**酒仙网版本管理规范**

**(v1.0)**

目 录

[1 文档介绍 2](#_Toc441583321)

[2 纳入版本管理的内容 2](#_Toc441583322)

[3 版本管理工具 3](#_Toc441583323)

[4 版本管理原则 3](#_Toc441583324)

[5 Git-flow版本管理流程 3](#_Toc441583325)

[5.1 工作方式 4](#_Toc441583326)

[Master/Develop分支（主分支） 4](#_Toc441583327)

[Master分支 4](#_Toc441583328)

[Develop分支 4](#_Toc441583329)

[Feature分支 5](#_Toc441583330)

[release分支 6](#_Toc441583331)

[Hotfix分支 7](#_Toc441583332)

[5.2 版本号命名规范 8](#_Toc441583333)

[5.3 提交代码规范 8](#_Toc441583334)

[5.4 多个版本共存 9](#_Toc441583335)

[5.5 长期版本和短期版本共存怎么管理 9](#_Toc441583336)

[5.6 CodeReview 9](#_Toc441583337)

[5.7 整体流程 10](#_Toc441583338)

[6 历史生产包管理 11](#_Toc441583339)

[7 环境管理 11](#_Toc441583340)

[8 架构分拆 12](#_Toc441583341)

[9 为什么两周一版本 12](#_Toc441583342)

[10 附录 14](#_Toc441583343)

[10.1 Git-flow示例(该流程没有演示CodeReview和开发Leader的流程) 14](#_Toc441583344)

# 文档介绍

|  |  |
| --- | --- |
| **文档目的** | 为了规范思源技术版本管理，减少版本错乱导致的生产环境事故，同时规范版本管理也能提升开发、测试、运维的效率。 |
| **使用范围** | 技术研发团队 |
| **读者范围** | 开发人员、测试人员、运维人员、产品人员 |

# 纳入版本管理的内容

生产环境代码、各种环境配置、测试代码、各种脚本等。

开发人员的代码严格按照git-flow流程进行版本管理。运维和测试的代码、脚本需要有版本工具管理，但不强制使用git-flow流程。

**注意：**运维和测试编写的脚本也需要纳入版本管理。

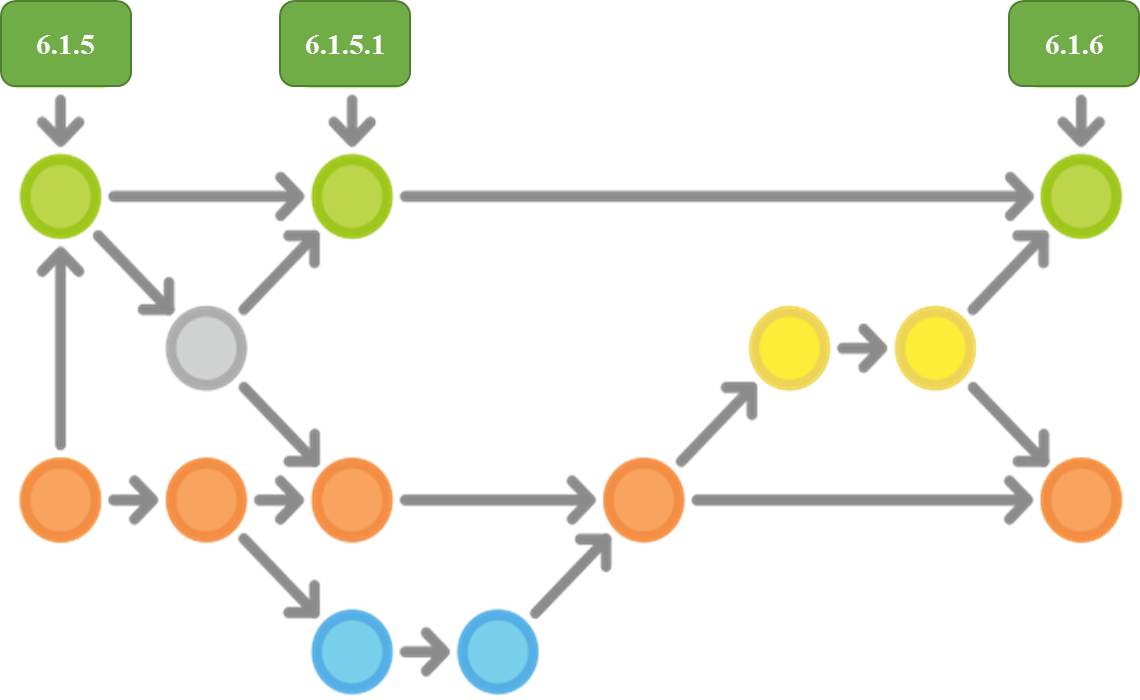
# 版本管理工具

公司统一使用git进行版本管理，使用git-flow流程作为版本管理流程，使用gitlab作为管理界面，使用gitlab进行codereview。以前使用svn做版本管理的团队需要逐步迁移到新的环境中。

# 版本管理原则

1. 分支数量在业务没有冲突情况下越少越好；
2. 分支存在时间越短越好。

# Git-flow版本管理流程



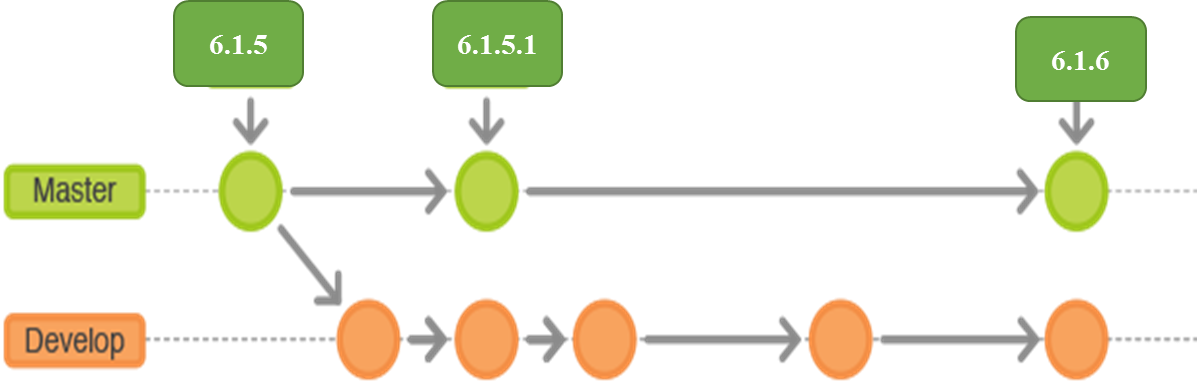
Git-flow工作流定义了一个围绕项目发布的严格分支模型，git-flow为各分支分配明确的角色，并定义分支之间如何交互。

## 工作方式

Git-flow工作流仍然用中央仓库作为所有开发者的交互中心。和其它的工作流一样，开发者在本地工作并push分支到要中央仓库中。

### Master/Develop分支（主分支）

Git-flow工作流使用2个分支来记录项目的历史。master分支存储了正式发布的历史，而develop分支作为功能的集成分支。



### Master分支

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。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，添加对应的版本号标签（TAG）。

### Develop分支

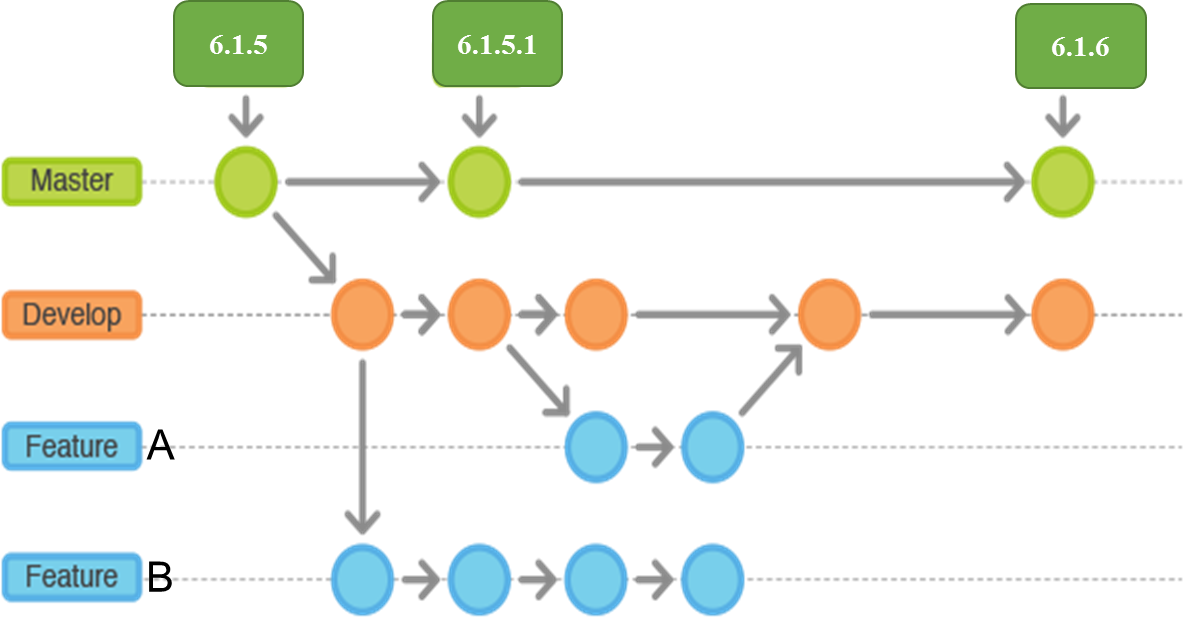
develop分支是保存当前最新开发成果的分支，这个分支上的代码可以时刻进行持续集成，确保所有的自动化测试能够通过。这个分支有时也可以被称作“integration branch”。

### Feature分支

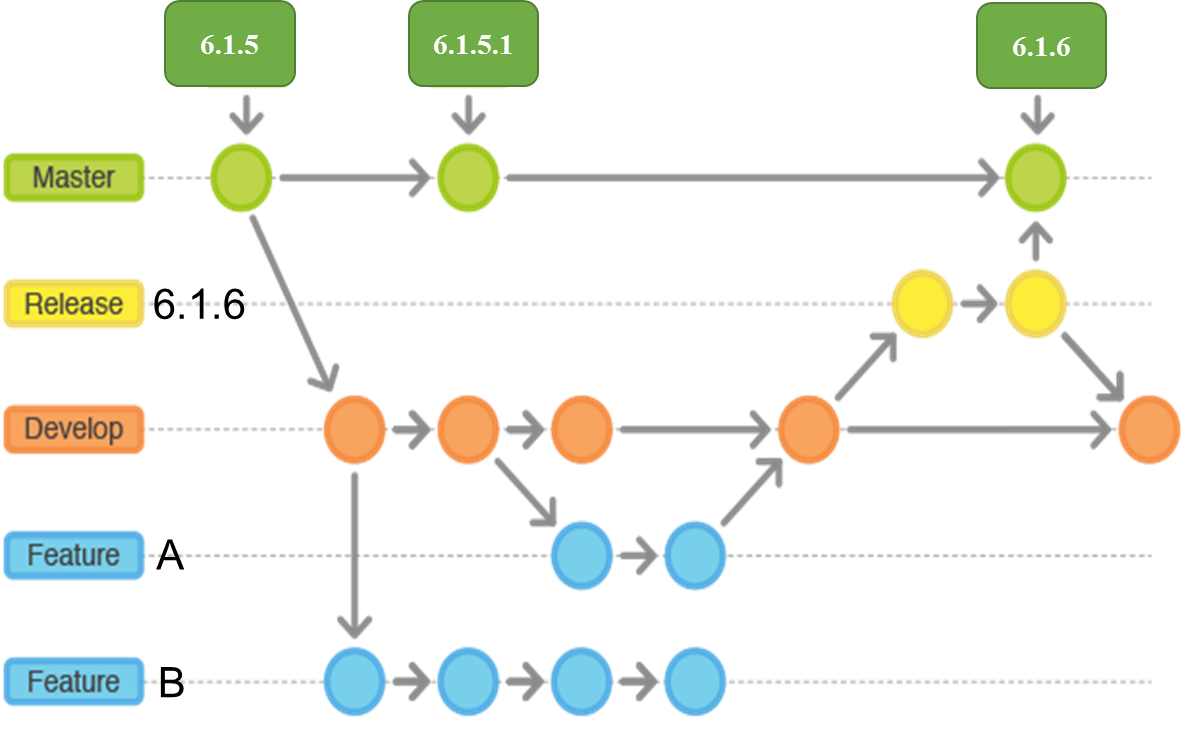
使用规范：

* 可以从develop分支发起feature分支
* 代码必须合并回develop分支
* feature分支的命名featrue/featruename

每个新功能有一个自己的分支，功能分支不是从master分支上拉出新分支，而是使用develop分支作为父分支。当新功能完成时，[合并回develop分支](https://www.atlassian.com/git/tutorial/git-branches#!merge)，并删除这个功能分支。新功能提交不直接与master分支交互，develop分支以及master分支开发人员不能直接提交代码（权限控制，只能通过技术Leader进行代码操作）。功能分支的命名需要起一个有意义的分支名，杜绝使用无意义的分支名。



### release分支



使用规范：

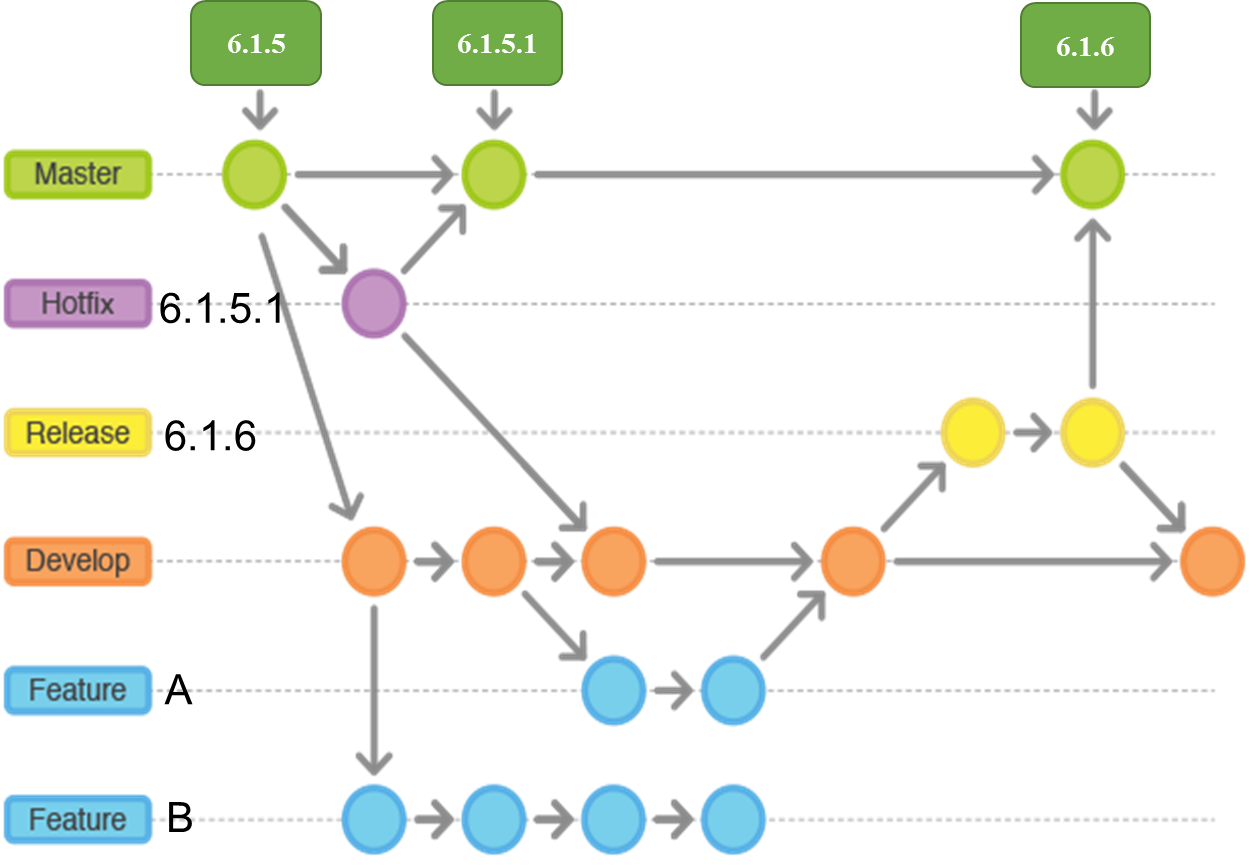
* 可以从develop分支派生
* 必须合并回develop分支和master分支
* 分支命名惯例：release/\*
* 开发和测试人员可以通过release分支与develop分支以及master分支，做影响分析。通过gitlab可以图形化显示

一旦develop分支上的功能产品人员验收通过后，开始提测，开发leader就在develop分支上拉一个release分支。Release分支命名为release/\*，\*为当前发布的版本号，如release/6.1.5。从这个时间点开始之后新的功能不能再加到这个分支上，这个分支只应该做Bug修复、文档生成和其它面向发布任务。开发人员在release分支上修复bug，测试通过后，技术leader通过gitlab或git命令Review一下release分支与master分支的差异，以及release分支与develop分支的差异，检查release分支新修改的代码的合理性。然后进行对外发布，接着将release分支合并到master分支并分配一个版本号打好Tag。同时将新建release分支以来做的修改合并回develop分支。使用一个用于发布准备的专门分支，使得团队可以在完善当前的发布版本的同时，同时可以继续开发下个版本的功能。

### Hotfix分支

使用规范：

* 可以从master分支派生
* 必须合并回master分支和develop分支
* 分支命名惯例：hotfix/\*



热修复（hotfix）分支用解决线上的**紧急必须快速修复**的bug，直接从master分支拉出一个分支。修复完成，马上合并回master分支和develop分支，master分支应该用新的版本号打好Tag。（hotfix的版本号一般为4位版本号，区别开与正常的版本号）

## 版本号命名规范

正常版本使用3位版本号，如6.1.5

紧急从hotfix出来的版本使用4位版本号，如：6.1.5.1

## 提交代码规范

1. 每次Commit代码在Comment上面必须写明改变内容，如：fix什么bug，新功能说明功能具体的功能。不允许无意义的Comment；
2. Push代码到生产服务器前，必须自测通过。
3. 每3天每人至少发起一次Merge代码到Develop分支的 Request。通过技术Review通过后进入Develop分支，除长期版本分支外。

## 多个版本共存

1. Git-flow流程建议最多有两个版本同时开发，1个版本处于release分支，1个版本处于develop分支；
2. 不建议存在更多同时开发的版本，如果超过2个以上同时开发的版本，需要拉一个独立的featrue分支进行某版本的开发。

## 长期版本和短期版本共存怎么管理

1. 不鼓励超过2周以上的长期项目存在，如果项目超过2周以上，需要将大项目拆成小项目分期上线，看线上效果再决定后续开发。
2. 实在需要存在长期项目，无法拆分的。可以拉一个稍微长期的featrue分支作为长期版本。

## CodeReview

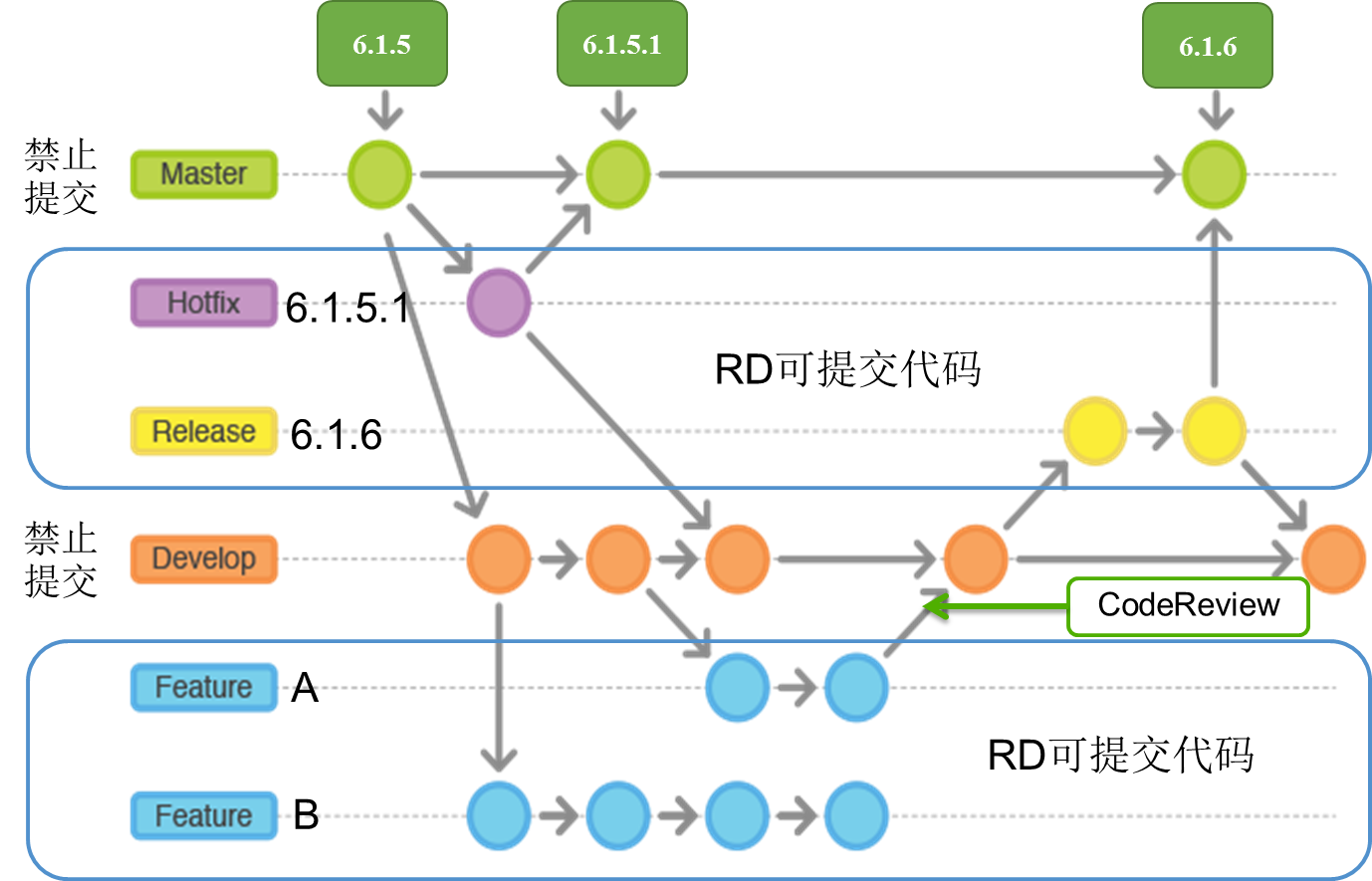
**开发人员写的任何代码都需要有人Review通过才能进入Develop分支。**

**代码标准：各团队依据编程语言制定代码规范，通过CodeReview和处罚措施（发红包）确保规范能够落地。**

**CodeReview流程：**

1. 开发人员开始一个新功能开发，从Develop分支拉出一个Featrue分支进行开发；
2. Featrue分支开发结束（原则上不能超过3天），自测通过后，通过gitlab提交合并代码到Develop分支的请求；
3. 团队技术骨干（每个模块至少有两个以上技术骨干有Review代码的权限）通过gitlab Review代码，Review没有通过的代码，开发人员需要修改问题重新提交Review请求。Review通过的代码，自动进入Develop分支。

## 整体流程



**版本管理流程归纳如下：**

1. 开发人员从Develop分支拉出各自功能的featrue分支；
2. 开发人员在featrue分支上开发、自测通过后，通过gitlab发起合并代码到develop分支请求；
3. 资深开发人员对合并代码请求进行Review，通过的请求进入develop分支，没有通过需要对应开发人员继续修改代码直到通过为止；
4. 开发人员继续拉featrue分支开发其它功能；
5. 等本版本所有功能通过review进入develop分支，自测联调通过，产品初步验收通过后，开发leader从develop分支拉出一个release/xxxx分支进行提测，本版本的新功能冻结；
6. 测试人员通过release/xxxx分支打版本测试，开发人员在release分支上修改bug；
7. 等测试通过、确定发布后，开发leader将release代码merge到develop分支和master分支，并在master分支打上版本号tag；
8. 线上版本出现紧急bug（必须要马上修改上线），开发leader从master拉hotfix/xxxx分支。修改测试通过后上线。合并hotfix/xxx分支代码到master分支，并打tag。同时合并hotfix代码到develop分支。

# 历史生产包管理

1. 运维人员统一设置一个目录，保存历史版本，每个版本需要包括该系统/服务的**全量**包，确保可以快速拿到生产环境的所有历史包。
2. 标注清楚上线日期，以及统一的版本号。

# 环境管理

1. 每个产品团队有独立的开发、测试、预发布环境，共用的服务可以通用；
2. 开发环境由开发人员管理，测试环境由测试人员管理，预发布环境由运维人员管理，每个角色都能独立进行环境的部署；开发人员的环境可以使用自己的电脑作为服务器，调用后端开发环境的缓存、数据库以及公共服务进行调试；
3. 开发、测试环境的部署脚本相同；
4. 支持一键部署测试环境，一键部署开发环境，不需要其它手工参与；
5. 测试的版本来自release分支，开发的版本来自develop分支；
6. 不允许开发人员在IDE环境给测试人员打包；
7. 开发代码中不应该存在与环境相关的配置，所有的环境相关的配置放在独立的配置文件中；
8. 建议配置文件可以分成开发环境、测试环境、预发布环境、生产环境多个配置，配置文件通用在代码库中管理；
9. **基础共用的服务**除了测试环境外，还需要一套更稳定的环境（一般为生产环境的包），用于支持其它产品线的开发和测试；
10. **最佳实践：**通过Jenkins实现自动化打包，自动化测试，自动化部署。整个过程不能超过15分钟。

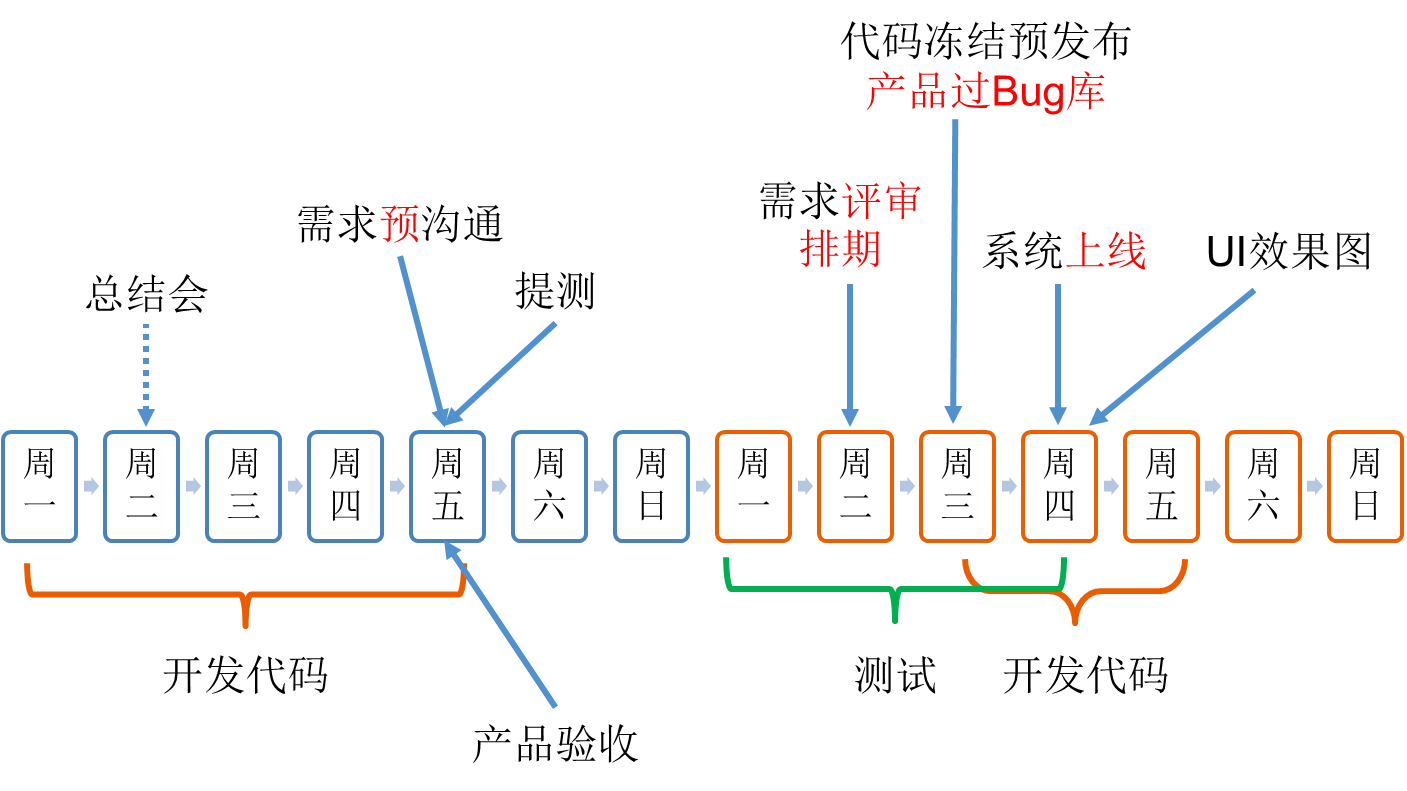
# 架构分拆

1. 对于多个系统共用的服务，需要从架构上拆分出来独立编译独立上线，同时确保这些服务能够兼容历史版本。测试一键部署公共服务后，再一键部署前端系统就可以进行联调。

# 为什么两周一版本

两周一版本意味着小项目合并成一次上线，大项目拆分成小项目上线。

两周一次的系统上线，可以参考下图（研发四部迭代项目管理）进行项目管理：



**为什么两周发布一个版本：**

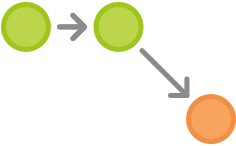
1. **提升业务的应变能力**，每两周就能决定一次下一个迭代开发的内容；
2. 减少大型项目风险，迭代上线，强迫拆分，优先业务优先上线
3. 固定时间的版本，提高项目管理的**可预知性**；
4. 减少版本管理的复杂度；
5. 减少并行项目导致的环境混乱以及测试资源重复测试的问题；
6. 减少测试人员的重复劳动；
7. 减少上线频度，提高上线质量。

# 附录

## Git-flow示例(该流程没有演示CodeReview和开发Leader的流程)

下面示例演示如何管理单个发布循环。假设你已经创建了一个中央仓库。

**创建开发分支**



第一步为master分支配套一个develop分支。简单来做可以[本地创建一个空的develop分支](https://www.atlassian.com/git/tutorial/git-branches#!branch)，push到服务器上：

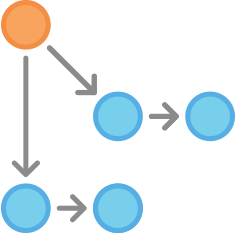
git branch develop  
git push -u origin develop

以后这个分支将会包含了项目的全部历史，而master分支将只包含了部分历史。其它开发者这时应该[克隆中央仓库](https://www.atlassian.com/git/tutorial/git-basics#!clone)，建好develop分支的跟踪分支：

git clone ssh://user@host/path/to/repo.git  
git checkout -b develop origin/develop

现在每个开发都有了这些历史分支的本地拷贝。

**小红和小明开始开发新功能**

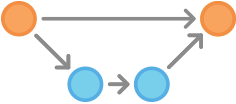


这个示例中，小红和小明开始各自的功能开发。他们需要为各自的功能创建相应的分支。新分支不是基于master分支，而是应该[基于develop分支](https://www.atlassian.com/git/tutorial/git-branches#!checkout)：

git checkout -b some-feature develop

他们用老套路添加提交到各自功能分支上：编辑、暂存、提交：  
git status  
git add  
git commit

**小红完成功能开发**

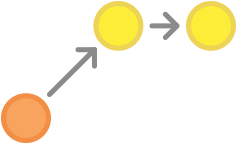


添加了提交后，小红觉得她的功能OK了。如果团队使用Pull Requests，这时候可以发起一个用于合并到develop分支。否则她可以直接合并到她本地的develop分支后push到中央仓库：

git pull origin develop  
git checkout develop  
git merge some-feature  
git push  
git branch -d some-feature

第一条命令在合并功能前确保develop分支是最新的。注意，功能决不应该直接合并到master分支。冲突解决方法和[集中式工作流](http://blog.jobbole.com/76847/)一样。

**小红开始准备发布**



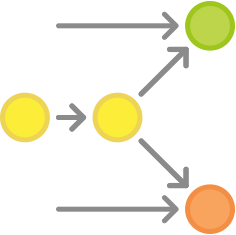
这个时候小明正在实现他的功能，小红开始准备她的第一个项目正式发布。像功能开发一样，她用一个新的分支来做发布准备。这一步也确定了发布的版本号：

git checkout -b release/0.1 develop

这个分支是清理发布、执行所有测试、更新文档和其它为下个发布做准备操作的地方，像是一个专门用于改善发布的功能分支。

只要小红创建这个分支并push到中央仓库，这个发布就是功能冻结的。任何不在develop分支中的新功能都推到下个发布循环中。

**小红完成发布**



一旦准备好了对外发布，小红合并修改到master分支和develop分支上，删除发布分支。合并回develop分支很重要，因为在发布分支中已经提交的更新需要在后面的新功能中也要是可用的。另外，如果小红的团队要求Code Review，这是一个发起Pull Request的理想时机。

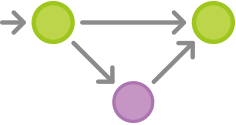
git checkout master  
git merge release/0.1  
git push  
git checkout develop  
git merge release/0.1  
git push  
git branch -d release/0.1

发布分支是作为功能开发（develop分支）和对外发布（master分支）间的缓冲。只要有合并到master分支，就应该打好Tag以方便跟踪。

git tag -a 0.1 -m "Initial public release" master  
git push --tags

Git有提供各种勾子（hook），即仓库有事件发生时触发执行的脚本。可以配置一个勾子，在你push中央仓库的master分支时，自动构建好对外发布。

**最终用户发现Bug**



对外发布后，小红回去和小明一起做下个发布的新功能开发，直到有最终用户开了一个Ticket抱怨当前版本的一个Bug。为了处理Bug，小红（或小明）从master分支上拉出了一个维护分支，提交修改以解决问题，然后直接合并回master分支：

“`  
git checkout -b issue-#001 master

**Fix the bug**

git checkout master  
git merge issue-#001  
git push

就像发布分支，维护分支中新加这些重要修改需要包含到develop分支中，所以小红要执行一个合并操作。然后就可以安全地[删除这个分支](https://www.atlassian.com/git/tutorial/git-branches#!branch)了：

git checkout develop  
git merge issue-#001  
git push  
git branch -d issue-#001