**Git分支、标签管理**

## 创建与合并分支

### 介绍

在版本回退里，你已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长：

当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

不过，从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

### 实践

* 首先，我们创建dev分支，然后切换到dev分支：



git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

* 然后，用git branch命令查看当前分支：

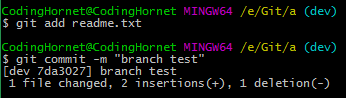


git branch命令会列出所有分支，当前分支前面会标一个\*号。

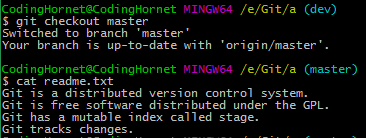
* 然后，我们就可以在dev分支上正常提交，比如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

* 然后提交：



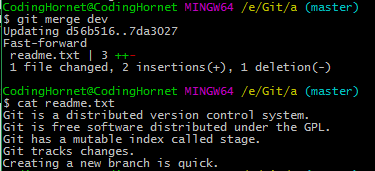
* 现在，dev分支的工作完成，我们就可以切换回master分支：



切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



* 现在，我们把dev分支的工作成果合并到master分支上：



git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

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。

* 合并完成后，就可以放心地删除dev分支了：

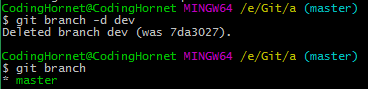
$ git branch -d dev

Deleted branch dev (was fec145a).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master



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。

### 命令

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>

删除分支：git branch -d <name>

## 解决冲突

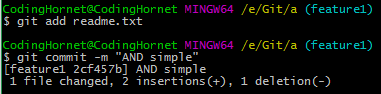
* 准备新的feature1分支，继续我们的新分支开发：



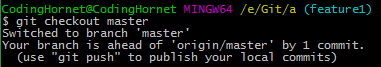
修改readme.txt最后一行，改为：

Creating a new branch is quick AND simple.

* 在feature1分支上提交：



* 切换到master分支：

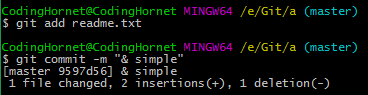


Git还会自动提示我们当前master分支比远程的master分支要超前1个提交。

* 在master分支上把readme.txt文件的最后一行改为：

Creating a new branch is quick & simple.

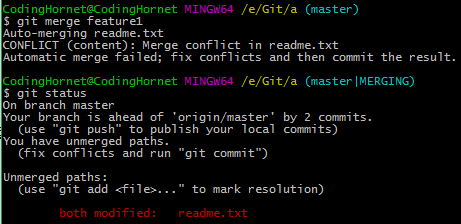
提交：



现在，master分支和feature1分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：



果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

* 我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

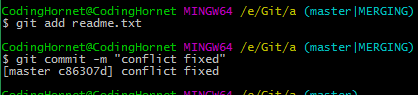
Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

Creating a new branch is quick and simple.

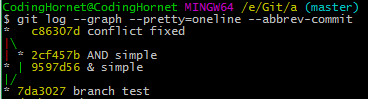
* 再提交：



现在，master分支和feature1分支变成了下图所示：



* 用带参数的git log也可以看到分支的合并情况：



* 最后，删除feature1分支：



工作完成。

**Tips：用git log --graph命令可以看到分支合并图。**

## 分支管理策略

### no-ff

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。

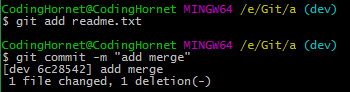
如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

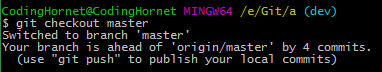
* 首先，仍然创建并切换dev分支：



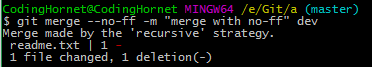
修改readme.txt文件，并提交一个新的commit：



* 现在，我们切换回master：

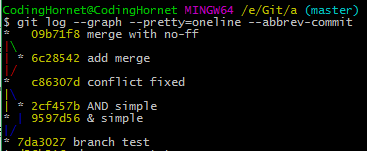


* 准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：



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，平时不能在上面干活；

那在哪干活呢？干活都在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；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

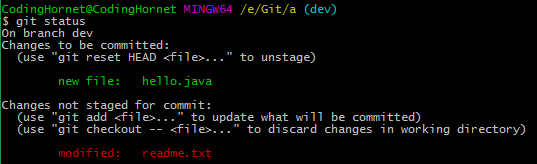
所以，团队合作的分支看起来就像这样：



## Bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以，每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

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，等等，当前正在dev上进行的工作还没有提交：



并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

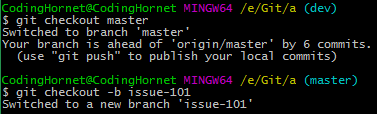
### stash

幸好，Git还提供了一个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：

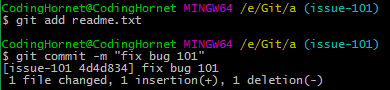


现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

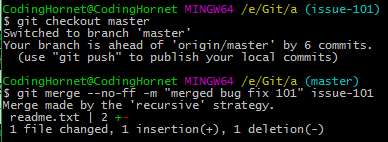
首先确定要在哪个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：



现在修复bug，需要把“Git is free software ...”改为“Git is a free software ...”，然后提交：

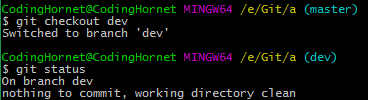


修复完成后，切换到master分支，并完成合并，最后删除issue-101分支：





太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到dev分支干活了！

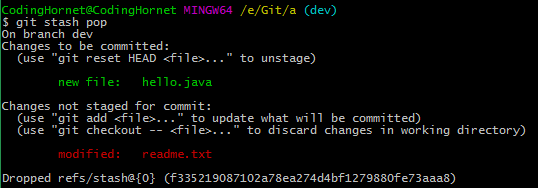


工作区是干净的，刚才的工作现场存到哪去了？用git stash list命令看看：



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：

$ git stash apply stash@{0}

## Feature分支

软件开发中，总有无穷无尽的新的功能要不断添加进来。

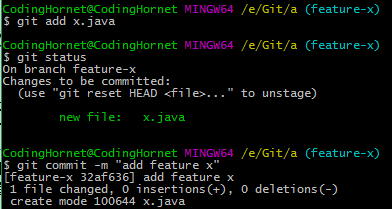
添加一个新功能时，你肯定不希望因为一些实验性质的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在，你终于接到了一个新任务：开发代号为x的新功能，该功能计划用于一下git。

于是准备开发：



5分钟后，开发完毕：



切回dev，准备合并：



一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。

但是，就在此时，接到通知，需求改版了，新功能必须取消！虽然白干了，但是这个分支还是必须就地销毁：



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：

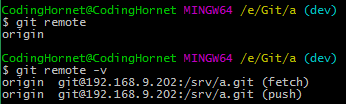


终于删除成功！

## 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：

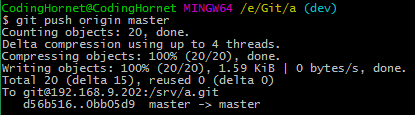


或者，用git remote -v显示更详细的信息：

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：

$ git push origin dev

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；

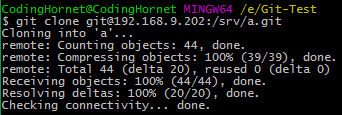
**feature**分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发。

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。

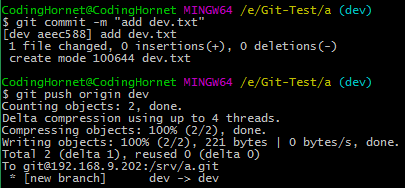
现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：



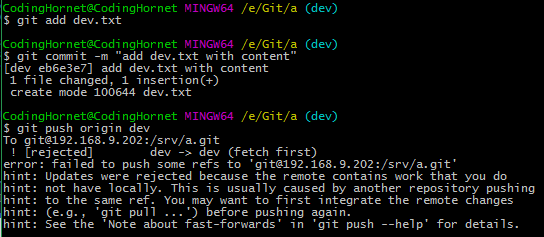
当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。不信可以用git branch命令看看：



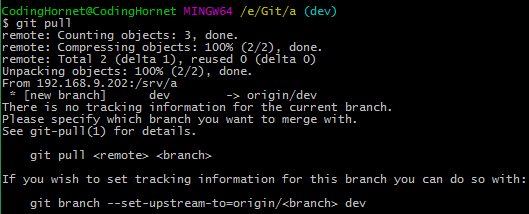
现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，然后，时不时地把dev分支push到远程：



你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件作了修改，并试图推送：



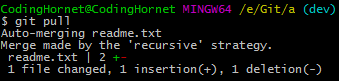
推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，然后，在本地合并，解决冲突，再推送：



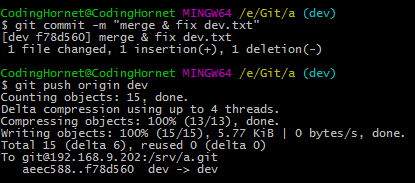
git pull也失败了，原因是没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：



再pull：



这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：



### 工作模式

因此，多人协作的工作模式通常是这样：

* 首先，可以试图用git push origin branch-name推送自己的修改；
* 如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；
* 如果合并有冲突，则解决冲突，并在本地提交；
* 没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！
* 如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

**Tips:**

* 查看远程库信息，使用git remote -v；
* 本地新建的分支如果不推送到远程，对其他人就是不可见的；
* 从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；
* 在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；
* 建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；
* 从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

## 标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的指针（跟分支很像对不对？但是分支可以移动，标签不能移动），所以，创建和删除标签都是瞬间完成的。

Git有commit，为什么还要引入tag？

“请把上周一的那个版本打包发布，commit号是6a5819e...”

“一串乱七八糟的数字不好找！”

如果换一个办法：

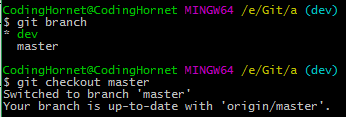
“请把上周一的那个版本打包发布，版本号是v1.2”

“好的，按照tag v1.2查找commit就行！”

所以，tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

### 创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上：



然后，敲命令git tag <name>就可以打一个新标签：

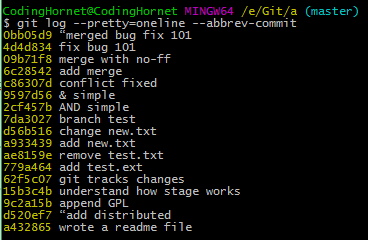


可以用命令git tag查看所有标签：

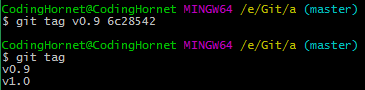


默认标签是打在最新提交的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

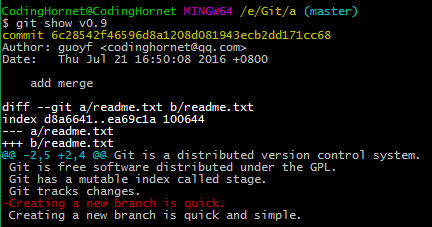
方法是找到历史提交的commit id，然后打上就可以了：



比方说要对add merge这次提交打标签，它对应的commit id是6c28542，敲入命令,再用命令git tag查看标签：



注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show <tagname>查看标签信息：

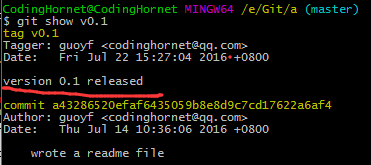


可以看到，v0.9确实打在add merge这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：



用命令git show <tagname>可以看到说明文字：



还可以通过-s用私钥签名一个标签：

$ git tag -s v0.2 -m "signed version 0.2 released" fec145a

签名采用PGP签名，因此，必须首先安装gpg（GnuPG），如果没有找到gpg，或者没有gpg密钥对，就会报错：

gpg: signing failed: secret key not available

error: gpg failed to sign the data

error: unable to sign the tag

如果报错，请参考GnuPG帮助文档配置Key。

用命令git show <tagname>可以看到PGP签名信息：

$ git show v0.2

tag v0.2

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...

-----END PGP SIGNATURE-----

commit fec145accd63cdc9ed95a2f557ea0658a2a6537f

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 10:37:30 2013 +0800

branch test

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。验证签名的方法比较复杂，这里就不介绍了。

### 标签操作

如果标签打错了，也可以删除：



因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin <tagname>：



或者，一次性推送全部尚未推送到远程的本地标签：



如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：



然后，从远程删除。删除命令也是push，但是格式如下：



**Tips:**

* 命令git push origin <tagname>可以推送一个本地标签；
* 命令git push origin --tags可以推送全部未推送过的本地标签；
* 命令git tag -d <tagname>可以删除一个本地标签；
* 命令git push origin :refs/tags/<tagname>可以删除一个远程标签。