这一讲，非常非常非常的重要853769620

因为在git学习中，我觉得最重要的是两块：

第一块，就是前面的10讲，里面讲解了3个区域，6种状态，提交历史，等等；

第二块，分支（背后的原理，合并的原理，解决冲突的原理，远程分支的原理）

全部放在这一讲，用一个比较经典的多人协作开发模型，就是功能分支工作流，实战多人去开发OA系统的代码，然后演示整个分支是怎么玩儿的，每做一步分支的操作，幕后的原理图，对应着画出来

1、分支的基础知识

（1）分支的介绍

所谓的分支功能，就是可以同时拉出来多个代码副本，然后在不同的代码副本上，可以进行对应功能的开发。完成开发之后，可以将多个分支合并在一起，形成最终的代码。

分支，其实就是指针

在git中，每一个项目，不管你有多少个分支，不管你在哪个分支上开发

最终都会形成一个完整的提交历史，树形结构

每个分支，其实就只是一个指针而已，分支就指向了提交历史中的某个commit object

每个commit object就代表了这个项目的所有代码在那次提交的时候一个完整的快照版本，包含了之前没有变更的代码文件，也包括了这次提交的最新的修改/新增/删除的代码文件

每个commit object就代表了项目的一个版本

所以你的分支实际上就是指向了历史上某个时刻的一个版本的代码，就是一个commit object

Git的分支功能是一个很大的特色，非常的轻量级，分支的来回切换速度是很快的，在实际的开发过程中，分支绝对是最重要的功能。

（2）commit object与分支的原理

Git并不是存储一系列的文件差异，而是存储一系列的文件快照的。实际上每次我们执行一次commit，git都会存储一个commit object，这个commit object中会包含一个指针，指向这次提交文件的快照。这个commit object同时也包含作者的姓名和邮箱，提交说明，以及对上一次commit object的指针。

将一个文件版本放入暂存区的时候，就会计算一个校验和，然后提交的时候会将文件内容以blob的方式放入版本库中，同时在暂存区放入这个文件版本的校验和。接着git会创建一个commit object，其中会包含元数据，以及一个指针指向版本库中的文件快照。

也就是说，每次执行一次提交，都会在版本库中包含这么几个东西：一个blob，每个文件都会有一个blob来存储这个文件的本次提交的快照；一个tree，这个tree会包含对本次提交的所有文件的blob的指针；一个commt object，指向了tree的指针，作者，等信息。

接着如果再次执行一个提交，那么下一个提交同样会包含那些东西：每个文件一个blob，一个tree指向所有blob，一个commit object指向那个tree，同时这个commit object会有一个指针，指向上一个commit object。

最后多次提交，就会得到一颗完整的commit树。

分支是啥？分支就是一个轻量级的指针，默认的分支是master，那么每次提交，master分支的指针默认就是指向最新的那个commit object的。每次提交一次，master指针就会挪动，继续指向最新的commit object。

同时如果你当前工作在某个分支上，比如工作在master分支上，那么git还维护了一个特殊的指针，HEAD，这个指针指向master分支指针。如果你创建了其他的分支，那么其他的分支也会指向某个commit object，而且此时如果工作在那个分支上，那么HEAD指针会指向那个分支。

（3）创建一个分支

git branch testing，可以创建一个新的分支，此时这个分支的指针会指向当前你所在的分支所指向的commit object上。

如何查看各个分支指向哪个commit object呢？使用git log --oneline --decorate命令即可，会给你显示出来。

（4）切换分支

git checkout testing，此时就可以切换到testing分支，此时HEAD指针会指向testing分支指针。

此时如果在testing分支上提交代码，那么会commit树会长出来一个新的commit object，而testing分支指正会指向最新的commit object，HEAD继续指向testing分支指针，而master指针还是指向之前的那个commit object。

git checkout master，会切换回master分支，此时HEAD指针会指向master指针，同时将master指针指向的那个commit object，对应的tree和其中的blob，也就是对应的文件快照恢复到工作区中来。

再次在master分支上提交一个修改，此时从这个commit object会再长出来一个新的commit object，看起来就是跟testing分支当前指向的commit object形成了两个分叉。

此时如果要查看commit树，可以用命令：git log --oneline --decorate --graph --all，这个命令很有用，可以打印出整颗commit树，同时告诉你各个分支当前指向哪个commit object。

在Git中，分支升级上就是一个很简单的文件，其中包含了一个40位的SHA-1校验和，就是分支指向的那个commit object的SHA-1。所以创建分支的代价是很低的，不过就是创建这么一个文件罢了。而一些集中式版本控制系统，对于分支可能需要拷贝一个完整的文件副本，导致速度很慢，磁盘空间占用很大。

（5）远程分支

在本地你可以创建一个分支，然后开发代码，但是后续的话，你肯定是要将本地分支推送到远程仓库里面去的

git push -u origin 分支名称

这个的意思，就是将本地分支推送到远程仓库里，形成一个同名的远程分支，同时-u这个选项就是将本地分支和远程分支关联起来

下一次如果修改了这个分支的代码，直接执行git push origin 分支名称，就直接可以将本地分支的代码推送到远程分支

如果其他人要将某个远程仓库的分支拉取下来，应该执行一个命令，叫做git fetch origin，就会抓取下来远程仓库新增了哪些分支

接下来可以执行一个命令，git checkout -b 本地分支 origin/远程分支，用这个命令，origin/远程分支，就代表了远程仓库里的那个分支，然后这个命令一执行，就是在本地创建一个远程分支对应的本地分支，互相关联起来

以后呢，每次如果别人更新了那个分支的代码，push到了远程仓库，你可以执行git pull命令，将这个分支在远程仓库的代码拉取下来，跟本地分支的代码进行合并

远程版本库的分支，在本地都有追踪分支，remote-tracking 分支。

比如说本地的master分支，对应的远程分支就是origin/master。比如本地的feature/iss53分支，对应的远程分支就是origin/feature/iss53。

假设我们公司的git服务器地址是git.zhss.com，然后如果我们用git clone命令从这个git服务器克隆了一个版本库下来，git默认会将远程版本库命名为origin，同时在本地创建一个指向远程版本库的master分支的本地分支，叫做origin/master。此外，git也会给你在本地创建一个master分支，内容就是跟origin/master分支一样的。

从git服务器克隆版本库下来的时候，远程版本库的commit树会一同拷贝下来，然后origin/master指向的commit，就是远程版本库的master指向的commit，同时给本地创建的commit也是指向这个commit。

此时，如果在本地你做了不少开发，然后本地master移动了好几个commit，同时origin/master还是指向最开始的那个commit；而同时，远程版本库上，其他同事也提交了几次代码，因此远程版本库上的commit也移动了几个commit。

此时如果要让本地和远程保持同步，需要使用git fetch origin命令，该命令会将远程版本库的commit树和所有的分支都拉取下来，跟本地的commit树进行合并，此时可能就会在本地形成一棵有两个分叉的commit树。

本地的origin/master会指向远程版本库的master指向的那个commit，本地的master继续指向之前本地最新的那个commit。

使用git push origin serverfix，可以将你在本地的分支推送到远程版本库上去，此时你本地的commit树也会推送到远程版本库，跟远程版本库的commit树进行合并。

下一次别人执行git fetch origin的时候，就会获取到一个remote -tracking分支，origin/serverfix分支，这个远程分支是只读的。接着你使用git checkout -b serverfix origin/serverfix命令，就可以在本地获取到一个分支跟origin/serverfix关联起来，然后就可以跟你一起对一个分支进行修改了。

使用git checkout -b serverfix origin/serverfix命令，创建出来的本地分支叫做tracking brach，而这个本地分支track的远程分支叫做upstream branch。此时本地分支就会track那个远程分支，跟远程分支之间会建立关联关系。

如果我们在tracking分支上，执行git pull命令，git就会自动将对应的远程分支在远程版本库上的代码拉取下来跟本地的tracking分支做合并，如果有冲突的话，还需要解决冲突。

如果我们是使用clone命令克隆的远程版本库，那么默认就会将本地的master分支创建为追踪origin/master远程分支的。

使用git branch -u origin/serverfix，可以让当前本地分支track某个指定的远程分支。

使用git branch -vv，可以查看每个本地分支track的远程分支。

git fetch origin命令，仅仅就是将远程版本库的commit树拉取下来，同时拉取下来所有最新的远程分支，给我们使用。

如果我们已经建立起来了本地分支和远程分支的track关系，那么就可以直接使用git pull命令，将远程分支的代码拉取下来合并到本地分支。但是通常来说，我们也可以先git fetch origin，然后git merge 远程分支，一样可以将远程分支代码合并到本地分支上去。

删除远程分支，比如一个远程分支，已经结束了工作了，可以删除一个远程分支，使用git push origin --delete serverfix即可。同时可以使用git branch -d serverfix删除本地分支。

2、功能分支工作流介绍

工作流，git工作流，简单来说，就是多个人如何同时基于git远程仓库，加上分支，去非常良好的协作开发，包括去执行各种集成测试，QA测试，预发布测试，生产环境上线

第一个工作流，功能分支工作流，比较适合用于5人左右或者以内的小团队

集中式工作流、功能分支工作流、GitFlow工作流、互联网公司一种GitFlow工作流的变种

每种工作流都没有绝对的好坏，主要是适用于某个场景下，某种类型的项目

一般来说，都会准备一个master分支，作为你的稳定分支，这里的代码是随时可以上线的；有多个feature分支，每个feature分支用来开发一个功能。

如果线上有bug，一般分为两种，一种是影响正常用户使用流程的紧急bug，就是hotfix，比如说电商网站无法下单了，门户网站无法查看新闻了；一种是不影响正常用户使用流程的非紧急bug，比如说有某个广告显示的位置出现了偏差，就是bugfix。

一般线上发现了bug，就了一个hotfix或者bugfix分支，然后在分支上复现bug，修复bug，然后在测试环境进行验证以及回归测试，最后将分支合并到master去上线，修复线上问题。

这边，我根据过往的经验，给大家说一下，我觉得这个工作流比较适合什么样的项目？

非常适合对公司内部的系统、平台或者工具，5人以内小团队，3人左右，去开发一个比如公司内使用工具，报表平台

报表平台，不是什么核心的系统，可能就是投入两三个人，就维护一个系统，各种在里面加入功能，满足大家看报表的需求，平时有bug就修复一下。

新的需求频率也不是很高，可能就是比较稳定的新需求一步一步提出来，大家就比较稳的在哪儿维护这个报表平台就可以了

分布式爬虫系统，3个人在搞，也是这样子，就是3个人开发和维护一个小爬虫的系统，然后有需要就在里面加入功能，有bug就修复

没有什么版本的概念，对公司内部，1.0，1.1.2，1.5.3，不需要这么复杂的版本

就是就很简单，就是一个系统，比如说，你的需求方，某个看数据的运营同学，或者是产品经理同学说，好，现在你的报表系统能不能给我显示一个折线图，因为之前只能是柱状图

然后你说，好，没问题，拉一个feature分支下来，feature/line\_graph，开发开发，测试，最后feature/line\_graph合并到master分支，上线。。。

然后平时有人突然说，饼状图显示不出来了，有bug，bugfix/bar\_graph\_cannot\_show，复现，修复，测试，最后bugfix/bar\_graph\_cannot\_show合并到master分支，上线

3、功能分支工作流实战

我们来按照真实的网站开发过程体验一下分支的使用和合并过程：

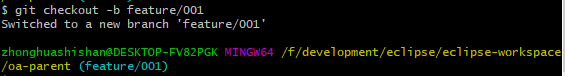
（1）开发一个OA系统，俩人，张三和李四

张三：/f/development/eclipse/eclipse-workspace/oa-parent

李四：/f/development/workspace/oa-parent

（2）张三从master分支上，拉一个新的feature分支，叫做feature/001，然后在分支上做一些开发，开发新功能；李四从master分支上，拉一个新的feature分支，叫做feature/002，然后在分支上做一些开发

拉一个分支下来：git checkout -b feature/001，就会从当前分支拉一个分支出来



比如你现在在master分支上，然后你执行了git checkout -b feature/001这个命令，此时会创建一个新的分支对应的指针，feature/001，指向了master当前指向的那个commit object，同时如果切换到了feature/001分支，HEAD会指向feature/001

git为什么要这么玩儿？

git不会说，每次创建一个分支，就直接拷贝一份代码，效率太低了

在git中，整个所有的版本和文件只存储一次，只存储一份，提交历史，commit树，只有一份

每个commit object就代表了这个项目在当前时刻一个完整的快照版本，代表了一份完整的代码了

所以创建分支，就是创建一个轻量级的指针而已，指向了某个commit object，你的当前所处的分支指向哪个commit object，就是指向了哪个版本的代码，你的工作区中的代码就是那个版本的代码

李四：git checkout -b feature/002

张三进行开发提交了两次，oa-auth里面加入新功能

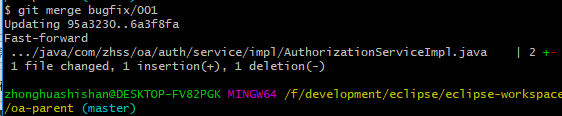
李四进行开发提交了一次，oa-doc里面加入新功能

（3）这个时候张三接到一个通知，有一个bug，需要进行hotfix。这个时候我们需要切换到master分支上，拉一个hotfix分支，拉bugfix/001。在hotfix分支上修复bug，然后测试和验证一下，合并到master分支，用master分支代码部署，解决线上bug。删除bugfix/001分支

张三：git checkout master，git checkout -b bugfix/001，修改bug，提交，再将bugfix/001分支合并到master分支

如果你要git checkout master，首先需要将当前分支上修改的代码全部git add然后git commit，才能切换分支

合并分支：切换到master分支，然后执行git merge bugfix/001



这里执行的是一种merge，叫做fast-forward，快进式merge

我们此时就不需要bugfix/001分支了？为啥，因为master分支就指向了那个commit了，要bugfix/001分支还有何用？

给删除掉bugfix/001分支：git branch -d bugfix/001

（4）张三继续切换回feature/001分支，继续开发工作，开发完之后合并到master，删除feature/001分支



3-way merge，三路合并

还没有完，张三还需要将feature/001分支删除

（5）李四同时抑制在开发feature/002分支，完成之后合并到master，删除feature/002分支

接着我们从Git分支的角度来说一下上面整个实验步骤和内幕原理：

首先，假设我们的master分支已经走过了几个commit了，当前master指向的是最新的一个commit。比如有三个commit object，c0，c1，c2，而master指向的就是c2，HEAD指向master。

此时我们需要开发一个新的功能，可以从master分支拉一个feature/iss53分支

git checkout -b feature/iss53，这个命令是在创建分支的同时也切换到这个分支上去，相当于是git branch feature/iss53，git checkout feature/iss53

此时master继续指向c2，但是多了一个feature/iss53指针，指向c2，同时HEAD切换为指向feature/iss53了。

接着我们就可以在feature/iss53分支上做一些开发了，主要是为了新功能写代码，而且一般每天都会执行一次提交，保证工作内容不会丢失，这个过程之后，c2 commit之后会长出来一个新的c3 commit，然后feature/iss53指向了这个c3，HEAD继续指向feature/iss53。

此时接到一个通知，需要解决一个线上的bug。此时就需要切换到master分支，然后拉一个新的hotfix分支。但是在切换到master分支之前，是需要先提交你的feature/iss53的所有代码的，否则是不允许切换分支的。

git checkout master

git checkout -b hotfix/bug61

一旦切换到master分支，工作区的所有代码会给git还原为master指针指向的那个commit object的所有快照，git会负责对代码文件进行新增、修改或者删除，彻底还原到那个快照版本。

接着拉取并且切换到hotfix/bug61之后，我们就可以基于master的代码来进行bug修复了，修改代码，解决bug，进行测试，然后在hotfix/bug61上可以提交代码。这里注意一点，可以让hotfix/bug61分支的代码和feature/iss53分支的代码，都对同一行进行不同的修改，为了后面的合并冲突解决进行准备。

这个时候，master继续指向c2，feature/iss53指向c3，然后从c2长出来一个新的c4 commit object，hotfix/bug61指向c4。也就是说，从c2之后，分叉长出来两个commit object，c3和c4。

接着，对hotfix/bug61的代码需要进行测试，然后测试通过之后，就可以将hotfix/bug61分支合并到master分支，准备上线了。

git checkout master

git merge hotfix/bug61

这次merge被称之为fast-forward，快进式的merge，为啥呢？因为看一下此时的commit树，master指针只要直接移动到指向c4，就可以完成本次合并了，这就是fast-forward。但是此时HEAD就是指向master的。

此时就可以用master代码来部署解决bug了。

同时，我们已经不再需要hotfix/bug61这个分支了，因为已经合并到master分支了，此时可以删除hotfix/bug61这个分支。

git branch -d hotfix

然后切换回feature/iss53分支，继续你的新功能的开发：git checkout feature/iss53

完成一些代码，然后最终commit一次，结束feature/iss53的代码开发，同时基于这个分支的代码完成测试。此时会从c3长出来一个c5，feature/iss53指向c5，而且HEAD指向feature/iss53。

接着就要将feature/iss53分支合并到master分支了，完成功能的合并

git checkout master

git merge feature/iss53

此时，因为master指向的commit，是不能直接移动到feature/iss53指向的commit的，因此需要执行一个3-way merge。此时会创建一个新的commit object，叫做c6，然后父commit是两个，c4和c5。此时，master会指向最新的c6，feature/iss53继续指向c5，同时HEAD指向master。

接着就可以删除feature/iss53分支了，git branch -d feature/iss53

但是合并的时候有可能会出现冲突，就是可能两个分支对同一行代码的修改不一样，此时会提示有CONFLICT，然后automatic merge failed，需要我们去解决冲突。

此时，我们首先需要执行git status命令，来看一下哪些代码中出现了冲突，会提示你unmerged paths，里面有both modified的一些文件。

接着进入那些文件，开始解决冲突

会提示你HEAD中的某一行是什么样子的，然后要合并进HEAD的其他分支的这一行是什么样的。此时你需要自己决定该如何修复冲突。

修复冲突之后，需要执行git add命令，将这些修改后的文件放入暂存区，接着执行git commit命令，来提交冲突解决的修改。此时就会继续创建一个新的commit出来，c6，它包含的就是master和feature/iss53合并以后，同时解决了冲突以后的快照。c6的父节点也是c4和c5。

4、分支管理命令

git branch命令，直接执行，就会显示出当前所有分支列表，以及你在哪个分支上工作

git branch -v命令，可以显示出每个分支当前指向的commit object

git branch --merged，可以看到哪些分支被merge进了当前分支；git branch --no-merged，可以看到哪些分支还没有被merge进当前分支

如果git branch -d命令删除一个分支，可能会提示你那个分支还没merge到当前分支来，不让你删除该分支，此时可以使用git branch -D命令，强制删除一个分支。

5、分支合并冲突

（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。