从 SVN 迁移至 Gitlab + Gitflow 总结

转载请注明出处<http://blog.csdn.net/uxyheaven/article/details/50373076>

背景

之前在的公司一直都是用svn做源代码管理, 人少的时候也木有发现什么太大的弊端. 后来换了一个团队后, 业务发展速度实在是超乎想象.   
短短的几个个月时间, 我们单语种的开发人员扩充到了10人以上, 接下来的两个月中, 人数要到20人. 我们的并行版本也是增加了好几个. 有的版本是由于特殊原因, 一直木有发版, 并行了几个月(代码也就是一直木有同步了…); 有的则是业务线的开的多, 就是需要同时开很多个分支, 一个分支一个版本, 在发版的时候在合并起来.

在这期间, 我们次次发版前的合并代码是一项艰巨的任务. 既然代码管理已经是个问题了, 那么就需要改了.

业务分层

业务层的代码只要不涉及到其他业务模块, 总是相对独立的.   
我在刚来的时候先横一刀斩出了业务层. 从物理目录上隔离了业务模块代码和其它代码. 再竖几刀切出了各个业务模块. 虽然代码还是原来的代码, 不过最起码物理文件夹不一样了, 合并代码的时候再不济直接整个文件夹替换便是.

然而事实证明我是too young, too simple. 如果次次的代码变动只是业务层, 理论上也不会有啥冲突. 问题是出在我们的业务发展太迅速, 我们的业务代码以外的其它代码也是动不动就不能满足需求了, 需要随着业务代码的改变而改变. 这就导致了业务A为了满足自己的需求直接就改了其它代码成PPAP, 然后业务B为了满足自己的需求改了其它代码成PPBP, 嘛, 冲突就这么来了.

Gitflow

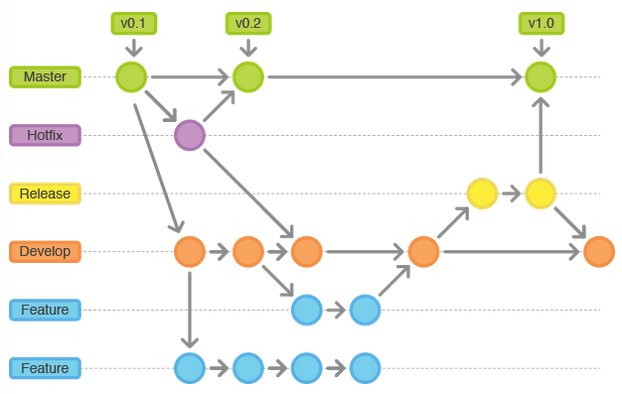
俗话说没有经历过业务高速发展的架构师都是扯淡. 很庆幸我们团队能有这次机会随着业务一起成长.

我们先来分析下问题所在:

* 开发人员多
* 并行开发的版本多
* 底层代码一直在改

我决定采用 Gitflow工作流 去解决这次的问题.

Gitflow工作流通过为功能开发, 发布准备和维护分配独立的分支, 让发布迭代过程更流畅. 严格的分支模型也为大型项目提供了一些非常必要的结构.   
Gitflow工作流没有用超出功能分支工作流的概念和命令, 而是为不同的分支分配一个很明确的角色, 并定义分支之间如何和什么时候进行交互.



主要分支:

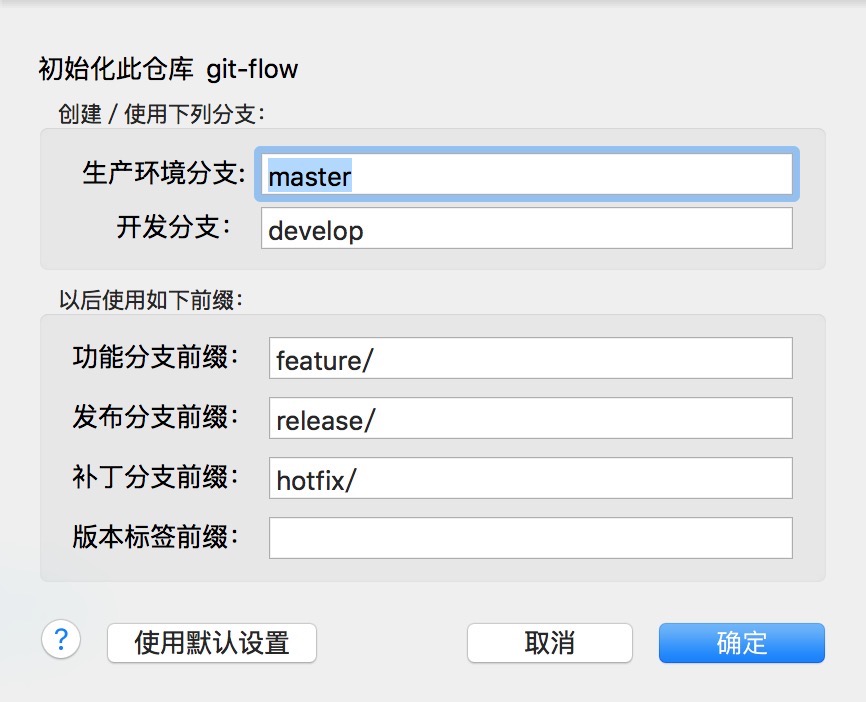
* Master(绿色): 永远处在 production-ready 状态
* Develop(橙色): 最新的下次发布开发状态

支援性分支:

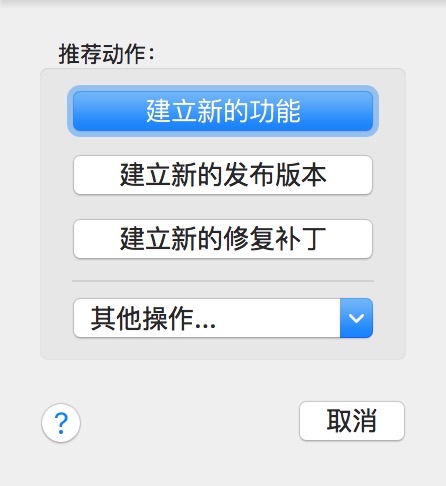
* Feature(蓝色): 开发新功能都从 develop 分支出来，完成后 merge 回 develop
* Release(黄色): 准备要 release 的版本，只修 bugs。从 develop 分支出来，完成后 merge 回 master 和 develop
* Hotfix(灰色): 等不及 release 版本就必须马上修 master 赶上线的情况。会从 master 分支出来，完成后 merge 回 master 和 develop

工具方面, 我决定用SourceTree, 因为这个APP不仅自带官方汉化还自带gitflow, 学习成本很低.

先点击下Git工作流按钮(first blood), 初始化下环境. 这样就有了master 和 develop 两个分支. 还为功能分支, 发布分支, 补丁分支分别制定了前缀.



当需要开发一个新功能的时候. 切换到 develop 分支上, 接着还是点击 Git工作流按钮(double kill), 选择建立新的功能, 然后为功能起一个名称b. 这样本地分支目录下就会多了一个feature文件夹, 里面有一个b分支.



当这个功能b开发完成后, 还是点击Git工作流按钮(trible kill), 点完成当前版本.代码会自动合并到developer分支上, 并且可以选择删除当前的b分支.



当需要发布的时候, 切换到developer分支上, 点击Git工作流按钮(urltra kill), 选择建立新的发布版本1.2. 这样本地分支目录下就会多了一个release文件夹, 里面有一个1.2分支.   
当1.2没问题后, 还是点击Git工作流按钮(rammpage), 点完成当前版本, 给当前版本打个tag. 代码会自动合并到master分时和developer分支上.



如果线上版本一步小心有了一个bug, 切换到master分支上, 还是点击Git工作流按钮, 点建立新的修复版本1.1.1. 完成后还是点击Git工作流按钮, 点完成当前版本, 给当前版本打个tag. 代码会自动合并到master分时和developer分支上.



这期间, 统一的入口点都是点击Git工作流按钮(这个按钮已经holy shit), 是不是很方便? 这也是为啥我建议大伙用客户端的原因. 可以不用手把手教了, 都是汉语的, 一看就懂了.

Gitlab

我们的代码之前是托管在公司的svn上, 现在是托管在公司司的gitlab上. 为啥用都用公司的呢, 1方便, 2公司禁止私建服务器, so怎么搭建的流程就直接跳过了.

具体的步骤是这样子:

* 用户访问公司gitlab地址
* master权限用户给其他用户添加添加权限
* 用户安装git or git客户端
* 用户添加ssh

gitlab的使用主要是一个权限问题, 虽然help里写的也很详细, 不过就是太详细了反而显的啰嗦. 我在这里把需用关注的一些权限总结如下:

项目权限

* private 私有, 项目必须显式地为每个用户访问权授予。
* internal 内部, 任何登录用户都可以克隆这个项目。
* public 共有, 所有用户都可以看到这个项目

用户权限

* guest 只能提issue
* reporter 可以看文件, 不能上传任何修改
* developer 可以看, 可以上传developer分支的修改, 可以建分支, 可以删远程分支. 不能上传master分支的修改
* master 都可以

wiki权限

* guest 只能看
* reporter 只能看
* developer 可以增改页面, 但是不能删除页面
* master 可以增删改

我先是为我们团队建立了一个group, 然后默认所有人都是guest权限, 然后再单独为每个项目为相关的开发开通developer权限, 为部分开发开通了master权限, 让他们也可以自己分配自己的项目内的权限.

迁移

将旧代码迁移到新平台. 我先是给两个同事share了gitflow, 让他们了解到新的开发模式有哪些优点. 然后安排他们一个去研究Gitlab+Cocoapods(都换成git了, 肯定要上这货了), 一个去研究配套的jenkins.

当一切都研究的差不多的时候, 我让他俩在公司内部准备了一次share, 用于让公司各个业务线的负责人了解到现在存在哪些问题, 我们用怎样的方法去改善它, 我们最近弄出的东西到底是什么. 然后我在团队内部分享了gitflow, 用于让团队内部的开发GG了解我们接下来要怎么做.

接下来就是真正的迁移工作. 迁移的时候, 没得商量, 直接冻结了svn, 告知所有小伙伴禁止在svn上提交代码, 今后统一换成git.

至此这次的迁移工作就算是完成了. 在整套全新的持续集成环境落地的期间我们不是没有遇到问题, 反而是遇到了很多问题. 这里列出一些问题, 给诸位当做参考了.

问题

权限问题

master权限太大一不小心什么都删了. 如果只有个别开发配置master权限, 一般开发用developer权限的话, developer还是可以删除远程的feature.

解决

配置分支权限, 把重要的分支设置成保护的. 还可以设置成develpoer都不能push, 不过这个没啥必要.

合并冲突问题

两个feature建立了新文件后, 合并代码必冲突.   
第一个完成的feature肯定没问题拉.   
第二个完成的feature需要需要解决xcodeproj里的project的冲突才可以点完成.

解决

每天先把developer的代码往feature合并一次后, 再进行开发.

push merge问题

当一个feature开发完成后, 可以申请合并到developer里, 申请的同学可以自己点accept.

解决

developer分支设置成保护, 同时禁止developer分支的push.

依赖问题

当featureA和featureB都依赖一个底层的接口的时候, feature如何处理.

解决

先拉出featrueC写底层接口, 完成后, 在拉出featureA和featureB. 但这样的缺点是影响进度. 还可以featureA, featureB, featureC, 都拉出来, featureA和featureB先用挡板开发. 当featrueC开发完成后, 合并到develop上去, 然后把develop的代码合并到featureA, featureB上.

打包问题

业务线多的话feature也很多, 还经常改名字, 对应的jenkins需要经常配置.

解决

jenkins可以多配几个jobs

svn问题

代码在git上, 但是有些资源在svn上, 打包的时候需要下载下来, 暂时不知道怎么拉下来

解决

把那些资源丢到git上. 再没丢之前先用命令行拉, but小伙伴说木有拉下来…

# [GitLab常用命令整理](http://www.cnblogs.com/luweiwei/p/4866930.html)

进入本地仓库访问位置之后执行命令

**1) 远程仓库相关命令**

检出仓库：$ **git** clone **git**://github.com/jquery/jquery.**git**

查看远程仓库：$ **git** remote -v

添加远程仓库：$ **git** remote add [name] [url]

删除远程仓库：$ **git** remote rm [name]

修改远程仓库：$ **git** remote set-url --**push**[name][newUrl]

拉取远程仓库：$ **git** pull [remoteName] [localBranchName]

推送远程仓库：$ **git push** [remoteName] [localBranchName]

**2）分支(branch)操作相关命令**

查看本地分支：$ **git** branch

查看远程分支：$ **git** branch -r

创建本地分支：$ **git** branch [name] ----注意新分支创建后不会自动切换为当前分支

切换分支：$ **git** checkout [name]

创建新分支并立即切换到新分支：$ **git** checkout -b [name]

删除分支：$ **git** branch -d [name] ---- -d选项只能删除已经参与了合并的分支，对于未有合并的分支是无法删除的。如果想强制删除一个分支，可以使用-D选项

合并分支：$ **git** merge [name] ----将名称为[name]的分支与当前分支合并

创建远程分支(本地分支**push**到远程)：$ **git push** origin [name]

删除远程分支：$ **git push** origin :heads/[name]

我从master分支创建了一个issue5560分支，做了一些修改后，使用**git push** origin master提交，但是显示的结果却是'Everything up-to-date'，发生问题的原因是**git push** origin master 在没有track远程分支的本地分支中默认提交的master分支，因为master分支默认指向了origin master 分支，这里要使用**git push** origin issue5560：master 就可以把issue5560推送到远程的master分支了。

    如果想把本地的某个分支test提交到远程仓库，并作为远程仓库的master分支，或者作为另外一个名叫test的分支，那么可以这么做。  
  
$ **git push** origin test:master         // 提交本地test分支作为远程的master分支 //好像只写这一句，远程的github就会自动创建一个test分支  
$ **git push** origin test:test              // 提交本地test分支作为远程的test分支  
  
如果想删除远程的分支呢？类似于上面，如果:左边的分支为空，那么将删除:右边的远程的分支。  
  
$ **git push** origin :test              // 刚提交到远程的test将被删除，但是本地还会保存的，不用担心

**3）版本(tag)操作相关命令**

查看版本：$ **git** tag

创建版本：$ **git** tag [name]

删除版本：$ **git** tag -d [name]

查看远程版本：$ **git** tag -r

创建远程版本(本地版本**push**到远程)：$ **git push** origin [name]

删除远程版本：$ **git push** origin :refs/tags/[name]

**4) 子模块(submodule)相关操作命令**

添加子模块：$ **git** submodule add [url] [path]

如：$ **git** submodule add **git**://github.com/soberh/ui-libs.**git** src/main/webapp/ui-libs

初始化子模块：$ **git** submodule init ----只在首次检出仓库时运行一次就行

更新子模块：$ **git** submodule update ----每次更新或切换分支后都需要运行一下

删除子模块：（分4步走哦）

1)$ **git** rm --cached [path]

2) 编辑“.gitmodules”文件，将子模块的相关配置节点删除掉

3) 编辑“.**git**/config”文件，将子模块的相关配置节点删除掉

4) 手动删除子模块残留的目录

**5）忽略一些文件、文件夹不提交**

在仓库根目录下创建名称为“.gitignore”的文件，写入不需要的文件夹名或文件，每个元素占一行即可，如

target

bin

\*.db

2. git pull：相当于是从远程获取最新版本并merge到本地

git pull origin master