Java项目版本管理规范 - 简书

Java项目版本管理规范

版本命名规则

Prong Boot / Prong Cloud的版本命名规范在maven的规范上做了进一步的严格要求，具体格式为：

<主版本>.<次版本>.<增量版本>-<代号>

部分	说明	含义
主版本	必须	主版本一般来说代表了项目的重大的架构变更，比如说Maven 1和Maven 2，在架构上已经两样了，将来的Maven 3和Maven 2也会有很大的变化。
次版本	必须	次版本一般代表了一些功能的增加或变化，但没有架构的变化，比如说Nexus 1.3较之于Nexus 1.2来说，增加了一系列新的或者改进的功能，但从大的架构上来说，1.3和1.2没什么区别。
增量版本	必须	增量版本一般是一些小的bug修复，没有有重大的功能变化。
代号	必须	分为不稳定版本（SNAPSHOT）和稳定版本（非SNAPSHOT）两类。 SNAPSHOT是指开发分支中的“最新”代码，表示代码可能随时变化，发布到maven的snapshot仓库。 相反，“稳定”版本中的代码（非SNAPSHOT后缀的任何版本值）都是不变的，发布到maven的release仓库。

代号的取值范围：

代号	分类	版本	说明
SNAPSHOT	不稳定版本	开发版本	指develop分支中或者hotfix/xxx分支上的最新代码，表示代码可能随时变化。
RCx	稳定版本	预发布版本	当代码实现了全部功能，清除了大部分 bug，接近发布倒计时。x是数字，如RC1、RC2。
RELEASE	稳定版本	正式发布版本	指master分支中的某个tag的对应的代码，表示正式发布的版本。

例子：

- 开发版本：0.1.0-SNAPSHOT、0.2.0-SNAPSHOT、2.1.0-SNAPSHOT
- 稳定版本：
    - 候选发布版本：0.1.0-RC1、1.2.0-RC2
    - 正式发布版本：0.1.0-RELEASE、0.1.1-RELEASE、0.1.2-RELEASE、1.2.0-RELEASE

git-flow流程介绍

Prong Boot 和 Prong Cloud 项目遵循的是 git-flow 的分支流程规范。git 客户端我们建议使用 SourceTree，因为SourceTree 对 `git-flow` 提供了内置的可视化支持，而不需要你去记住一大堆的命令。菜单入口是 `仓库` - `git-flow`，不同的 SourceTree 版本可能会有差异。

> 如果你坚持用命令行，可以参考这里。

`git-flow` 的总体流程示意图如下：

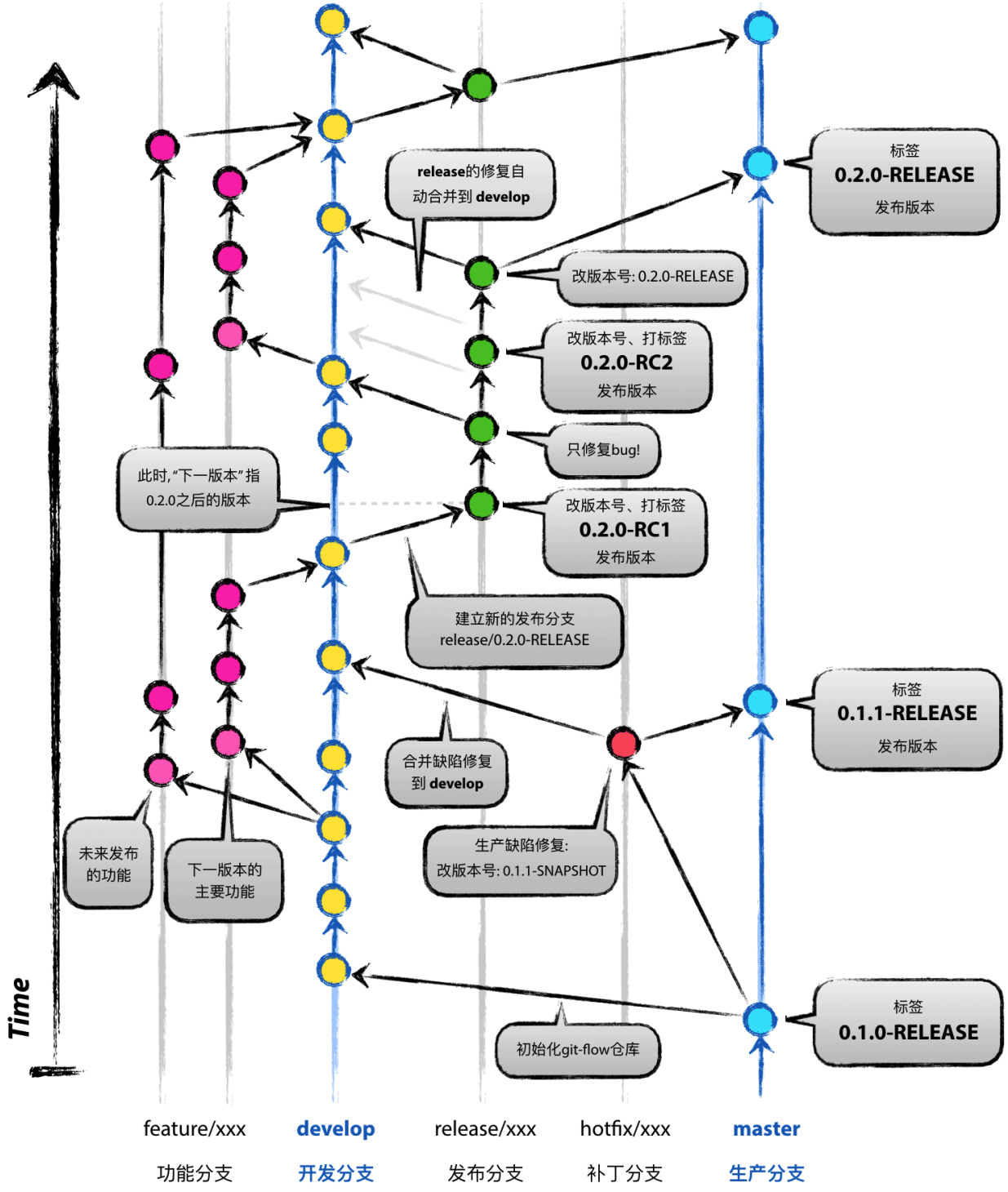

image.png

branch（分支）

在 `git-flow` 中，我们有两个永久分支：

- develop：开发分支。日常开发使用的分支，项目协作者主要工作在这个分支上，同时所有的 pull request 都应该发到这个分支；
- master：主分支。所有提供给用户使用的正式版本（RELEASE），都在这个主分支上发布。

其实，永久分支只需要这两条就够了，不需要其他了。但是，除了永久分支以外，还有一些临时分支，用于应对一些特定目的的版本开发。临时分支主要有三种：：

- **feature/xxx**：功能分支。它是为了开发某种特定功能，从**develop**分支上面分出来的，开发完成后，要再并入**develop**。功能分支可以有多个，这些分支通常只是个人使用，存在开发者本地库中，如果需要多人协作，则需要推送到远程仓库；
- **release/xxx**：发布分支。它是指发布正式版本之前（即合并到**master**分支之前），我们可能需要有一个预发布（RC）的版本进行测试。发布分支对代码进行了封版，不允许在发布分支上开发新功能，只允许修复测试发现的bug；
- **hotfix/xxx**：补丁分支。软件正式发布以后，难免会出现bug。这时就需要创建一个分支，进行bug修补。补丁分支是从**master**分支上面分出来的，修补结束以后，再合并进**master**和**develop**分支。

tag（标签）

标签是用于对应每个预发布版本或发布版本的版本标识，即 `x.y.z-RCx` 或 `x.y.z-RELEASE`

开发角色

我们在项目中定义两类角色：

开发工程师

作为开发工程师，你有两类分支：

- **develop **开发分支，用于下一个发布版本。
- **feature/xxx** 功能分支，用于开发新功能 **xxx**，根据需要可能会有多个功能分支。

新功能开发流程

要开发下一个版本的功能，你要从develop分支开出新的功能分支，最后合并回develop分支，如此往复：

```
*   4. (develop) 合并 'feature/work-with-correcting-a' 到 develop
|\
| * 3. (feature/work-with-correcting-a) Correcting a
|/
*   2. 合并 'feature/work-with-a' 到 develop
|\
| * 1. (feature/work-with-a) a
|/
*
```

> 注意示意图顺序都是从下往上看！
>
> 为了更轻松地与其他开发工程师合作开发一个大功能，你可以从这个功能分支再开出新的功能分支。你可以将这个大型功能分支叫做**集成分支**。

- 在该 **feature** 分支上进行开发，提交代码，如需多人协作，push 该分支到远程（origin）仓库。
- 你应定期*rebase*（变基）或合并 **develop** 分支的代码到你的 **feature** 分支，使你的代码保持最新，并避免合并冲突。
- 你应在完成某个功能后，尽可能快地合并 **feature** 分支代码到 **develop** 分支并删除该 **feature** 分支，快速传播你的提交以避免合并冲突。

配置管理员

作为配置管理员，你有两类分支：

- **master **生产分支，代表正式发布的版本（生产版本）。
- **release/xxx** 发布分支，用于准备发布版本。

版本发布流程

当准备封版的时候，假设你这次要发布的版本是 **0.2.0-RELEASE**，你需要：

- 创建一个候选发布版本（*release candidate*），即从 SourceTree 中选择“建立新的发布版本”，`发布版本号` 填写 `0.2.0-RELEASE`。
- 将 **release/0.2.0-RELEASE** 分支的版本号修改为 **0.2.0-RC1**，发布到测试环境进行测试。
- 将 **develop** 分支的版本号修改为下一个版本，即将 **develop** 分支的版本号修改为 **0.3.0-SNAPSHOT**。

```
| *  3. 发布 RC1 到测试环境测试
| *  2. (release/0.2.0-RELEASE) 修改版本号为 0.2.0-RC1，打标签: 0.2.0-RC1
* |  1. (develop) 修改版本号为 0.3.0-SNAPSHOT
|/
```

注意，**release/xxx** 发布分支起到了**冻结**代码的作用，此时不应在这个分支上开发新的功能，`RC1` 版本如果测试没有问题，就可以发布了。但如果有bug需要修复，有以下几种做法：

- 从 **develop** 进行 *Cherry Pick*（遴选）
  这是最漂亮的，但也可能不可行，因为合并冲突可能会阻碍。 还有一个风险是 **develop** 分支太遥远，因此很难知道修复程序在 **release** 分支中是否会起作用。 但我觉得它为**配置管理员**提供了对发布过程的最大控制权。
- 开发工程师从 **release** 分支开一个新分支并合并回去
  这通常是在大型团队中的首选。 它使**配置管理员**可以控制发布中包含的内容，避免了合并冲突的风险。 可以信任在该 **feature** 分支上完成的测试。 可以通过PR的方式（pull requests）来完成这一切。
- 直接提交到 **release** 分支
  这种适合纯技术型团队，例如**配置管理员**同时也是**开发工程师**。 在 **release** 分支上修复bug的同时，**开发工程师**也可以在 **develop** 上继续开发下一个版本。这种做法缺乏代码审查，但在小项目中可能也不需要。

如果RC1还有bug，则需要在修复后发布下一个版本 RC2。

如果你用到 *Cherry Pick*（遴选），大概是这样子：

```
| *  7. 发布 RC2 到测试环境测试
| *  6. 修改版本号为 0.2.0-RC2，打标签: 0.2.0-RC2
| *  5. 修改RC1中的bug
| *  4. 修改版本号为 0.2.0-SNAPSHOT
| *  3. 发布 RC1 到测试环境测试
| *  2. (release/0.2.0-RELEASE) 修改版本号为 0.2.0-RC1，打标签: 0.2.0-RC1
* |  1. (develop) 修改版本号为 0.3.0-SNAPSHOT
|/
```

> 其中，第4、5两个commit是从 **develop** 进行 *Cherry Pick*过来的。

如果RC2版本测试OK，就可以准备发布生产版本了。

- 修改 **release/0.2.0-RELEASE** 分支的版本号为 **0.2.0-RELEASE**。
- 在SourceTree中，选择 `完成发布版本` ，此时SourceTree会自动做以下工作：
  - 将 **release/0.2.0-RELEASE** 分支代码以及RC1、RC2标签合并到 **master** 分支。
  - 将 **release/0.2.0-RELEASE** 分支代码合并到develop。

因为合并后 **develop** 的版本号被改成 **0.2.0-RELEASE** 了，所以此时需要将 **develop** 的版本号改回下一个开发版本，如 **0.3.0-SNAPSHOT**。

打补丁流程

> 并非每个 bug 都有打补丁的必要，对于不紧急的 bug，可以在 **develop** 里修复后随下一个版本发布。

打补丁是指对提供给用户使用的正式版本进行修复。

假设要对正式版本 **0.1.0-RELEASE** 打补丁。

- SourceTree中，选择 `建立新的修复补丁` ，修复补丁版本为 `0.1.1-RELEASE` ，此时SourceTree 从 **master** 分支创建了一个补丁分支。
- 在补丁分支上修复bug，修改bug的三种做法参考版本发布流程。
- 将 **hotfix/0.1.1-RELEASE** 分支的版本号修改为 **0.1.1-RC1**，发布到测试环境进行测试。
- 如果测试有bug，重复以上第2、3步，其中第3步中的RC版本号进行递增。
- 测试没问题后，将 **hotfix/0.1.1-RELEASE** 分支的版本号修改为 **0.1.1-RELEASE**
- 在 SourceTree中，选择 `完成修复补丁` ，此时SourceTree会自动做以下工作：
  - 将 **hotfix/0.1.1-RELEASE** 分支合并到 master 分支。
  - 将 **hotfix/0.1.1-RELEASE** 分支合并到 develop 分支。
  - 在master 分支上打标签 **0.1.1-RELEASE**。
  - 最后，删除该补丁分支。

版本发布环境

下表列出了 Prong Boot 和 Prong Cloud 的在不同分支上发布版本时的代号，以及发布的目标环境。

git分支	feature分支	develop分支	release分支	hotfix分支	master分支
Prong Boot	SNAPSHOT，发布到本机	SNAPSHOT，发布到maven的snapshot仓库	RCx，发布到maven的release仓库	RCx，发布到maven的release仓库	RELEASE，发布到maven的release仓库
Prong Cloud	SNAPSHOT，发布到本机	SNAPSHOT，发布到开发测试环境	RCx，发布到准生产环境	RCx，发布到准生产环境	RELEASE，发布到准生产环境

Prong Boot 是一个开发脚手架，发布到maven仓库中，作为maven的组件来使用，用于快速开发一个业务微服务。

Prong Cloud 是一个微服务运行框架，发布docker的私有仓库中，再从 CaaS 平台拉取到相应运行环境。

Prong Boot规范

1、同一项目中所有模块版本保持一致

2、子模块统一继承父模块的版本

3、统一在顶层模块Pom的<dependencyManagement/>节中定义所有子模块的依赖版本号，子模块中添加依赖时不要添加版本号

4、开发测试阶段使用SNAPSHOT

5、生产发布使用RELEASE

6、新版本迭代只修改顶层POM中的版本

用于线上发布的代码，不可以使用包含“SNAPSHOT 版本”的依赖包，从而确保每次构建出的产物都是一致的。