Git开发版本管理规范V1.0

Git版本开发主要事项

- master ————> 归档
- release ————> 发布上线
- dev ————> 开发
- test ————>test
- Bugfix(更新是否使用)
- Tag 项目版本归档，与版本编号一一对应

> 多个研发人员在dev分支上开发, 在test分支上进行测试，【预发布，合并到release, 进行版本发布】，正式上线后，release合并到master, 打标签tag进行归档

各分支使用办法说明如下：

master分支

master和dev分支都是主分支，主分支是所有开发活动的核心分支。所有的开发活动产生的输出物最终都会反映到主分支的代码中。

master分支上存放的应该是随时可供在生产环境中部署的代码（Production Ready state）。当开发活动告一段落，产生了一份新的可供部署的代码时，master分支上的代码会被更新。同时，每一次更新，都有对应的版本号标签（TAG）。

dev分支

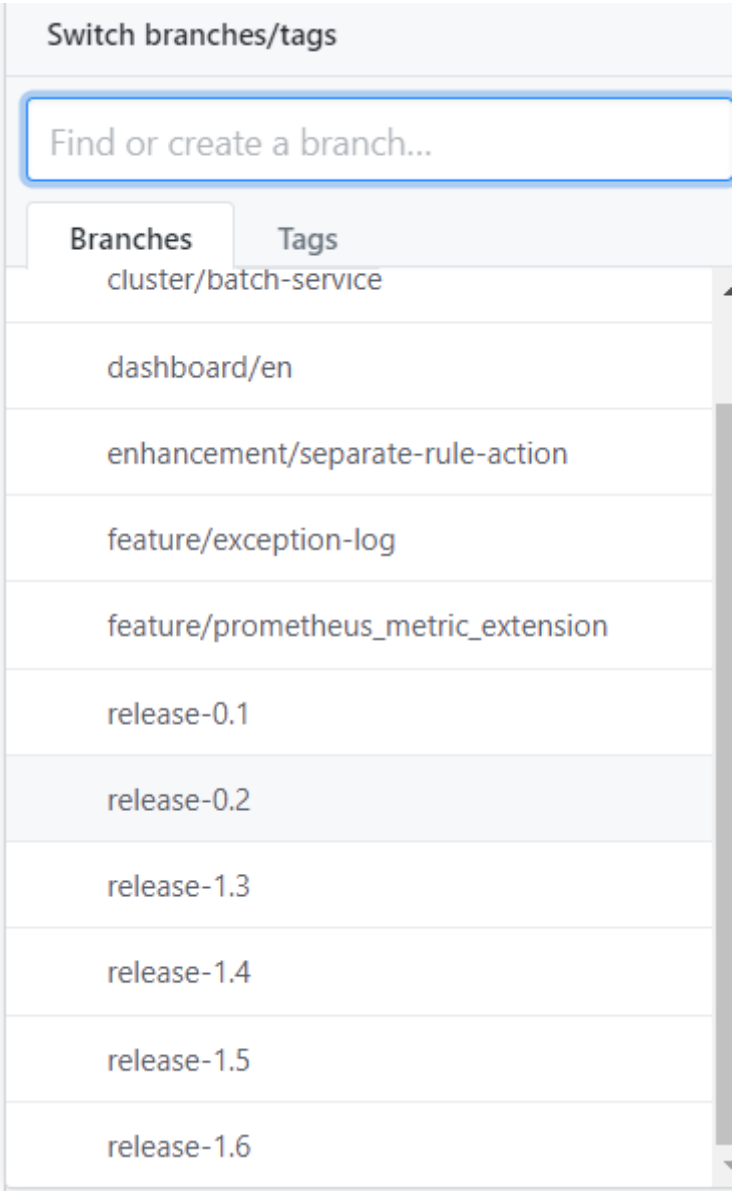
dev分支是保存当前最新开发成果的分支。通常这个分支上的代码也是可进行每日夜间发布的代码（Nightly build）。因此这个分支有时也可以被称作“integration branch”。

当develop分支上的代码已实现了软件需求说明书中所有的功能，通过了所有的测试后，并且代码已经足够稳定时，就可以将所有的开发成果合并回master分支了。对于master分支上的新提交的代码建议都打上一个新的版本号标签（TAG），供后续代码跟踪使用。

test分支

当开发人员将代码交付测试部门时，测试人员，将代码merge到test分支中，此时触发测试分支的构建的流程，完成构建后，通过管理平台进行测试环境的发布。

release分支



使用规范：

1.可以从dev分支派生

2.必须合并回dev分支和master分支

3.分支命名惯例：release-*

release分支是为发布新的产品版本而设计的。在这个分支上的代码允许做小的缺陷修正、准备发布版本所需的各项说明信息（版本号、发布时间、编译时间等等）。通过在release分支上进行这些工作可以让develop分支空闲出来以接受新的feature分支上的代码提交，进入新的软件开发迭代周期。

当develop分支上的代码已经包含了所有即将发布的版本中所计划包含的软件功能，并且已通过所有测试时，我们就可以考虑准备创建release分支了。而所有在当前即将发布的版本之外的业务需求一定要确保不能混到release分支之内（避免由此引入一些不可控的系统缺陷）。

成功的派生了release分支，并被赋予版本号之后，develop分支就可以为“下一个版本”服务了。所谓的“下一个版本”是在当前即将发布的版本之后发布的版本。版本号的命名可以依据项目定义的版本号命名规则进行。

hotfix分支

使用规范：

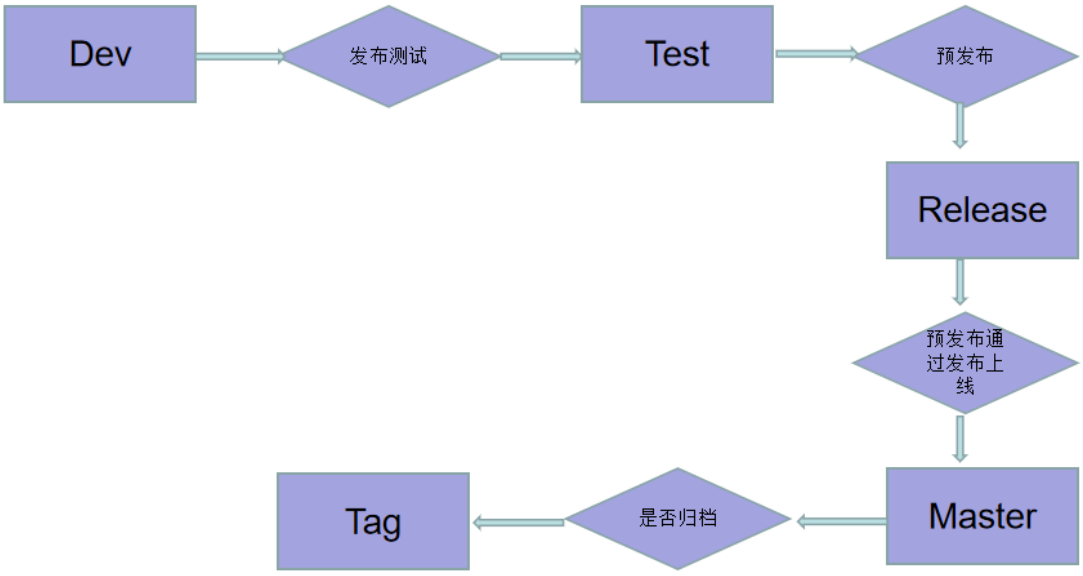
1.可以从test分支派生

2.必须合并回master分支和dev分支

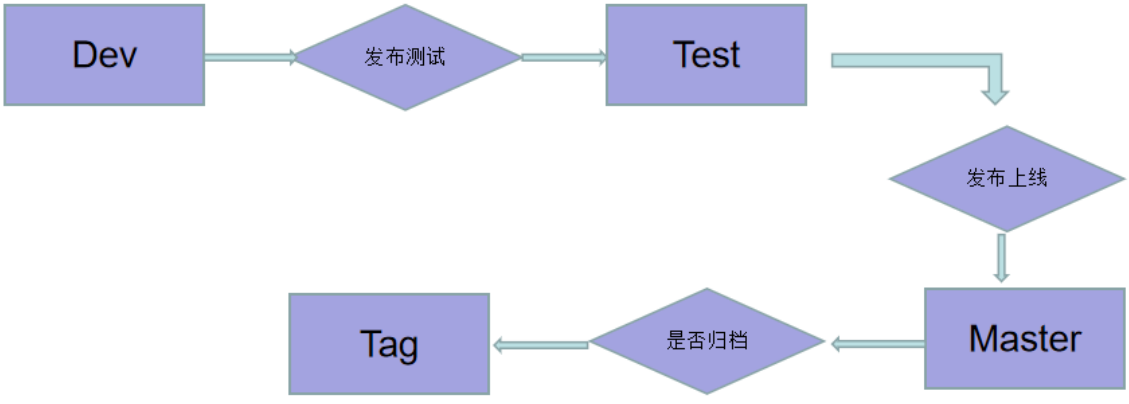
3.分支命名惯例：fix-*

除了是计划外创建的以外，hotfix分支与release分支十分相似：都可以产生一个新的可供在生产环境部署的软件版本。当生产环境中的软件遇到了异常情况或者发现了严重到必须立即修复的软件缺陷的时候，就需要从master分支上指定的TAG版本派生hotfix分支来组织代码的紧急修复工作。

整体交付流程如下：



根据实际项目情况选择是否需要RELEASE分支



dev(开发分支)

开发分支是研发同学最需要关心的，所有的开发工作应该围绕dev分支进行展开，创建dev分支脚本如下：

dev分支在实际工作中主要有两种常用场景：

1. 单个版本开发
2. 多个版本并行开发

第一种情况，属于比较常见的情况，即开发过程中都是按顺序进行的，即1.0.0开发完成之后才会进行开发1.1.0的需求开发，1.1.0完成上线后再进行1.2.0的版本开发，针对这种，工作直接在dev分支上开展就可以了，不需要做特殊处理，因为不会产生任何干扰~~

相关操作脚本：

```
#切换到dev分支下进行开发
git checkout dev
#同步最新的代码
git pull
#。。。。。
开发代码，各种commit,push，这里就不列了
#。。。。。
#开发完测试通过后合并到主干，先把本地的dev代码更新到最新，接着执行下面脚本
git checkout master
git pull
git merge --no-ff dev
git push
```

针对第二种情况，可能在互联网公司会比较常见，在开发资源比较充足的情况下，多个版本可能同时并行，即1.1.0和1.2.0或者更多版本在同时开发，但是又想快速试水一些功能，不可能等到全部做完都上线，所以1.1.0会先发，1.2.0会在另外一个时间点发，产品经理通常会很着急，认为错过这个时间点就会损失十几亿的感觉，碎碎的忧伤，所以分支的合理管理非常重要，不然在开发过程中会显得非常混乱，如果都在dev分支上进行开发就会把没做完的功能都发线上了，这时候就更碎了~~~ git 也给我们使用feature功能模式，即功能分支，可以通过这个来划分，如下：

先创建两个feature分支，跟创建dev分支一样，不过这次是从dev分支进行创建

```
git checkout -b feature-1.1 dev
git checkout -b feature-1.2 dev
```

开发1.1.0版本操作的相关脚本：

```
git checkout feature-1.1
#又是各种commit push操作，开发完成测试通过后执行下面操作，合并代码到dev分支：
git checkout dev
git pull origin dev
git merge --no-ff feature-1.1
git push
#删除本地feature分支
git branch -d feature-1.1
#删除远程feature分支
git push origin --delete feature-1.1
```

开发1.2.0版本操作的相关脚本：

```
git checkout feature-1.2
#又是各种commit push操作，开发完成测试通过后执行下面操作，合并代码到dev分支：
git checkout dev
git pull origin dev
git merge --no-ff feature-1.2
git push
#删除本地feature分支
git branch -d feature-1.2
#删除远程feature分支
git push origin --delete feature-1.2
```

Bug分支解决方案

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xxxx-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？

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：

```
git stash
```

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test分支上修复，就从test创建临时分支：

```
git checkout test
git checkout -b xxxx-101
```

现在修复bug，然后提交：

```
git add readme.md
git commit -m "fix bug 101"
```

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

```
git checkout test
git merge --no-ff -m "merged bug fix 101" xxxx-101
```

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！

```
git checkout devgit status
```

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：

```
git stash list
```

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：

```
git stash pop
```

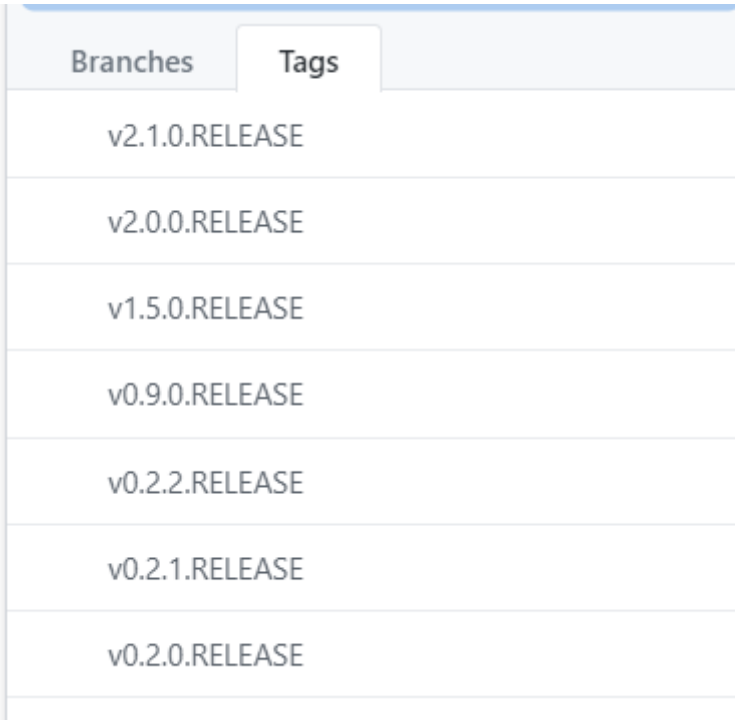
再用git stash list查看，就看不到任何stash内容了。

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}
```

标签管理

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，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。

命令git tag 用于新建一个标签，默认为HEAD，也可以指定一个commit id。

git tag -a -m “blablabla...”可以指定标签信息。

还可以通过-s用私钥签名一个标签：

```
git tag -s v0.5 -m "signed version 0.2 released" fec145a
```

git tag可以查看所有标签。

用命令git show 可以查看某个标签的详细信息。

如果标签打错了，也可以删除：

```
git tag -d v0.1
```

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：

```
git push origin v1.0
```

或者，一次性推送全部尚未推送到远程的本地标签：

```
git push origin --tags
```

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：

```
git tag -d v0.9
```

然后，从远程删除。删除命令也是push，但是格式如下：

```
git push origin :refs/tags/v0.9
```