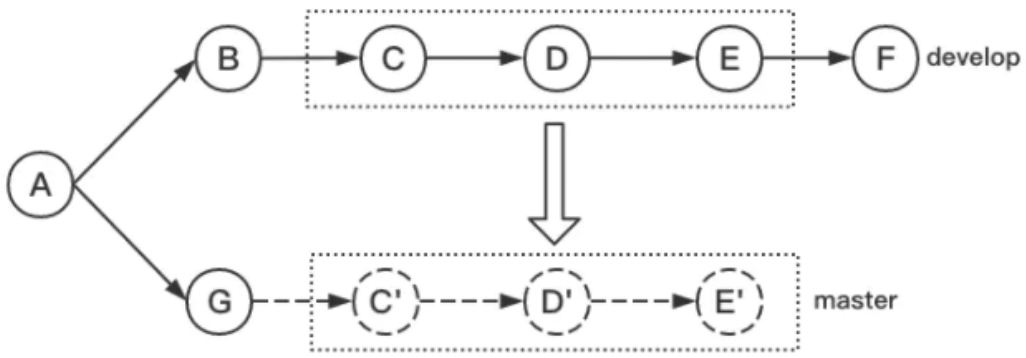
最终保留的是哪个commit，就修改哪个commit的message



编辑完保存即可完成commit的合并了：



将某一段commit粘贴到另一个分支上

> 当我们项目中存在多个分支，有时候我们需要将某一个分支中的一段提交同时应用到其他分支中，就像下图：
>
> 
>

rebase撤销

（5）merge和rebase用哪个?

- 说法一

  不要用merge，用rebase，https://zhuanlan.zhihu.com/p/75499871

- 说法二

  https://zhuanlan.zhihu.com/p/57872388

  merge和rebase都要用，rebase会隐藏分支的提交历史，最终呈现的结果只有master上的若干分支合并历史，而分支上的commit历史则会消失。

- 两者区别

  merge会保留待合并分支的提交记录。rebase最终只保留一条分支记录。

（6）git pull=git fetch +git merge

https://blog.csdn.net/weixin_41975655/article/details/82887273

使用git pull的会将本地的代码更新至远程仓库里面最新的代码版本

3. 总结

- 由此可见，git pull看起来像git fetch+get merge，但是根据commit ID来看的话，他们实际的实现原理是不一样的。
- 这里借用之前文献看到的一句话：

> **不要用git pull，用git fetch和git merge代替它。**
>
> **git pull的问题是它把过程的细节都隐藏了起来，以至于你不用去了解git中各种类型分支的区别和使用方法。当然，多数时候这是没问题的，但一旦代码有问题，你很难找到出错的地方。看起来git pull的用法会使你吃惊，简单看一下git的使用文档应该就能说服你。**
>
> **将下载（fetch）和合并（merge）放到一个命令里的另外一个弊端是，你的本地工作目录在未经确认的情况下就会被远程分支更新。当然，除非你关闭所有的安全选项，否则git pull在你本地工作目录还不至于造成不可挽回的损失，但很多时候我们宁愿做的慢一些，也不愿意返工重来。**

Git日常使用

1. 使用已有的rsa私钥在另一台电脑上使用已有远程仓库

设置用户名： git config --global user.name wk

设置密码： git config --global user.password p

将私钥复制到~/.ssh文件夹下：C:\Users\Administrator.ssh

身份验证： ssh -T git@github.com

验证成功：Hi 用户名! You've successfully authenticated, but Github does not provide shell access.

将远程仓库clone下来，就可以提交修改了（clone别人的代码修改后无法提交，需要别人同意）

2. gitk——启动图形查看模式

3. 修改上次commit的message——git commit --amend

```
git commit --amend
```

合并暂存区的修改和HEAD的commit，然后生成新的commit（**commit ID变化了**），用新的commit替代原来的commit。若暂存区没有修改，就相当于替换commit ID和重写message。（连带当前的暂存区内容提交，生成新的commit-id，替换HEAD）

功能1——修改message

```
git commit --amend -m "提交原因"
```

功能2——修改上次commit

> ```
> //做了一些修改
> git add .
> git commit --amend --no-edit    //--no-edit表示不重新编辑message
> ```
>
> 其实也是可以用git reset方法
>
> ```
> git reset --hard HEAD~    //用HEAD~还原本地仓库、暂存区、working tree
> //修改
> git commit -m "重新提交"
> ```

3. git push时会更新本地的origin/分支的指向

4. HEAD是什么？HEAD^/HEAD~/HEAD~2/HEAD^2是什么？Origin是什么？master是什么？Branch是什么？git push -u中的u是什么？

https://www.zhihu.com/question/20019419

- Branch——分支
- Origin——`origin` 指代的是当前的git服务器地址

- master——`master` 是 `Git` 为我们自动创建的第一个分支，也叫主分支，其它分支开发完成后都要合并到 `master`
- HEAD——指向**当前分支的最新的提交。作用很像是数据结构中指向二叉树根节点`root`的指针。有个 `root` 指针我们就可以对二叉树进行任意操作，它是二叉树的根基。**而 `git` 中的 `HEAD` 概念也类似一个指针，它指向是当前分支的"头"，通过这个头节点可以追寻到当前分支之前的所有提交记录。git 的提交记录之间的关系很像一棵树，或者说是一张图，通过当前的提交记录指向上一次提交记录串联起来，形成一个头结构，而在 git 中我们常常说的切换分支，只不过是 git 客户端帮你把要操作的那条路径的头节点，存储到了 HEAD 文件中。

  HEAD 在 git 版本控制中代表头节点，也就是分支的最后一次提交，同时也是一个文件，通常在版本库中 repository/.git/HEAD，其中保存的一般是 ref: refs/heads/master 这种分支的名字，而本质上就是指向一次提交的 hash 值，一般长成这个样子 ce11d9be5cc7007995b607fb12285a43cd03154b。
- HEAD^2/HEAD~2

  https://blog.csdn.net/albertsh/article/details/106448035

  https://www.cnblogs.com/mengff/p/12809911.html

  https://www.cnblogs.com/chjbbs/p/6418339.html
- git push -u中u——使用一次`git push -u origin master`后，告知Git记忆相关参数，下次只需要`git push`即可。（用-u（--up-stream）来建立本地分支与远程某个分支的关联，形成一个管道，之后 git push可以直接沿着管道 到达关联的分支 无需在加-u参数了）

5. 多次commit，想把这若干commit合并为一个——git rebase

当我们开发一个功能，不是一时半会可以完成的时候，为了保护代码不丢失，通常会把这次的修改先 commit，等到这个功能完全做好，再 push。不过这样一来，就会有很多零碎的 commit 记录，这会使远程的提交历史显得杂乱。

必要的时候，我们需要将这些相近的 commit 合并为一个 commit 再 push。当然了，如果你想合并远程的 commit 也是可以的，但请一定要提前跟团队里的其他人说一声，免得有人也在跟你做同样的事情，导致没必要的代码冲突（所以合并 commit 尽量在 push 前）。

https://blog.csdn.net/yinchuan_111/article/details/106913632

https://www.jianshu.com/p/571153f5daa1

```
git log --oneline    //可以更加简明的展示commit日志，且显示的commit-id是简化形式
```

```
git rebase -i commit-id    //commit-id为要最终合并的commit-id的上一条commit。执行完后，会从下条commit开始显示
```

```
修改除第一条commit-id（想要保留的commit）以外的下方若干commit，commit从上往下是逐渐接近最新提交的，最下方的是最新的commit
将除第一条以外的pick改为squash/s，然后:wq保存退出
成功的话，会出现编辑合并后commit的message。编辑新的message，然后:wq保存退出
```

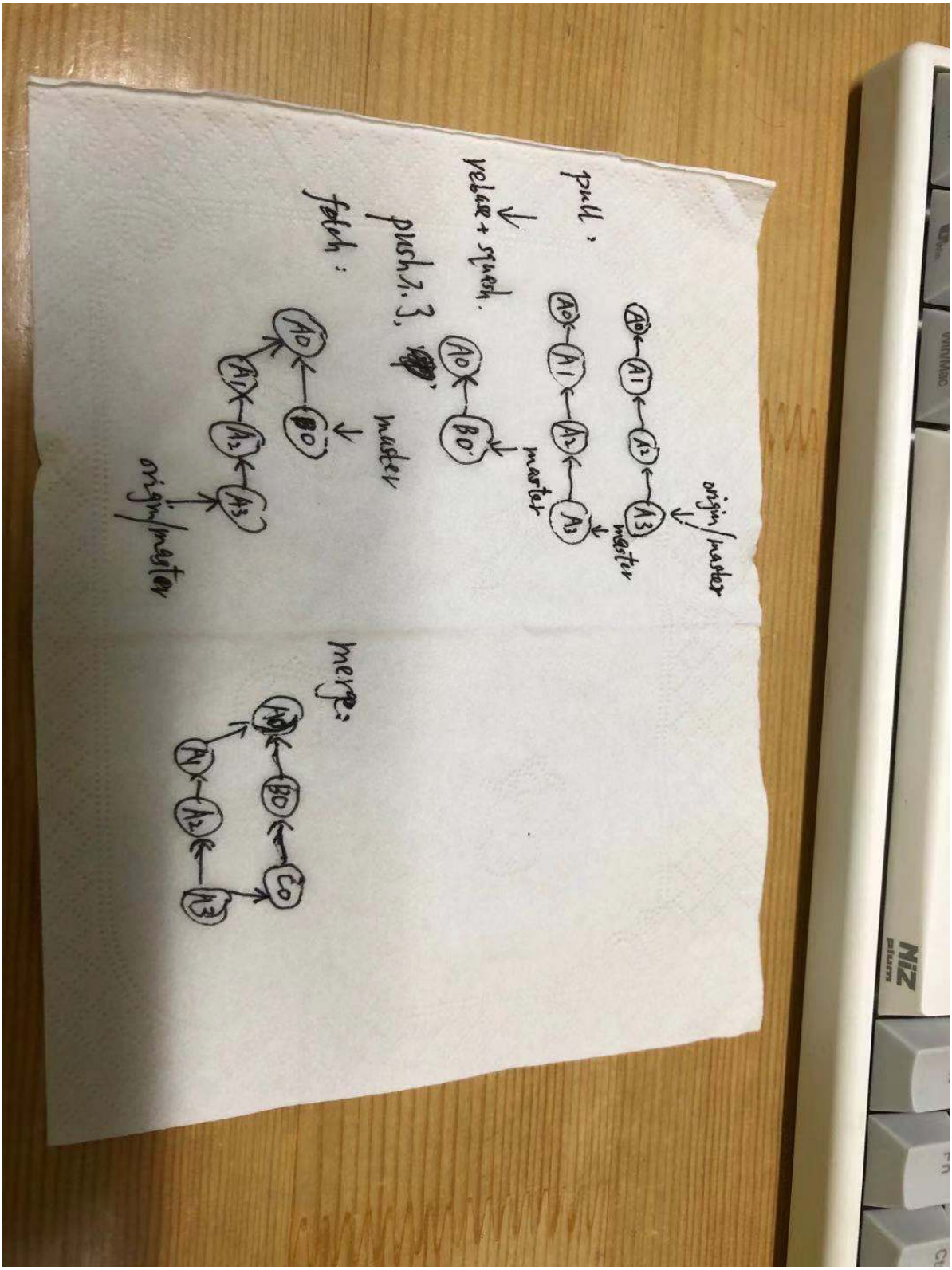
此时会遇到不能push

因为squash和变基，将几个commit合成了一个，而且**生成了新的commit id**。此时本地分支和远端分支就在新commit id处和远端产生了分支

本地master需要fetch+merge远端分支，才可以push。

1. 不要随便commit，产生多次commit后push。

2. 尽量使用 `git commit --amend --no-edit`



6. 查看文件的修改情况

https://blog.csdn.net/cnds123321/article/details/110731428

- git status

  git status 命令用于**显示工作目录和暂存区**的状态。使用此命令能看到那些修改被暂存到了, 哪些没有, 哪些文件没有被Git tracked到。`git status`不显示已经`commit`到项目历史中去的信息。看项目历史的信息要使用`git log`.

  总的来说：红色的是没有track的，需要add到暂存区。绿色的是已经track的。new file/modified/deleted显示文件是新建/修改/删除。根据提示to be commited表明是否commit。总的来说，就是是否track和是否commit。

> a) 已暂存、未提交 （changes to be committed）——红色
>
> new file //表示新建文件，待commit
>
> modified //表示修改文件
>
> deleted //表示删除文件
>
> b) 已修改（changed but not updated）——绿色
>
> modified //表示修改文件
>
> deleted //表示删除文件
>
> 另外，git 给出了可能需要的操作命令，git add/rm, gitcheckout --
>
> c) 未跟踪 （untracked files）
>
> 没有用add添加到暂存区的文件

- git status -s

  以更简单的形式显示状态。

  - ??标记：**新添加的未跟踪文件**前面会有红色的??标记，即没有执行git add .命令的文件。
  - A标记：**新添加到暂存区的文件**前面会有绿色的A标记，即已经执行了git add .命令的文件。
  - AM标记：A表示该文件**加入到暂存区了，而M表示该文件被修改过了，修改后的文件还没有添加到暂存区。**

- git log

  查看commit的日志

- git commit --amend

  功能1：会出现一个编辑器，然后可以修改上一次的提交信息，按键盘上的Insert键进行插入，修改完成后按Esc键并输入:wq保存退出

7.日常操作流程

（1）直接从别人的项目里拷贝到文件夹里

无需初始化仓库，直接在文件夹里就可以git clone。克隆下来的代码，修改要想提交到远程，必须向原作者发起pr，原作者merge后才可合并到远程仓库。

> git clone `ssh`

（2）从自己的远程仓库里拉取代码到本地新建的空文件夹

> git init　//建立本地仓库
>
> git remote add origin `ssh`　//连接远程仓库
>
> git pull origin master //拉取远程仓库代码

https://www.runoob.com/git/git-pull.html

git pull 其实就是 **git fetch** 和 **git merge FETCH_HEAD** 的简写。 命令格式如下：

```
git pull <远程主机名> <远程分支名>:<本地分支名>
```

将远程主机 origin 的 master 分支拉取过来，与本地的 brantest 分支合并。

```
git pull origin master:brantest
```

如果远程分支是与当前分支合并（拉取远程仓库代码），则冒号后面的部分可以省略。

```
git pull origin master
```

（3）将服务器上的最新代码拉取到本地文件夹（已是项目文件夹，有.git文件夹）

> git pull origin master

若本地代码被修改，但是未push到远程服务器。且在另一个主机上修改了服务器上代码。那么pull时，服务器上的改动和本地改动就会产生冲突。一般，git会自动合并冲突。但若涉及到同一行代码的改动，就需要手动合并代码

8. push时的忽略文件

```
touch .gitignore    //创建.gitignore文件，编辑该文件，每行代表push时忽略不上传的文件
```

```
//.gitignore
1.md    //忽略1.md文件
build/    //忽略build目录下的所有文件
```

9. *tag*

当完成某项需求时，需要将代码打上一个版本tag，并push到线上。

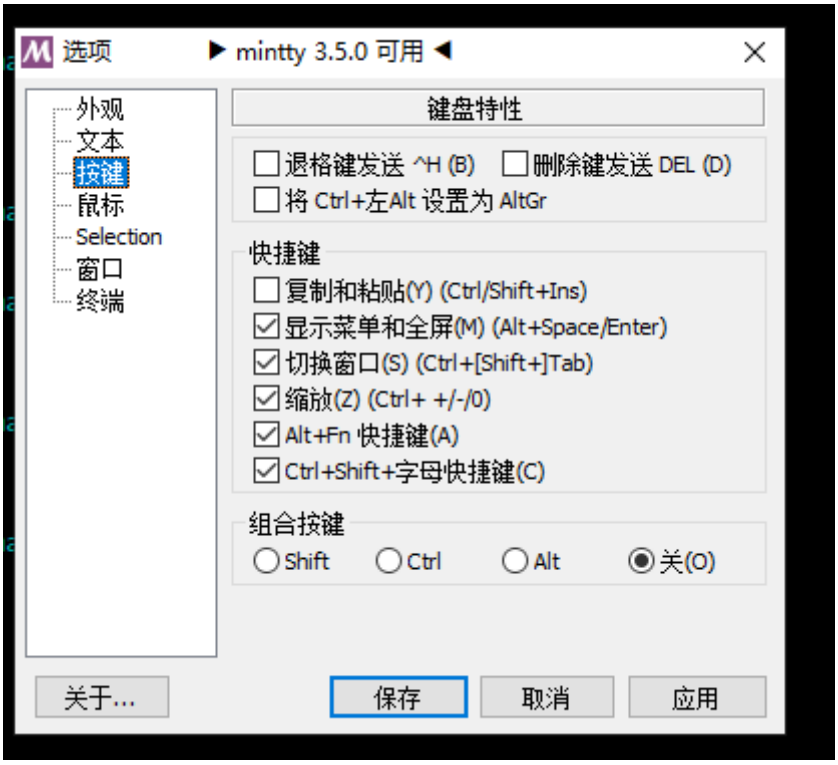
标签是用于标记特定的点或提交的历史，通常会用来标记**发布版本的名称或版本号**（如：`publish/0.0.1`），虽然标签看起来有点像分支，但打上标签的提交是**固定**的，不能随意的改动。

```
git tag publish/0.0.1    //给当前代码打上标签publish/0.0.1，表示当前代码以
publish/0.0.1发布
git push origin publish/0.0.1    //推送到服务器
```

10. git bash快捷键

- windows系统，在工作目录下，shift+F10，再按s，再按enter。
- 复制：ctrl+shift+c
- 粘贴：ctrl+shift+v

  复制和黏贴快捷键需要按下图设置后才能使用



Git高级

1. git add

2. git commit

git commit -m "" -m ""——多行提交原因

但是这种方式只能写一行的注释，如果你想要对commit的内容进行详细的讲解，以便仔细检查提交的文件，那你可能需要写多行注释，这个命令就不适用了。

```
git commit -m "commit title" -m "commit description"
```

```
Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (daily/0.0.0)
$ git commit -m '第一行提交原因' -m '第二行提交原因'
[daily/0.0.0 a1de935] 第一行提交原因
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 "git commit \345\244\232\350\241\214\346\217\220\344\272\244\345\216\2
37\345\233\240.txt"
```

```
Administrator@EIVISION MINGW64 /d/WorkSpace/git_test (daily/0.0.0)
$ git log
commit a1de93550aa8ec920eede5dbeb7715e46f06e9af (HEAD -> daily/0.0.0)
Author: WangKun19920124 <2522466153@qq.com>
Date:   Mon Jul 26 10:40:31 2021 +0800

    第一行提交原因

    第二行提交原因
```

git commit --amend——修改最新一条提交记录的message

若暂存区发生了改变，会将暂存区的改变提交

```
git commit --amend -m "提交原因"
git commit --amend --no-edit
```

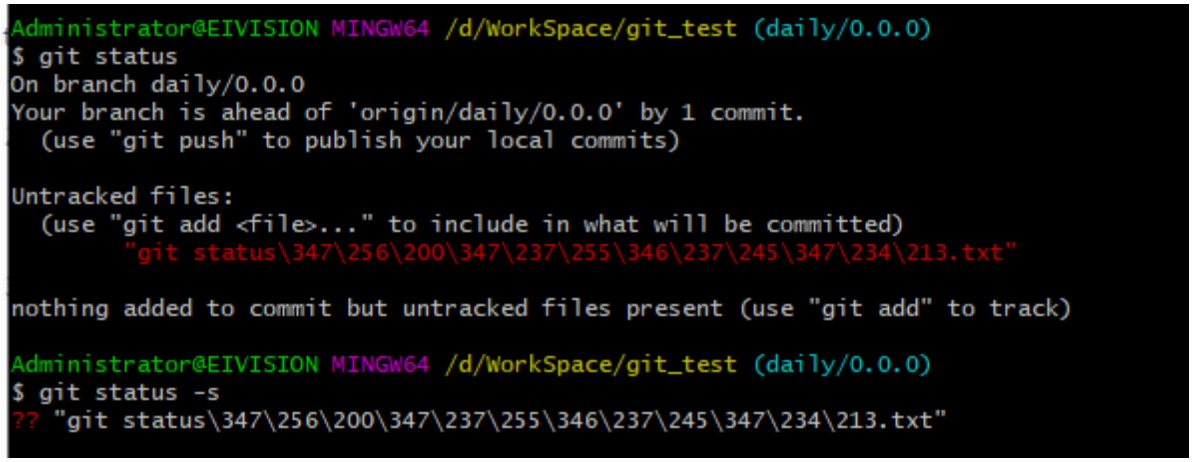
git commit -a -m

```
等价于git add + git commit -m
```

3. git status

- 简短方式查看status

```
git status -s
```



4. git blame <filename>

查看文件的修改者和修改内容

5. git whatchanged --since='2 weeks ago'

查看当前分支在某个时间节点前的commit log

6. 文件标识

> A: 增加的文件.
>
> C: 文件的一个新拷贝.
>
> D: 删除的一个文件.
>
> M: 文件的内容或者mode被修改了.
>
> R: 文件名被修改了。
>
> T: 文件的类型被修改了。
>
> U: 文件没有被合并(你需要完成合并才能进行提交)
>
> X: 未知状态。(很可能是遇到git的bug了，你可以向git提交bug report)

```
:000000 100644 0000000 e5e08cd A        C#/DevExpress20.1_V1.0.assets/image-20210727101946708.png
:000000 100644 0000000 5c86f42 A        C#/DevExpress20.1_V1.0.assets/image-20210727111359355.png
:000000 100644 0000000 52551d8 A        C#/DevExpress20.1_V1.0.assets/image-20210727111506947.png
:000000 100644 0000000 53c061f A        C#/DevExpress20.1_V1.0.assets/image-20210727141408458.png
:000000 100644 0000000 2ab9b1b A        C#/DevExpress20.1_V1.0.assets/image-20210727141435392.png
:000000 100644 0000000 2ab9b1b A        C#/DevExpress20.1_V1.0.assets/image-20210727141449186.png
:000000 100644 0000000 e302eb9 A        C#/DevExpress20.1_V1.0.assets/image-20210727141756830.png
:000000 100644 0000000 e4fe2ac A        C#/DevExpress20.1_V1.0.assets/image-20210727142330007.png
:000000 100644 0000000 bee4917 A        C#/DevExpress20.1_V1.0.assets/image-20210727142403870.png
:000000 100644 0000000 75f86dc A        C#/DevExpress20.1_V1.0.assets/image-20210727171219400.png
:100644 100644 fce48cc 0ed99c0 M        C#/DevExpress20.1_V1.0.md
:100644 100644 0e105e1 266eb3b M        "Git/Git\344\275\277\347\224\250\350\241\245\345\205\205.md"
```

7. git log --pretty=oneline --graph --decorate --all

```
--pretty=oneline    //一行显示完整commit-id
--oneline    //一行显示缩略commit-id
--graph    //图形化显示
--decorate    //显示tag
```

8. git reflog和git log

git log可以显示所有提交过的版本信息，不包括已经被删除的 commit 记录和 reset 的操作

git reflog是显示所有的操作记录，包括提交，回退的操作。一般用来找出操作记录中的版本号，进行回退

git reflog常用于恢复本地的错误操作

9. git push -f <remote-name> <branch-name>

强制推送到远端仓库

10. 修改git上传文件大小100M限制

方案一

上传项目到GitHub上，当某个文件大小超过100M时，就会上传失败，因为默认的限制了上传文件大小不能超过100M。如果需要上传超过100M的文件，就需要我们自己去修改配置。

首先，打开终端，进入项目所在的文件夹；

输入命令：git config http.postBuffer 524288000

之前git中的配置是没有这一项的,执行完以上语句后输入：git config -l可以看到配置项的最下面多出了一行我们刚刚配置的内容. (52428000=500×1024×1024,即500M)

```
zjdeMac-mini:AliyunUpload DT$ git config -l
credential.helper=osxkeychain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.name=懂 听
user.email=2510656506@qq.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
submodule.active=.
remote.origin.url=https://github.com/dt8888/AliyunUpload.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
http.postbuffer=524288000
zjdeMac-mini:AliyunUpload DT$
```

方案二

https://blog.csdn.net/weixin_37557729/article/details/107012028

11.VS中使用Git