**为何使用Git**

关于Subversion和Git的优劣比较有很多的文章已经进行了比较详细的介绍，这并不是这篇文章的重点。但是Git的一些优势却是这种模型的基础，因此对于这一部分应当进行必要的介绍。

**分支**

But with Git, these actions are extremely cheap and simple, and they are considered one of the core parts of your daily workflow, really. For example, in CVS/Subversion books, branching and merging is first discussed in the later chapters (for advanced users), while in every Git book, it’s already covered in chapter 3 (basics).

Git相比较Subversion而言，有一个显著的优势就是对分支的使用非常的方便。在Git中，分支的使用是非常鼓励和推荐的，因此在《Pro Git》中把分支的使用是作为基础章节来介绍的，也就是说分支是Git中经常会用到的操作。而本文所介绍的这种模型，就是建立在对分支频繁操作（创建，切换，合并等）的基础上的。

**非集中式**

非集中式的版本控制系统是Git的另一大优势。这就表示，在使用Git的时候，每一次代码的提交都不必同步到远程服务器中，开发人员可以在外部网络环境（无法连接内网），甚至离线的情况下进行代码的版本控制。作为非集中式的版本控制系统，开发人员可以在本地创建分支而不必同步到服务器上，这一点是构成文中Git分支模型的另一大基础。

**主分支(Main Branch)**

在Git分支模型中存在两个主分支，这两个分支是不可或缺的：

* master分支
* develop分支

**master分支**

master作为Git中默认的主分支，是使用Git的开发者们非常熟悉的默认主分支名称。在Git分支开发模型中，master分支的HEAD节点始终处于“准备好进行生产的状态”，即master分支的HEAD节点所指向的版本始终是可以用于生产环境的正式版本。当其他分支的代码版本合并到master分支时（随后打上版本标签），通常意味着一个新的正式版本已经发布。该过程的具体介绍详见后文。

**develop分支**

develop分支作为另一个主分支，其HEAD节点总是指向下一个待发布版本的最新变化。develop分支的版本变更通常来源于辅助分支的合并（稍后介绍），因为develop分支也常被称为“整合分支”。当develop分支达到某一稳定点，可进行新版本的发布时，develop分支上的所有变更应该被合并到master分支并打上tag标签，该过程详见后文。

**辅助分支(Supporting Branch)**

除了master分支和develop分支这两个主分支以外，Git分支模型中拥有一些“辅助分支”，在团队开发中对develop分支和master分支进行帮助，例如对新需求的研发，新版本发布前的准备工作以及新版本bug的紧急修复等。和主分支不同的是，这些分支的生命周期都是很有限的，最终都将会被删除。辅助分支中有以下三类分支：

* 需求分支(Feature Branch)
* 发布分支(Release Branch)
* 修复分支(Hotfix Branch)

上述三种辅助分支，每一种都有其特定的功能，并遵守各自严格的规则,例如分支的输入分支、分支的输出（合并）分支等等。下文将逐一描述。

**需求分支(Feature Branch)**

分支来源：develop分支

分支去向：develop分支

分支命名：任意名称，除master，develop，以“release-”开头，以“hotfix-”开头的分支以外。

需求分支用于为未来的软件版本开发新的功能需求。当进行一个需求的研发时，该需求将被整合进正式版本是未知，所以需要单独创建分支对该需求进行研发，只要该需求尚在开发中，该需求分支就会一直存在。需求分支最终会被合并到develop分支中作为下一个待发布版本的功能之一，或者由于该需求无法实现从而被抛弃。

**注：需求分支通常仅仅存在于开发者的代码仓库中（本地仓库），并不上传到远程分支。**

**如何创建需求分支**

创建需求分支时，该分支必须从develop分支得到：

$ git checkout -b feature\_branch develop

该命令从develop分支创建一个新的分支“feature\_branch”，并从当前分支切换到“feature\_branch”分支，相当于：

$ git branch feature\_branch develop

$ git checkout feature\_branch

**将已完成的需求分支合并到develop分支**

已完成的需求分支需要被合并到develop分支，作为待发布版本的需求之一：

$ git checkout develop #切换到develop分支

$ git merge --no-ff feature\_branch #合并分支

$ git branch -d feature\_branch #删除需求分支

$ git push origin develop #推送

--no-ff表示No Fast Forward，在合并使，即使可能是fast forward方式，也会创建一个新的commit节点。

**关于fast forward**

当前分支合并到另一分支时，如果没有分歧解决，就会直接移动文件指针。这个过程叫做fast forward。

例如，开发一直在master分支进行，但忽然有一个新的想法，于是新建了一个develop的分支，并在其上进行一系列提交，完成时，回到 master分支，此时，master分支在创建develop分支之后并未产生任何新的commit。此时的合并就叫fast forward，如下图：



git merge develop

可以看到master在合并develop分支的时候并没有产生新的节点

回到develop分支，对代码进行修改，提交。切换到master分支，使用git merge develop --no-ff 进行合并，此时会产生一个commit节点,如下图：



git merge develop --no-ff

删除分支develop后，如下图：



删除develop分支后

很明显使用--no-ff合并时，在删除develop分之后，该分支的合并信息仍然被保留，在以后的代码分析中可以便捷的查看到历史信息，而fast forward方式则无法辨识代码的合并信息。正如原文所说：

The --no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

### 发布分支(Release Branch)

分支来源：develop分支

分支去向：develop分支和master分支

分支命名：以"release-"开头

发布分支用于辅助新版本（生产环境）发布的准备工作，例如小bug的修复，或者版本号的修改等等。使用发布分支的好处是，当从develop分支中创建发布分支以后，develop分支便可以进行新版本之后需求的研发工作，从而既不会影响到最新的研发进度，也不会影响到新版本的发布。

#### 创建发布分支

发布分支以develop分支作为源分支。例如，目前develop分支上的所有需求将作为版本1.2发布，这时可以创建一个分支"release-1.2"。此时可以继续在develop分支上进行新需求的研发，而1.2版本的发布工作将由“release-1.2”分支来完成：

$ git checkout -b release-1.2 develop #创建并切换到"release-1.2"分支

$ vim file #表示对版本号的修改，或者小bug的修复等

$ git commit -a -m "更新版本至1.2" #提交代码

**注： 如果在发布分支进行小型bug的修改，则需要将提交后的代码合并到develop分支中**

#### 完成发布分支

当发布分支完成代码的提交（如果修复过bug，则要合并到develop分支）后，需要将发布分支合并到master分支上，并进行tag操作，如：

$ git checkout master

$ git merge --no-ff release-1.2

$ git tag -a "v1.2"

PS：合并到develop的操作：

$ git checkout develop

$ git merge --no-ff release-1.2

完成合并操作以后，删除该发布分支：

$ git branch -d release-1.2

### 修复分支(Hotfix Branch)

分支来源：master分支

分支去向：develop分支和master分支

分支命名：以"hotfix-"开头

修复分支用于正式版本的紧急修复，在紧急修复完成以后**必须同时被合并到master分支和develop分支**，这是修复分支和发布分支不同之处(二者的来源也不同)，和发布分支类似，修复分支在修复bug，提交，被合并以后，也要进行tag操作。

#### 创建修复分支

$ git checkout -b hotfix-1.2.1 master

$ vim file #表示修复bug

$ git commit -a -m "修复bug"

$ vim file #表示更新版本号

$ git commit -a -m "版本更新为1.2.1"

#### 完成修复分支

$ git checkout master

$ git merge --no-ff hotfix-1.2.1

$ git tag -a 1.2.1 #tag操作

不要忘记合并修复分支到develop分支

$ git checkout develop

$ git merge --no-ff hotfix-1.2.1

删除修复分支：

$ git branch -d hotfix-1.2.1

## 总结

以上就是这篇文章的主要内容，原文作者充分利用了git的诸多优势，为我们提供了一种优雅的版本管理模型，希望能够对大家有所帮助。整篇文章如果有任何翻译解释不到位的地方，还请各位指出，谢谢。最后，附上整个分支模型的示意图，感谢阅读。

