5、分支合并冲突853769620

常见于这样的一种场景，就是张三提交了一块代码，push到了远程仓库

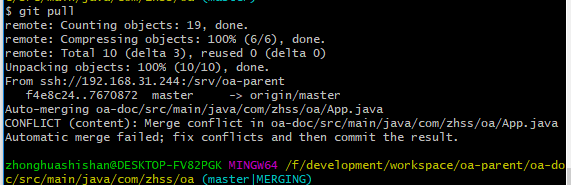
李四此时要push发现不行，要先git pull，会发现说跟张三的master分支代码合并的时候，很容易出现合并冲突

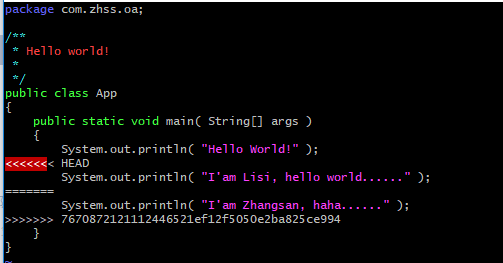
张三修改了一个代码的一行，然后李四也修改了这一行，会发现，俩人修改了同一行，此时应该听谁的呢？

所以说，这个时候就需要进行冲突的解决

李四和张三俩哥儿们会商量一下，看一下这个块儿代码，是怎么回事，为什么出现了冲突

一般来说尽量避免冲突，让不同的人负责不同的独立的模块对应的工程，大家修改的代码都是不一样的，从根本上避免代码出现冲突





出现冲突的代码，会出现上面这样乱七八糟的情况

但是大家不要着急，我们是可以看懂的

<<<<<<< HEAD

    System.out.println("I'm Lisi, hello wolrd......");

======

    System.out.println("I'am Zhangsan, haha......");

>>>>>> 7670872121112446521ef12f5050e2ba825ce994

这里的HEAD指的意思是说，李四本地的HEAD指针指向的分支对应的commit中的代码

7670872121112446521ef12f5050e2ba825ce994，指的是张三修改后提交的commit对应的hash值

所以这里就要解决冲突了啊，张三和李四一合计，张三说，哥儿们，行，那要不就用你的代码吧

就是要删除张三的代码，然后删除特殊符号，即可解决冲突

你每次解决冲突之后，会形成一个新的版本的代码，就是合并了两个commit并且解决冲突之后的代码，就是一个新的commit

（1）分支原理再次回顾

分支管理，是非常重要的

因为一般来说，都是多个人在协作开发，每个人可能做一个功能，这个时候每人都会并行开发一块代码。那么就需要为每个功能创建一个代码分支，那个分支有完整而且独立的代码，每个人在自己的分支代码上开发，不影响任何其他人。最后都开发完了，再将几个分支的代码合并在一起。

每个代码分支，都会有多次提交，然后每个代码分支都会处于最新一次提交的代码，这个分支的多次commit，就会形成一条时间轴一样的东西。

默认情况下，git给我们创建一个master分支，每次提交都会提交到master分支。

而master分支是指向最新的一次commit的，然后HEAD指针是指向master分支的，是这么一个关系。每次提交到master，master分支就会新增一个commit，然后master是指向最新一次commit的，同时HEAD又始终指向master。画图说明。

此时如果我们从master分支创建一个其他的分支出来，那么此时git会创建一个指针，叫做dev，这个dev是指向master分支最新一次commit的，然后HEAD指针会指向dev指针。此时如果我们再提交代码，就是提交到dev分支了，此时commit时间轴会延长一个节点，同时dev指针指向最新的commit，HEAD依然指向dev。画图说明。

git checkout -b dev，这个命令就是创建dev分支，dev指针指向最新一个commit，同时HEAD指针指向dev指针

git checkout -b，相当于两个命令，git branch dev，git checkout dev，先创建分支，再切换到分支

接着用git branch命令，可以查看当前的所有分支以及我们目前所处的分支

接着我们可以在dev分支上修改代码，在HelloWorld.java中增加一行：System.out.println(“I like maven also.”)，然后提交，这时commit时间轴就会长出来一个新的节点，同时dev指针指向最新commit。

这个时候master指针是指向上一个ccommit的，而且还没有做过任何修改，此时如果将dev分支合并到master分支，就是直接将master指针指向最新的一个commit而已，接着将HEAD指针重新指向master指针。画图说明。

用git checkout master，可以切换回master分支，这时会看到上一次commit的代码，因为commit还指向上一个commit。

用git merge dev，将dev分支合并到master分支，这个时候，master指针指向最新commit，HEAD重新指向master指针。

此时代码就是最新一次commit的代码了。

上面这种合并方式，实际上master是没有过任何修改的，合并dev到master，只不过相当于让master分支快进到dev分支指向的commit而已，就是fast-forward模式的merge。

合并完分支之后，可以删除dev分支，此时相当于就是删除dev这个指针而已。画图说明。

git branch -d dev，可以删除dev分支，此时就会删除dev这个指针了。

（2）解决分支冲突

如果在拉取了两个分支，然后两个分支并行开发，接着其中一个分支先合并到了master，相当于master分支的内容已经修改了，接着另外一个分支再合并到master，此时可能会出现，两个分支对同一行代码都做了修改，就会出现代码冲突问题！

git checkout -b feature1

git checkout -b feature2

创建两个功能分支，如上面的命令

在feature1分支将最后一行修改为System.out.println(“I like spark......”)，提交代码

在feature2分支将最后一行修改为System.out.println(“I like storm......”)，提交代码

这个时候的情况，在之前的最新commit，拉出来了两个并行的commit，同时两个分支的指针分别指向对应的commit，画图说明。

接着切换到master分支，将feature1先合并到master分支，那么此时master指针就会指向feature1指向的那个commit，同时HEAD指向master指针。画图说明。

接着继续将feature2合并到master分支，此时，因为master分支是无法执行fast-forward快进操作的，因为master也是有修改的，此时只能执行3-way merge。而且执行git merge feature2命令后，会提示出现了confiict冲突，让我们先解决冲突。

此时我们可以查看HelloWorld.java代码，发现其中最后一行变成了这个样子

<<<<<<< HEAD

System.out.println(“I like spark......”);

=======

System.out.println(“I liike storm......”);

>>>>>>> feature2

上面的意思，就是说master分支和feature2分支出现了冲突，第一行是HEAD代码，因为HEAD目前指向master，也就是master的代码；第二行是feature2的代码。

这个时候，我们可以将冲突的内容删除掉，保留正常内容即可。

接着提交HelloWorld.java代码，就解决了冲突并且提交了代码，同时完成了feature2到master的合并。此时会自动长出来一个新的commit节点，这个commit就是解决了master和feature2冲突并且合并之后的commit。同时master指向最新的这个commit，HEAD还是指向master。画图说明。

最后删除feature1和feature2两个分支，画图说明。