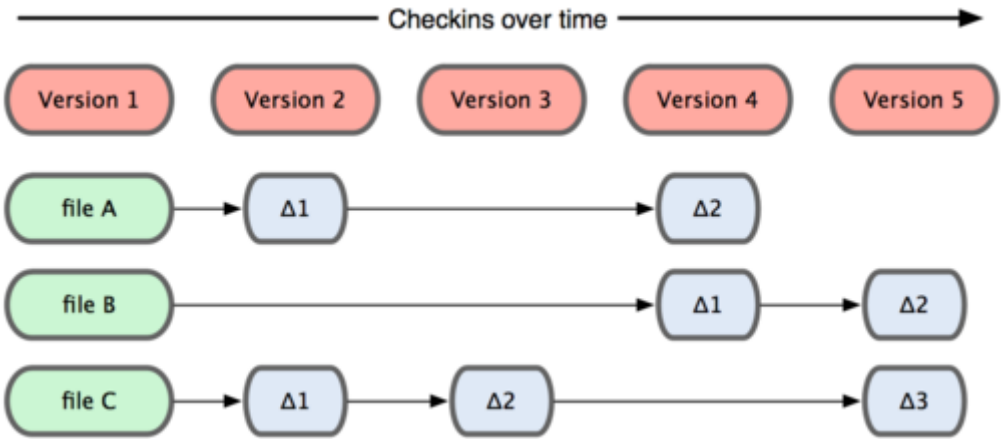
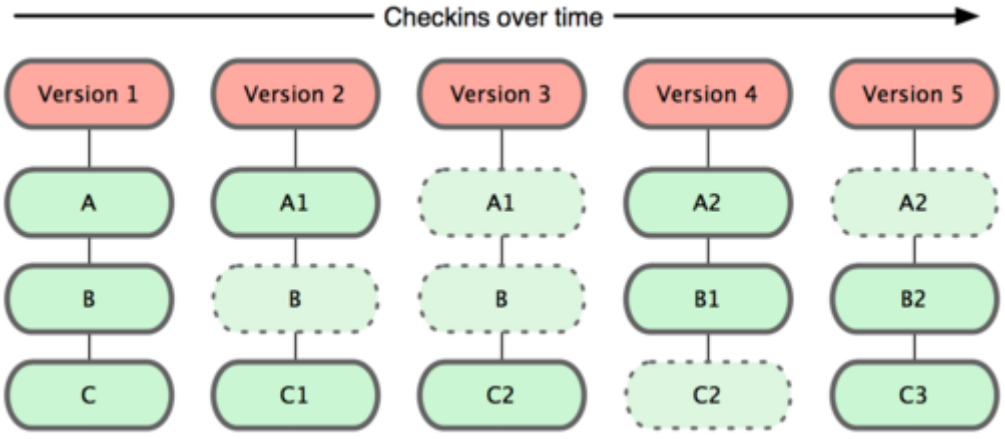
一.起步

1.Git的思想

其他的版本控制工具，大多关心文件内容的具体差异，比如哪些文件发生了更新，这些文件的第几行发生更新。如下图所示：



而Git与它们不同，Git关心的是一份文件的整体是否发生了变化。当我们commit时，只要某个文件发生了变化，Git就会对这份文件做一次快照并生成一个指向这次快照的索引index，要是文件没有变化，Git不会生成新的快照和索引，而是链接向上一次的快照。整个过程如下图所示：



2.索引和指纹

Git判断文件是否发生了变化，就是依赖于索引或是指纹，这是一个由特定算法计算出来的hash值，使用这种方式，可以保证对文件的任何改动都会被侦

测到。

3.文件的3种状态

任何一份文件在Git内都只有三种状态，已提交(committed)，已修改(modified)，已暂存(staged)。以我现在对git的了解，我如果新create了一份文件，那这份文件就是Untracked files即没有纳入Git控制，首先必须add使Git管理这份文件。

如果我们对某份文件进行修改，那这份文件就是modified状态，可以使用git add . 的命令使它们进入staged状态，保存进暂存区域中。如果后续有了git commit命令进行更新提交，这时就不仅仅是暂存区域，而是保存到本地的git仓库中，就是commited状态了。

二.Git基础

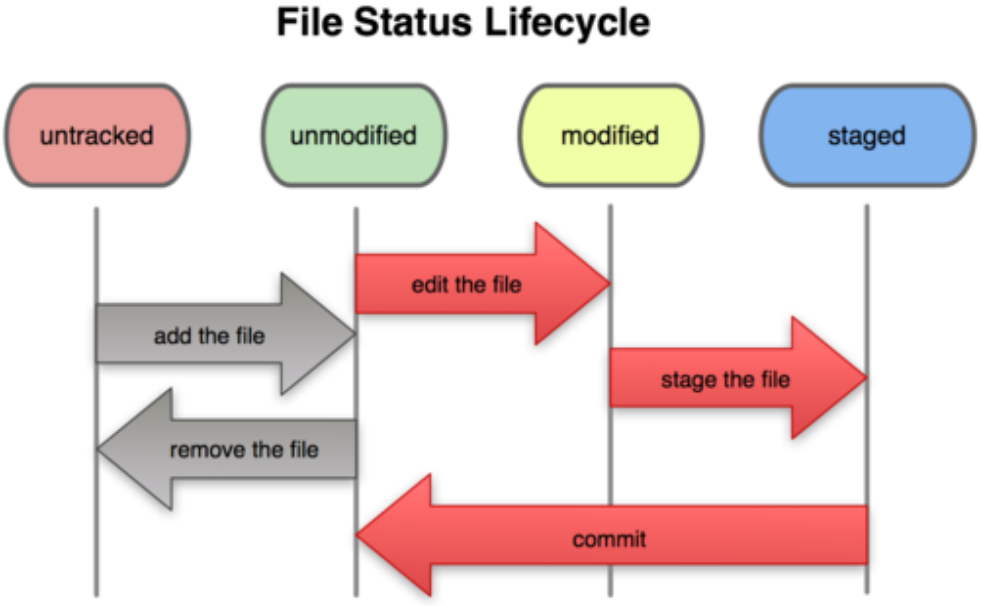
1.建立Git项目仓库

直接前往项目目录，在命令行执行git init即可：

```
git init
```

如果是从github上clone项目的话，就无需初始化仓库了。

2.文件status周期



首先，在git仓库工作目录下，所有文件可以分为两种状态：untracked和tracked，untracked就是这份文件不受git管理，这时一定要使用git add使文件进入tracked状态。

其次，修改文件会使文件进入modified状态，我们需要使用git add命令让这些文件进入staged状态（换言之，把这些文件的快照放入暂存区域，add file into staged area），staged状态下的文件就是下次commit时提交的文件清单。

最后，执行git commit命令，把更改更新到本地仓库，所有文件重新回到unmodifed状态。

3.检查文件状态

```
git status
```

要养成时常执行git status的习惯，在push，pull，rebase等命令之前习惯性的git status一下，保证工作空间是clean状态，再执行接下来的操作。

4.跟踪新文件

```
git add
```

当我们新创建一份文件时，这份文件是untracked状态，即没有加入到git管理中，需要执行git add命令才能受git追踪。值得一提的是git add是一个多功能命令，不仅能跟踪新文件，还能把文件添加进staged暂存区。

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        ignore.html
        src/main/resources/templates/new.html
```

5.暂存已修改文件

```
git add
```

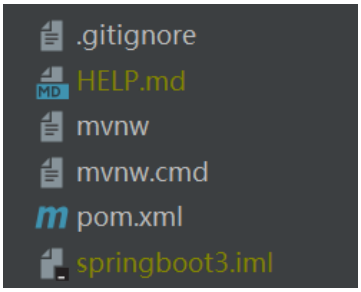
当我们对某一个unmodified文件进行更改时，git会把该文件标记为modified状态，需要执行git add命令让这份文件加入暂存区，以备下次commit时一起提交上去。

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/main/resources/templates/HelloWorld.html
        modified:   src/main/resources/templates/HelloWorld.js
```

6.忽略某些文件

有时我们希望某些文件不受git管控，这时可以在git工作目录下创建一个.gitignore文件，并根据格式排除我们不想要跟踪的文件。事实上无论是springboot项目还是angular项目 都自动帮我们创建好了.gitignore文件并默认排除了一些文件，排除掉的文件在idea中会以浅绿色的颜色标出。



7.提交更新

```
git commit
```

git commit命令会把现在存储在staged暂存区域的文件提交到本地仓库进行更新，并生成一次快照，生成一个index索引，以后可以随时回退到这个位置。

8.查看提交历史

```
git log
git log --2 //查看最近2次commit的情况
```

通过查看历史，获得每一次commit的信息，必要时可以commit返回。

9.撤销操作

```
git commit --amend
```

这个命令我虽然没有通过打字执行，但在idea的git操作中经常使用过，这条命令可以让我在上一次commit的基础上进行修改，再进行一次commit，而上

一次和这一次的commit会合并成为一个commit，有助于减少commit数量，让git进程树变得更加清晰简介。

但是这个命令我曾经不太了解，出了错。必须是上一次commit还没有push的情况下再amend commit才行，如果上一次commit已经push过了，就会出问题的，就不要commit amend了，直接commit即可。

```
git checkout readme.txt
```

这个命令我也用过不少次了，就是普通的撤销对某个文件的修改。

10.远程仓库

通过git clone可以复制一个远程项目到本地，这个就不多说了，这里重点说一下如何把本地的项目推送到远程仓库：

1.首先在github上创建一个仓库；

2.执行命令：

```
git add git remote add origin git@github.com:Jajia/learngit.git
```

这样把本地项目和一个远程仓库关联起来；

3.接下来，如果生成过SSH私钥，进入第4步，如果没有生成SSH公钥的话就要注意了，进入c://user/zheng/.ssh目录下，打开git bash，执行命令：

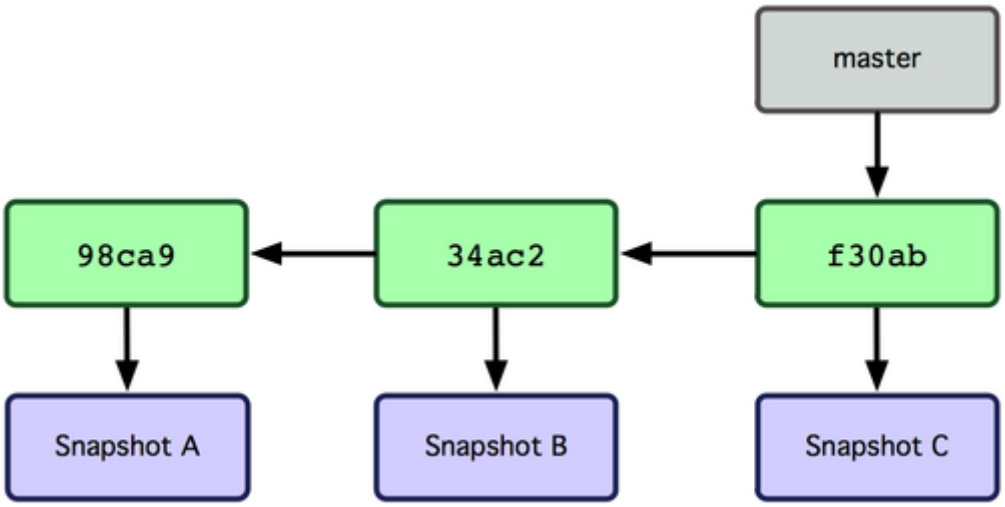
```
$ ssh-keygen -t rsa -C "your_email@example.com" //填写自己的邮箱
```

后面几个问题直接回车即可，看到生成了两个文件，打开pub文件复制里面的私钥。来到github，进入account/setting，左边菜单栏选择SSH，把我们的SSH私钥添加进去

4.接着就可以顺利push了。

三.Git分支

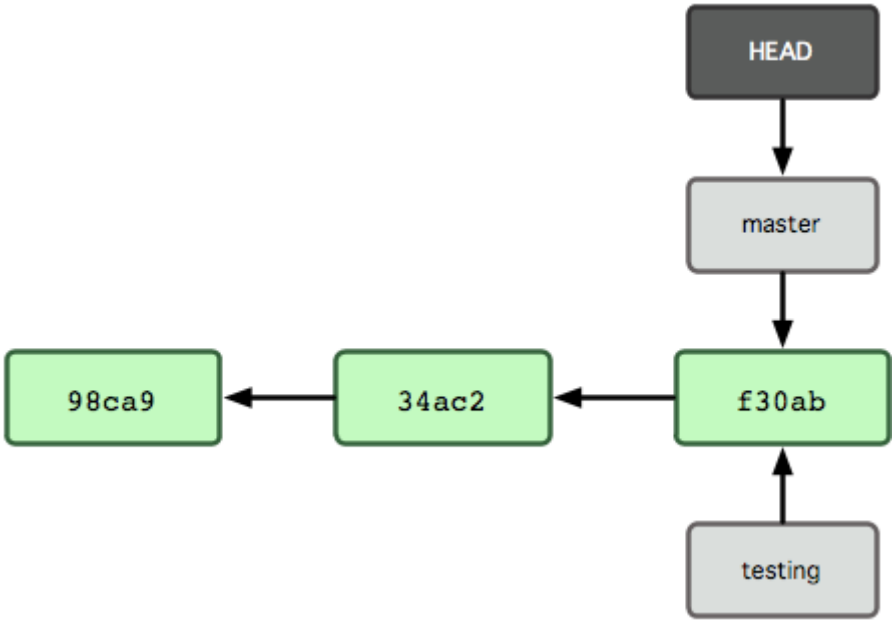
1.Git的指针思想



这一张图可以说明很多事情。我们每一次commit更新，都会创建一个新的commit对象，包含一个快照和索引，方便我们随时返回那一个版本，并且commit对象有一个父类，就是上一次的commit对象。

而所谓的分支，本质上是指向commit对象的指针，一个新的git仓库会默认有一个master分支，随着commit更新，指针也会同步向前移动。

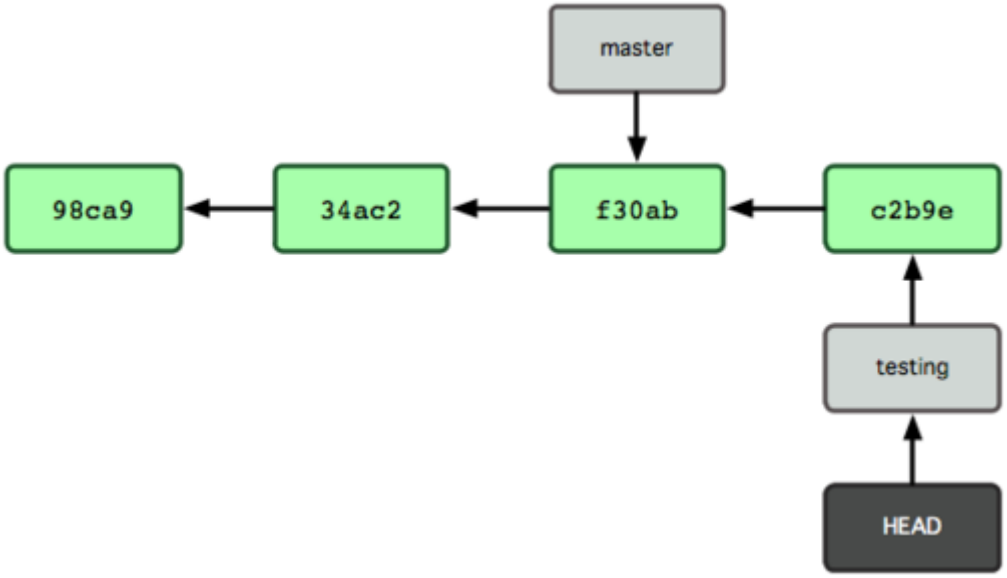
每一个commit对象都指向它的前任commit，当指针指向某一个commit的时候，其实就已经包含了以前所有的历史记录。



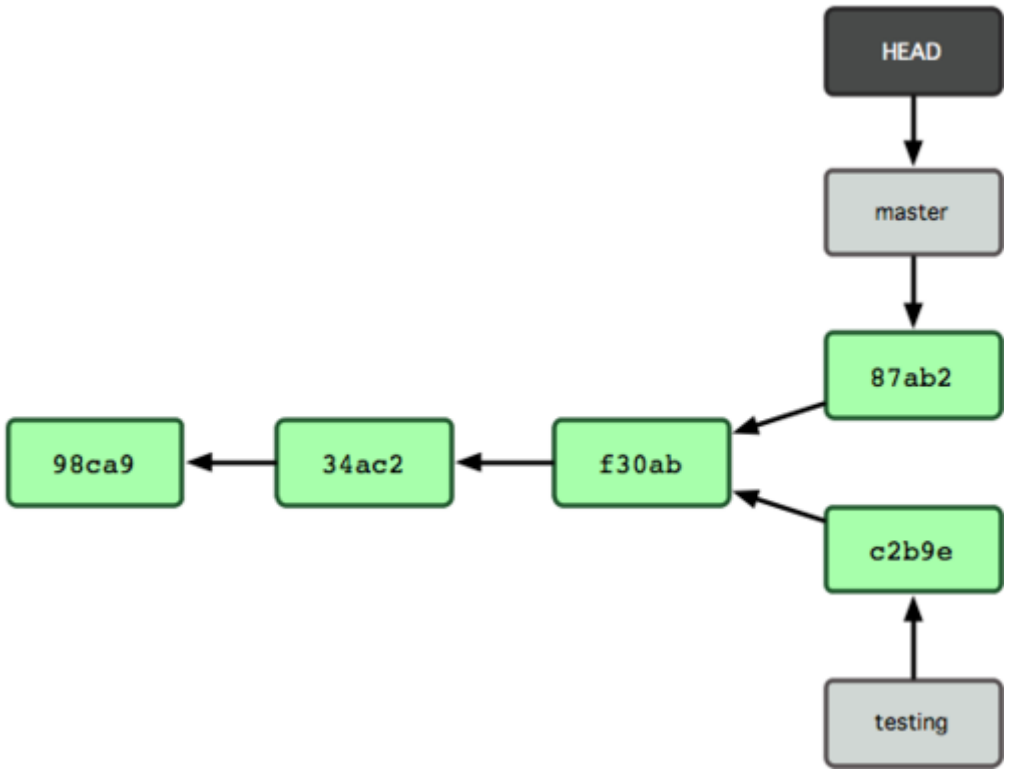
当我们创建一个新的分支时，就会在当前commit对象上新建一个分支指针。但是新创建分支并不会改变header指针的位置，header依然指向了原来的分支指针，换言之我们依然处于master分支，需要使用checkout命令转换分支。

```
git branch testing //创建testing分支，但不会切换分支
git checkout testing //切换到testing分支
```

```
3 git checkout -b testing //新建并切换到testing分支
```



我们在分支上修改文件，提交，testing分支前进了一步，但是master分支还停留在原地，我们checkout到master分支就会发现，在testing分支的改动不会对master产生影响。



这时如果我们在master分支上进行了修改和更新，就会出现分叉现象，如图所示，testing分支是从master分支处的print hello world处切出来的，之后testing分支和master分支各出现一次commit，就看到了分叉现象。

2.短期分支工作流程

1.我们正在feature1分支上工作，这时突然来了一个bug需要fix，首先需要把feature1分支上的改动先暂存起来，可以使用git stash分支做一个暂存：

```
git stash
```

2.接着切换到master分支

```
git checkout master
```

3.新创建一个fix1分支并切换到fix1

```
git checkout -b fix1
```

4.在fix1上工作并提交

5.回到master分支做合并

```
git merge fix1
```

6.这样就完成任务，fix1可以删除掉了

```
git branch -d fix1
```

7.然后可以切换回到feature1上继续工作

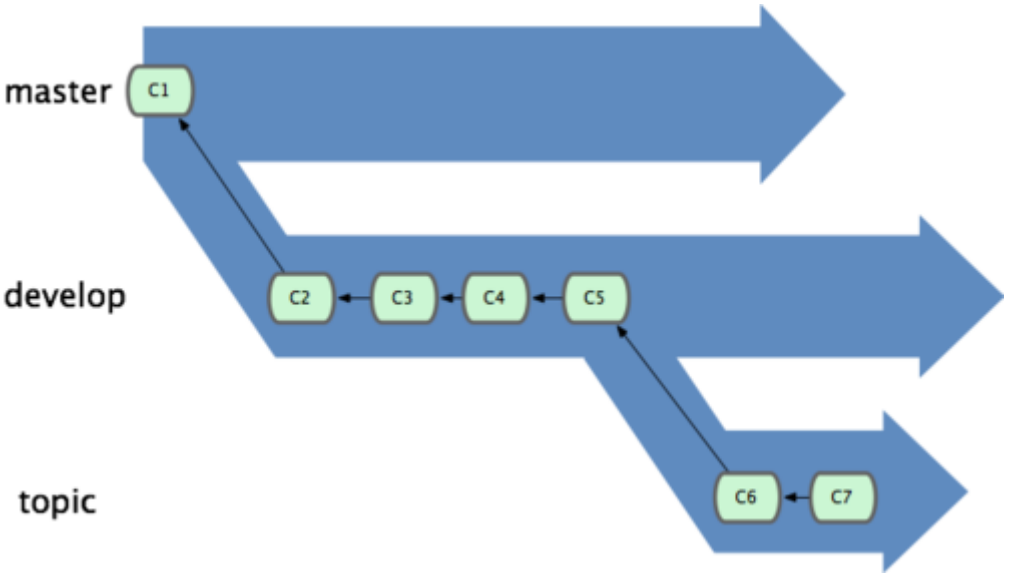
8.如果两个分支操作了同一个文件，就需要解决冲突，具体解决看实际情况，这里说一下冲突解决后git的操作。

冲突发生在branch的merge阶段，解决冲突必然产生文件的更改，需要进行一次commit，这个commit其实就是完成merge的一个步骤，所以message里可以写merge。

3.长期分支工作流程

我们在cloudai的项目就是采用这种流程：

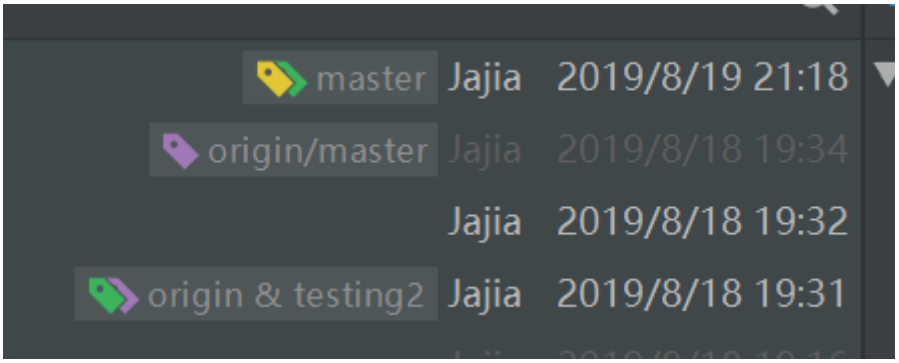
即有一个master分支只用来发布稳定的版本，另有一个develop分支专门用于开发，而在develop的基础上不断切出feature来实现各种各样的功能，合并到develop，到了一个阶段后从develop合并到master进行发布。



这张图可以很清晰的看到git分支的长期工作流程，将来我自己的项目也是走这条路了。

4.远程分支

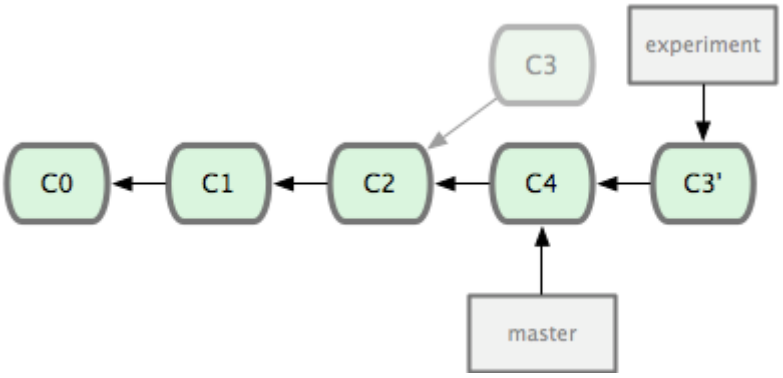
关于远程分支，先放一张图，简单解读一下，master是本地的master分支，origin/master是远程仓库的master分支，可以看见本地在上一次push远程之后我在本地又做了一次commit，所以本地比远程领先一个commit。下面有一个origin&testing2，这个意思是本地和远程各有一个testing2分支并且两者进度是一样的。



看了看progit，其实这一块讲的并不多，就是pull和push就能搞定了。

5.rebase分支

rebase的最终目的是让两个分支做合并操作，合并可以使用之前学过的merge操作，但如果使用rebase，会看到一个更加整洁的log进程，就像一条线一样。



rebase的原理简单来说就是根据你要rebase的那一条分支的所有commit对象，生成一系列的补丁文件，在你rebase基地分支（C4）为起点，逐步打上补丁文件。

但是rebase和之前的commit amend一样，commit amend不能对之前已经push过的commit做commit合并，而rebase则不能对已经push过的分支进行rebase。

这里有几个简单而正确的原则：

1.下游分支更新上游分支的内容，使用rebase，例如feature更新develop上的内容；

```
git rebase develop;
```

2.上游分支合并下游分支的内容，使用merge，例如develop合并origin/testing1的内容；

```
git merge origin/testing2
```

3.merge后记得删除远程分支，使用命令：

```
git push origin --delete testing2
```