git教程(5) 分支管理(2)

一 分支管理策略
1 小背景
合并分支时，git会尽可能的采用 Fast forward 模式【删除分支就会丢掉该分支的信息！！】，但这种模式下，删除分支后，会丢掉分支信息。

如果强制禁用 Fast forward 模式，git就会在 merge时 生成一个新的 commit，这样就可以从分支历史上看出分支信息。
提示： --no-ff 方式的 git merge。

2 --no-ff【不采用fast-forwardd方式的】git merge 实战步骤：
2.1 先创建并切换到 dev 分支【git checkout -b dev 等价于 git switch -c dev！！】

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

2.2 在 dev分支上 修改readme.txt文件， 并提交一个新的 commit 【修改文件、add、commit】。

```
$ git add readme.txt
$ git commit -m "add merge"
[dev f52c633] add merge
 1 file changed, 1 insertion(+)
```

2.3 切回 master 以准备合并 dev分支到 master分支上。【git checkout master 即 git switch master！！！】

```
$ git checkout master
Switched to branch 'master'
```

2.4 在master分支上进行 dev分支的合并。【git merge --no-ff -m 'merge with no-ff dev】

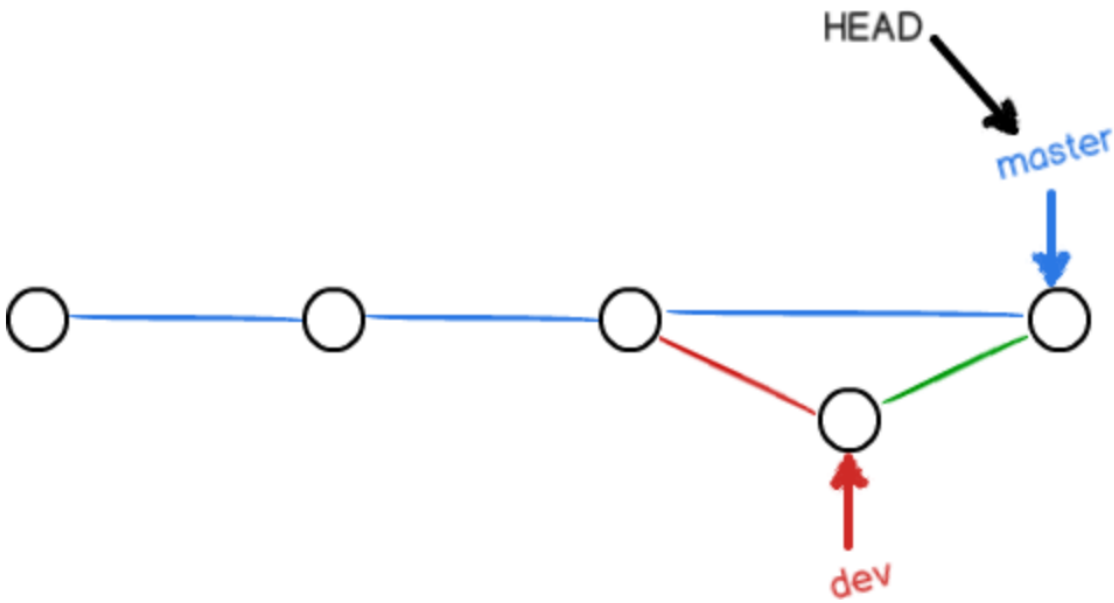
```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

注意：因为本次合并要创建一个新的提交【因为禁用了 “ff的合并模式”】。
所以需要加上 -m【毕竟创建了新的提交！！！】 参数，把 commit 描述写进去！！

2.5 合并后进行 分支历史的查看 git log【git reflog 可以找到“未来的版本号“，然后通过 git reset HEAD commit_id 回到”未来“】

```
$ git log --graph --pretty=oneline --abbrev-commit
*   e1e9c68 (HEAD -> master) merge with no-ff
|\
| * f52c633 (dev) add merge
|/
*   cf810e4 conflict fixed
...
```

不使用【那使用了的是啥情况？？没有绿色那一条多余的？？】 fast forward 模式的 merge 如下图：



3 分支策略

3.1 进行分支管理的几个基本原则：
master分支应该是非常稳定的，也就是仅仅用来发布新版本，平时工作不能在上面干活！！
“所有开发人员”在dev分支上干活，每个人都有自己的分支，时不时的往dev分支上合并即可！！ 【“感觉很奇怪？？！每个人在自己的分支上进行工作，写好了下合并到 dev分支上，在合并到 master 分支上？？！“】

所以，团队合作的分支看起来就像这样：

4 小结
合并分支时，加上 --no-ff 参数【如 git merge --no-ff -m '不使用 ff的合并模式' dev】就可以使用普通的合并方式【“而非快进的合并方式！！”】。合并后的历史有分支、就能看出来曾经做过合并操作！！ 而 fast forward 合并就不不能看出来曾经做过合并！！

二 bug分支
1 背景
软件开发中，bug就像家常便饭一样。有了bug就需要去修复。
之前说过，git上的分支功能很强大！ 每一个bug都可以通过一个临时分支来修复，修复、合并到 master分支上， 就可以将该临时分支删除了！！

2 stash【工作到一半，需要紧急修复某一个 bug 。“类似游戏里的一个检查点check point。git stash 和 git stash pop （git stash list）去使用“！！！】

2.1 git的 stash功能，可以把当前的工作现场 “储藏” 起来，等以后恢复现场后继续工作。

```
$ git stash
Saved working directory and index state WIP on dev: f52c633 add merge
```

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）因此可以放心大胆的创建分支去修复 bug。

2.2 假定需要在 master分支上修复，就从 master 创建临时分支【还有这种区别、说法？？？ Why？？？】。

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

2.3 现在 修复bug、提交 。

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

```
$ git add readme.txt
$ git commit -m "fix bug 101"
[issue-101 4c805e2] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
```

2.4 切回 master分支， 并完成合并 删除这个修复bug的临时分支！！

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
  (use "git push" to publish your local commits)

$ git merge --no-ff -m "merged bug fix 101" issue-101
Merge made by the 'recursive' strategy.
 readme.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

2.5 接着会到原来的 dev分支上进行干活。

```
$ git checkout dev
Switched to branch 'dev'

$ git status
On branch dev
nothing to commit, working tree clean
```

注意：通过git status，我们会发现工作区是干净的。需要进行 git stash pop “弹出之前未完成的工作内容”！！即 git stash apply + git stash drop 。 （git stash list ---> 查看 check point 列表！！）

```
$ git stash list
stash@{0}: WIP on dev: f52c633 add merge
```

2.6 git stash pop【即 git stash apply + git stash drop】

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：

```
$ git stash pop
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   hello.py

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:   readme.txt

Dropped refs/stash@{0} (5d677e2ee266f39ea296182fb2354265b91b3b2a)
```

注意如果进行了 多次的 git stash 操作,可以先使用 git stash list 找到我们想要的那一个“check point”，然后通过 git stash apply stash_id【如 git stash apply stash@{0} 】去恢复

到我们先预期的工作进度点！！

3 小问题，master上修复了bug，但是 dev分支上还存在该bug！！如何简单操作、修复bug？？？【cherry-pick 命令！！】
同样的bug,想要在dev上修复，我们只需要把 bug修复 这个提交的所做的修改“赋值”到 dev分支上即可。 ----> cherry-pick 命令！！

```
$ git branch
* dev
  master
$ git cherry-pick 4c805e2
[master 1d4b803] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
```

注意：git cherry-pick 4c805e2 中的 4c805e2 是那次修复 bug合并后产生的 commit_id值！！
执行完 cherry-pick 命令后，git自动给 dev分支做了一次提交，这次提交的 commit_id时 1d4b803 ，不同于 4c805e2 的。因为这2个 commit 知识改动相同，但确实是2个不同的commit！！
当然我们也可以在 dev分支上进行修复bug，然后再 master分支上 进行 cherry-pick 的操作！！！【一样需要 git stash 去保存现场！！】

5 小结

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；

在master分支上修复的bug，想要合并到当前dev分支，可以用`git cherry-pick <commit>`命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

三 feature分支
1 添加一个新功能时，你肯定不希望因为一些实验性的代码，把主分支搞乱了，最好新建一个 feature分支。
在 feature分支上面 开发，开发完成后、合并、删除 feature分支即可！！

2 开发“新特性”的具体步骤
2.1 创建并切换到 feature-vulcan 分支

```
$ git checkout -b feature-vulcan
Switched to a new branch 'feature-vulcan'
```

2.2 开发完成后，进行 add、commit 操作

```
$ git add vulcan.py

$ git status
On branch feature-vulcan
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   vulcan.py

$ git commit -m "add feature vulcan"
[feature-vulcan 287773e] add feature vulcan
 1 file changed, 2 insertions(+)
 create mode 100644 vulcan.py
```

2.3 切回 dev分支【git checkout dev】， 准备合并!

```
$ git checkout dev
```

2.4 “意外发生” —— 因为某种原因，我们这个分支【“需求”】 不需要了，白干了！！ 于是这个分支应该就地销毁！！

git branch -d feature-vulcan 【但是此时会销毁失败，因为 feature-vulcan分支还没被合并！！ 需要用上 —D参数！！】

```
$ git branch -d feature-vulcan
error: The branch 'feature-vulcan' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-vulcan'.
```

git branch -D feature-vulcan 去强行删除【如果我们在 feature-vulcan分支上也能正常执行这条命令吧？？？！】

```
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 287773e).
```

3 小结

开发一个新 feature【“新需求、新特性”】，最好新建一个分支。
如果真的要删除、丢弃一个没有被合并过【-D参数，】的分支，应该通过【—D参数】 git branch -D brach_name 命令！！

待续