Lesson4

你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

**创建与合并分支**

因为创建、合并和删除分支非常快，所以Git鼓励你使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

查看分支：git branch

创建分支：git branch <name>

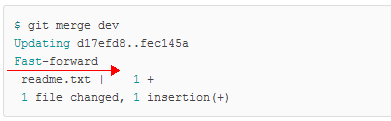
切换分支：git checkout <name>

**创建+切换分支**：git checkout -b <name>

合并某分支到当前分支：git merge <name> （先切换到master分支）

删除分支：git branch -d <name>

注意，合并时，



注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并时，要由帐户所有人同意后，才能合并。

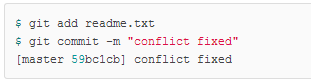
合并后，即可删除分支

**解决冲突**

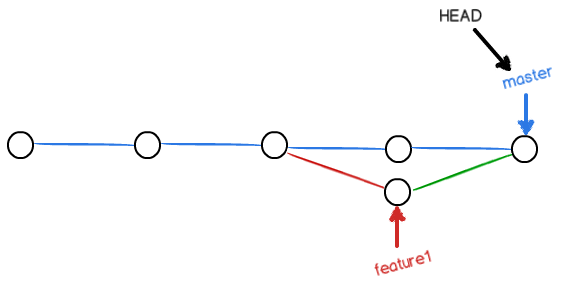
如果在新的branch和master里面，都更新了内容，则无法快速合并，只能试图把各自的修改合并起来。

Git status 可以告诉你冲突的地方，冲突文件中，会标明冲突的具体地方。

修改后，再提交



现在，master分支和feature1分支变成了下图所示：



小结

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

用git log --graph命令可以看到分支合并图。

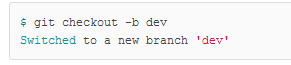
**分支管理策略**

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

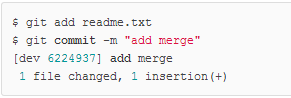
如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

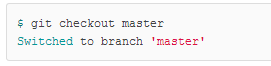
首先，仍然创建并切换dev分支：



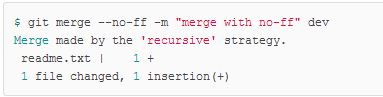
修改readme.txt文件，并提交一个新的commit：



现在，我们切换回master：

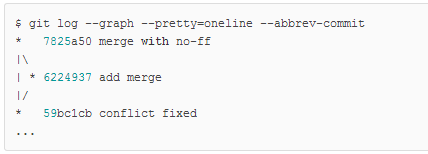


准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：



因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史：



可以看到，不使用Fast forward模式，merge后就像这样：



分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面工作；

工作都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

所以，团队合作的分支看起来就像这样：



小结

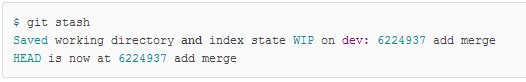
Git分支十分强大，在团队开发中应该充分应用。

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

**BUG分支**

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，若有当前正在dev上进行的工作还没有提交的话，则需要把现场先保存下来。

stash功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

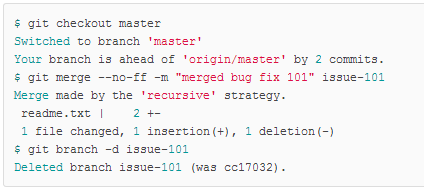


现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：



等修复完Bug, 切换到master分支，并完成合并，最后删除issue-101分支：



现在可以切换到原dev分支继续工作。



工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：



工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：



再用git stash list查看，就看不到任何stash内容了。

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令。

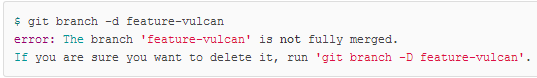
小结

修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。

**Feature分支**

若是想给软件添加新功能，则类似于bug修复。新建分支，然后处理。可是不想要这个新分支了，此时分支还没有合并。



销毁失败。Git友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：



小结

开发一个新feature，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过git branch -D <name>强行删除。

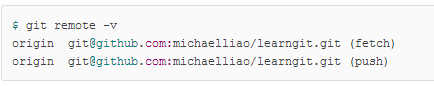
**多人协作**

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：



或者，用git remote -v显示更详细的信息：



上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

推送分支

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：



如果要推送其他分支，比如dev，就改成：



但是，并不是一定要把本地分支往远程推送，那么，哪些分支需要推送，哪些不需要呢？

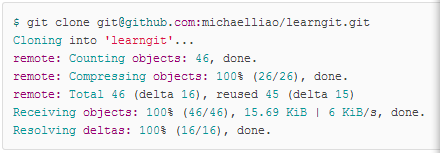
* master分支是主分支，因此要时刻与远程同步；
* dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；
* bug分支只用于在本地修复bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个bug；
* feature分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

### 抓取分支

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：



当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。可以用git branch命令查看：



现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：



现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

