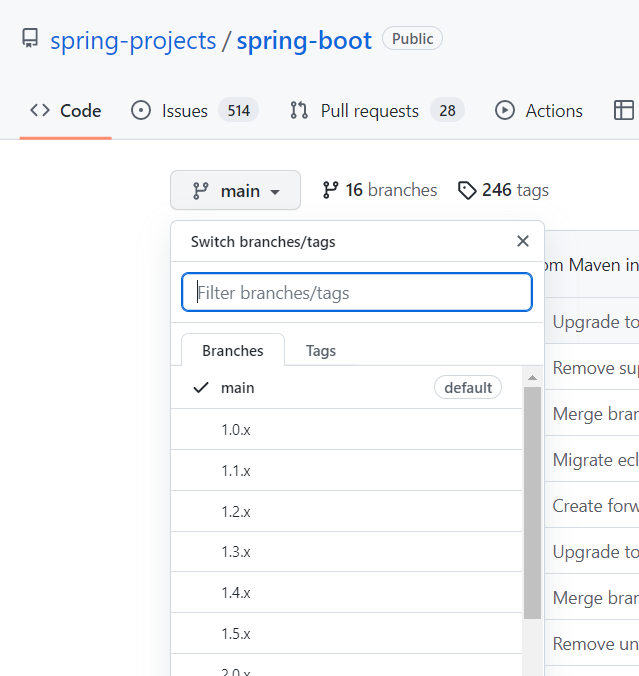
# 分支命名规范

master：系统版本的准线，版本通过测试并处于可发布状态时，才可合并入master，一直维持可构建状态。该分支管理员使用。

dev：协同开发分支，可直接提交，所有开发都在这里提交最新代码的分支。

如1.2.x ： 第一位为重大版本号、 第二位为需求版本号、最后一位x固定表示bug的修复不固定。（如果有些项目是3位定需求的，如1.2.1.x 。 总之就是最后一位是.x）



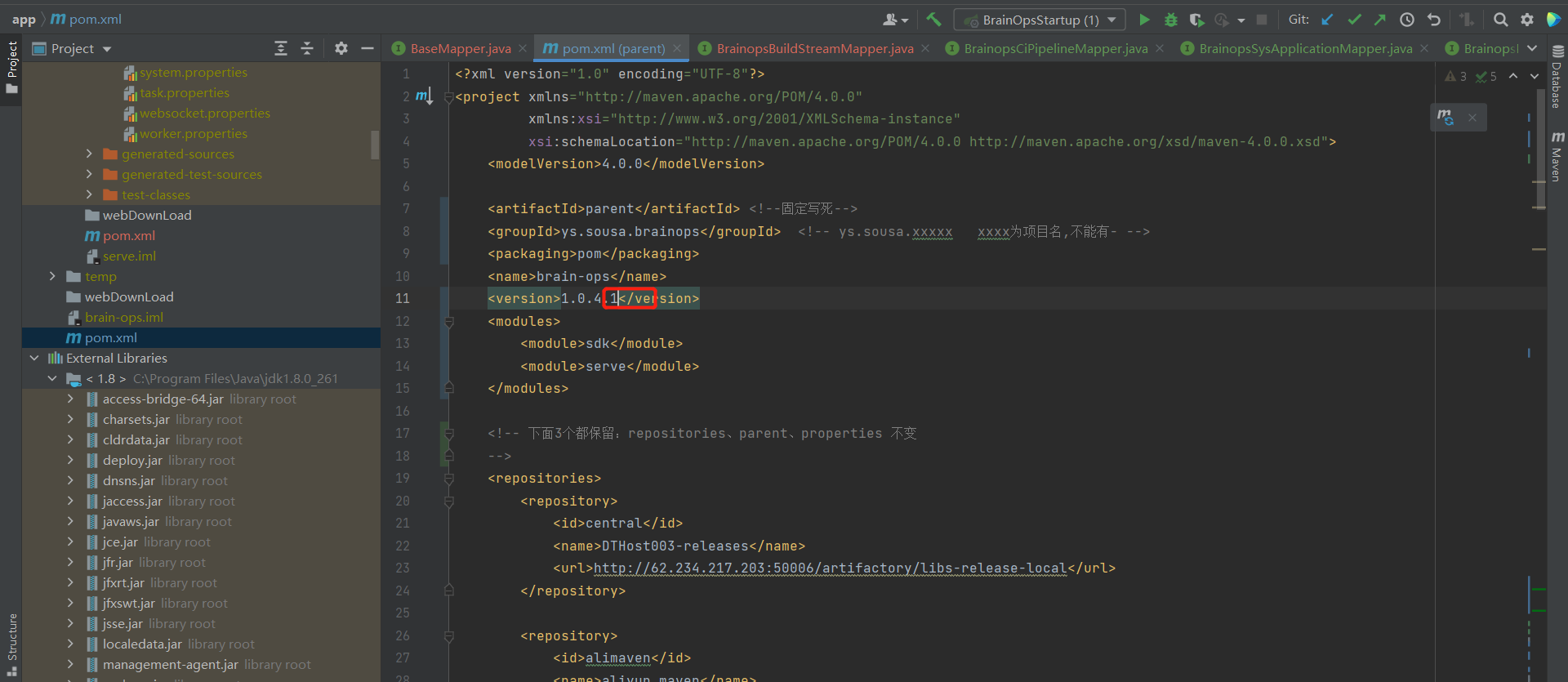
# Tag/release 规范

v1.2.x ： v开头以区分分支名，后面和分支版本对应，是具体对应的bug级别版本号。

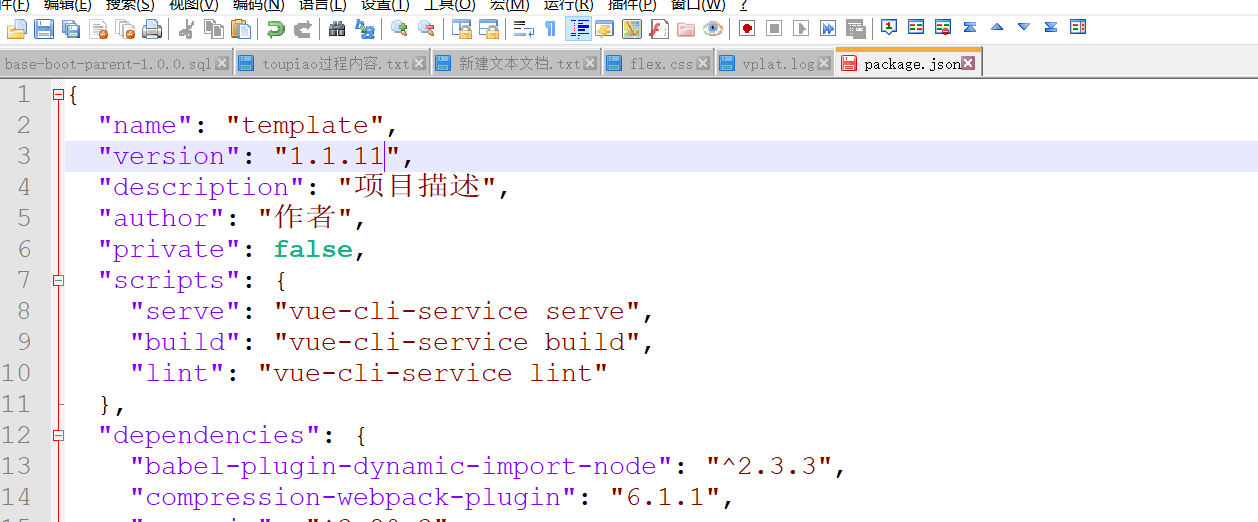
不能带有x同时不能断档，保持数字的自增。如下图。



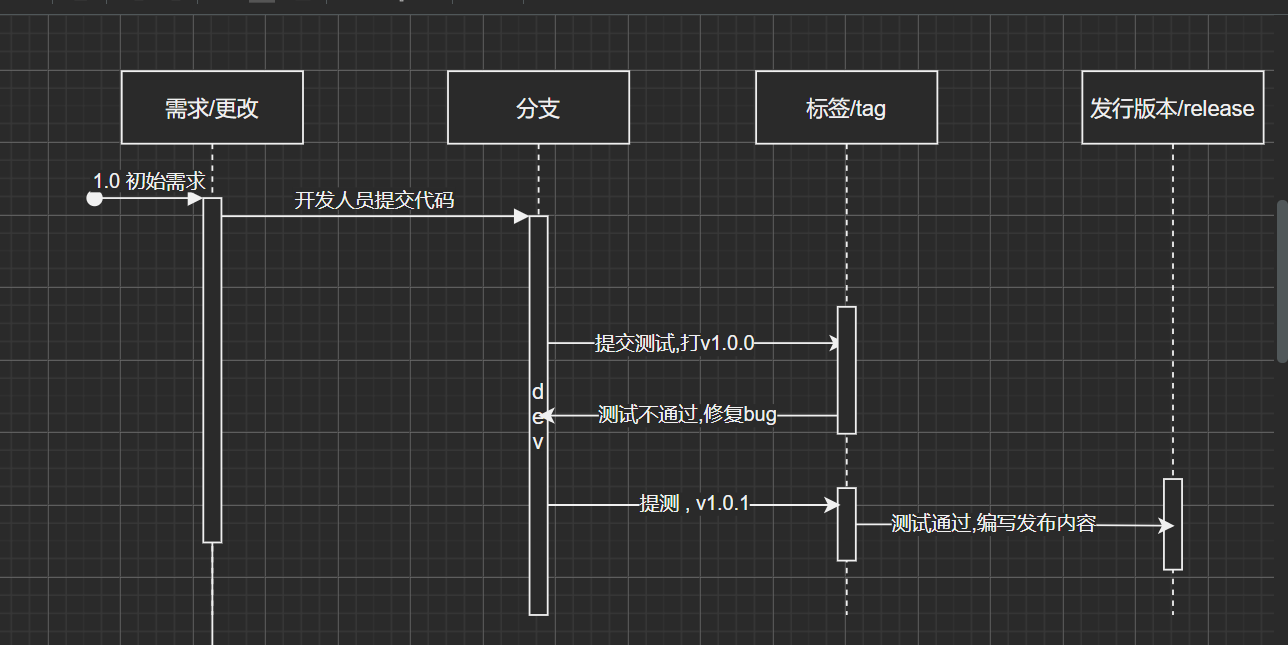
后端同时需改pom.



前端修改package.json



# 版本/tag/release管理



编写的发布内容应尽可能，简介但是全面。



# 新部署的内容

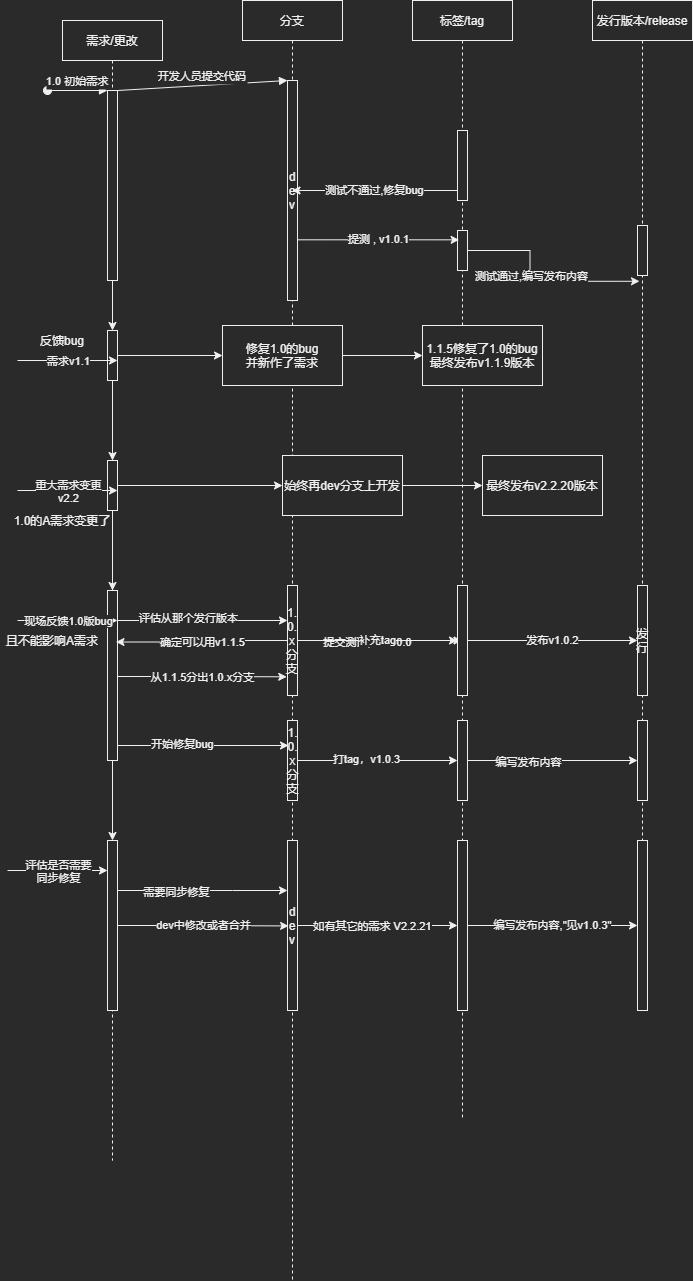
新的部署都以需求版号，最新发布的release 为准。

以保证改需求版本内是最新修复的版本、且可用的版本。

如发布1.0版本。 应该早到最新的1.0.1 属于发布的为准。

# 单分支切换到多分支管理

说明: 多分支管理在维护和同步代码中有很多不便。应该尽量避免。



## 功能需不需要再向上同步。

1. 需要向上同步: 则在dev和对应的1.0.x分支中同步修复。 同时tag根据也需要做小的tag管理。
2. 不需要向上同步: 只在对应的1.0.x分支中修复就行啦。
3. 向上2个版本同步,其余最新版本不同步: 其他的版本都需要打对应的tag做标识修复。

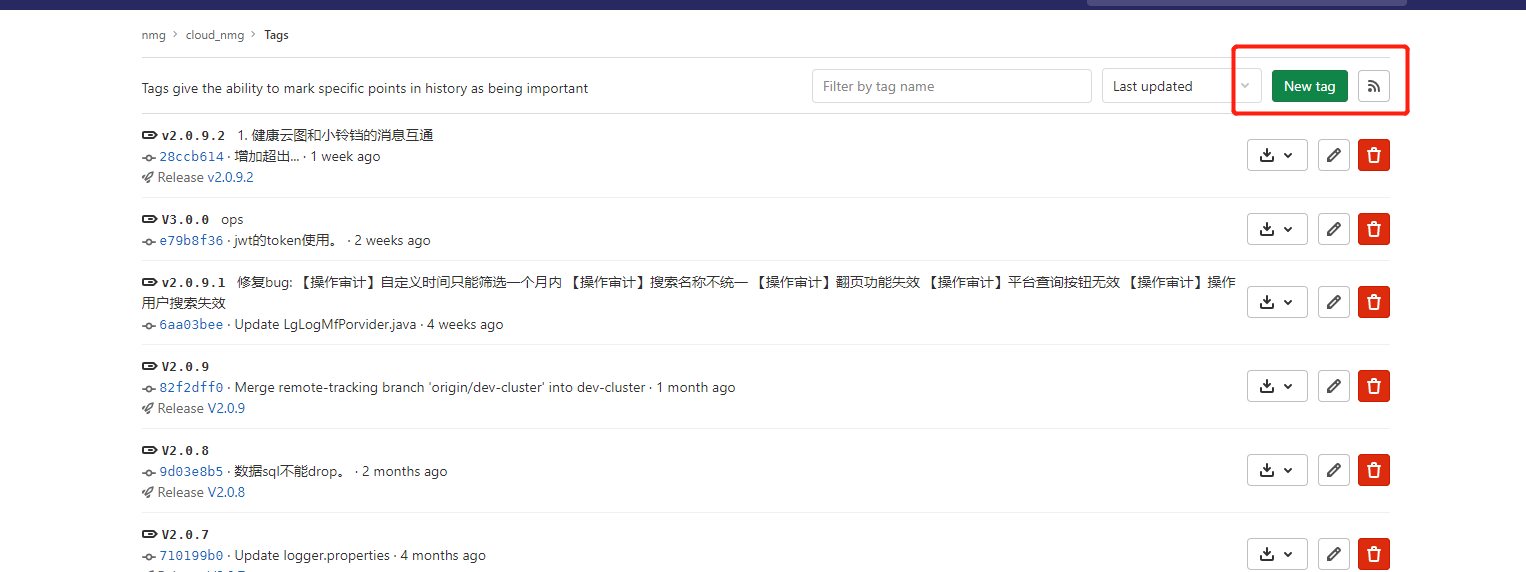
# 前后端项目分离后端版本管理

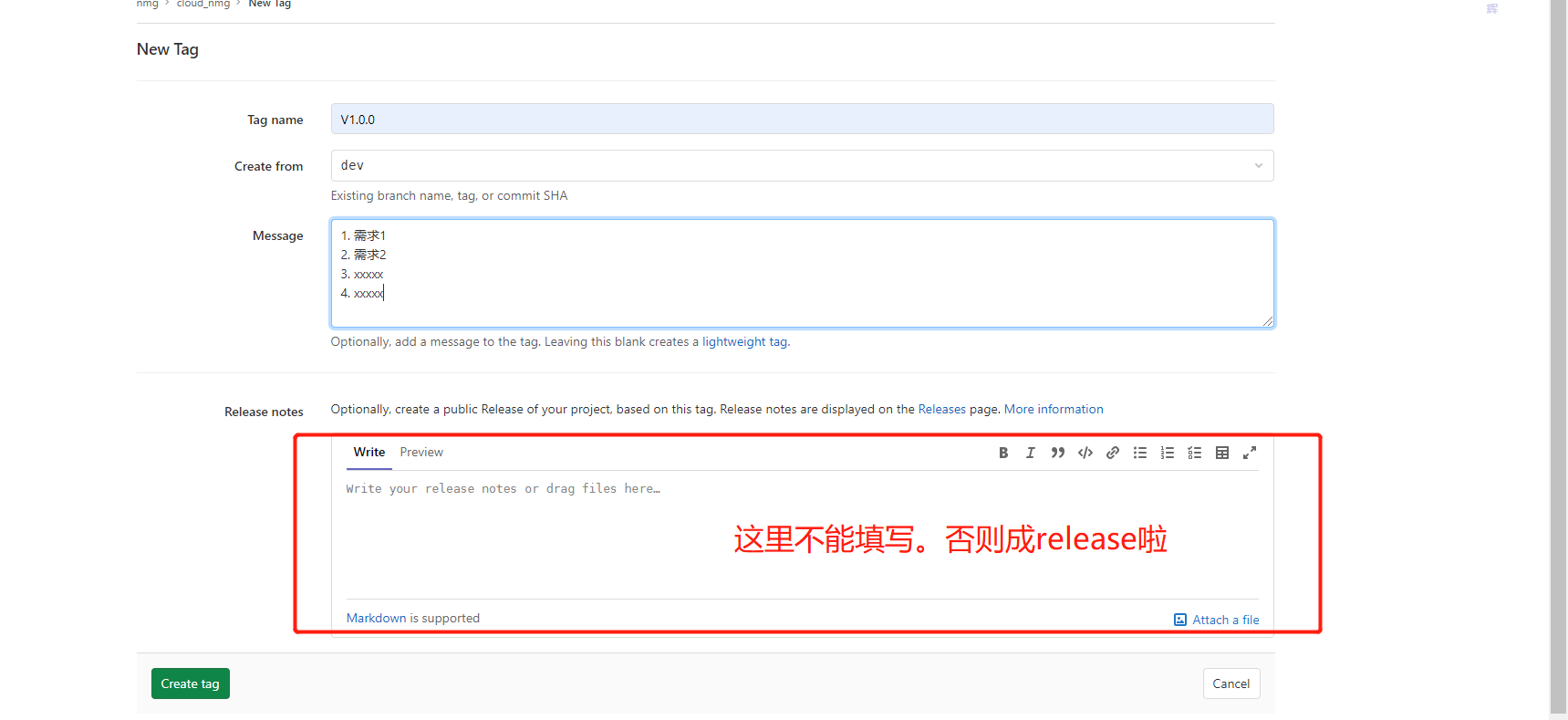
大的版本保持一致。也就是说需求版本保持一致。

1. 只有一端更新: 没有更新的项目版本号也要做空的升级。以满足需求版本保持。
2. 2端都更新: 都做需求版本号的更新。 各自的bug各自维护自己的x的版本内容。总体需求对接的接口和需求保持一致。
3. 部署时对应的版本: 同上[新部署的内容](#_新部署的内容)

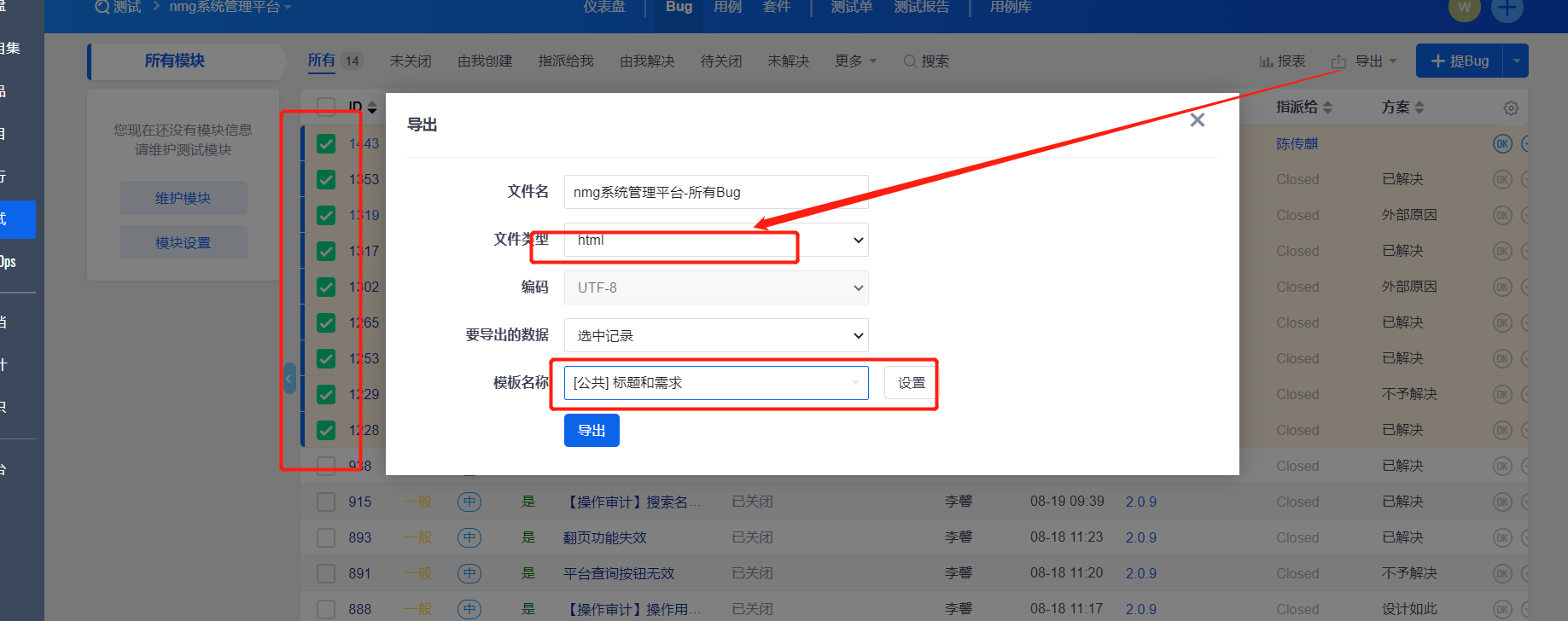
# Gitlab的操作

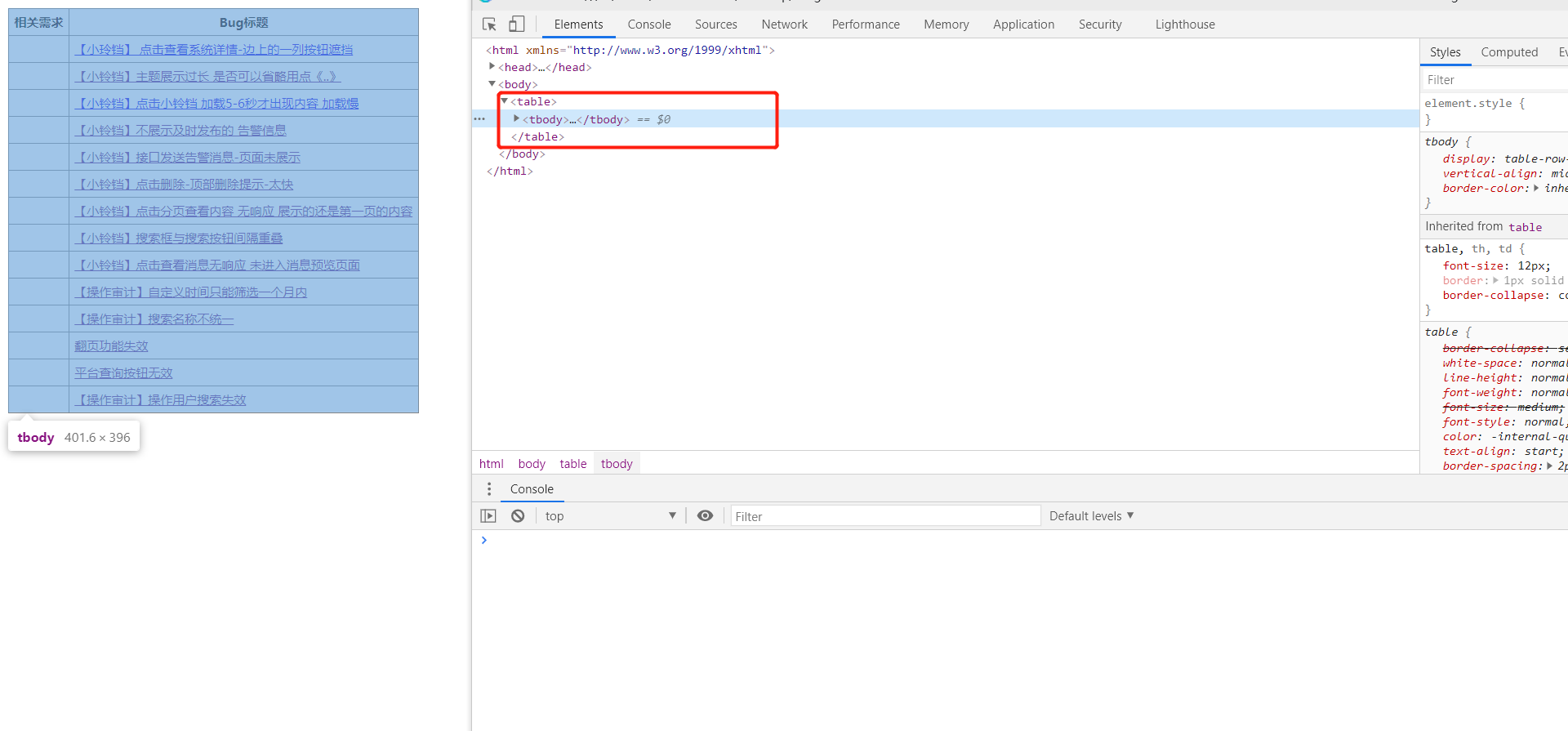
## 打tag



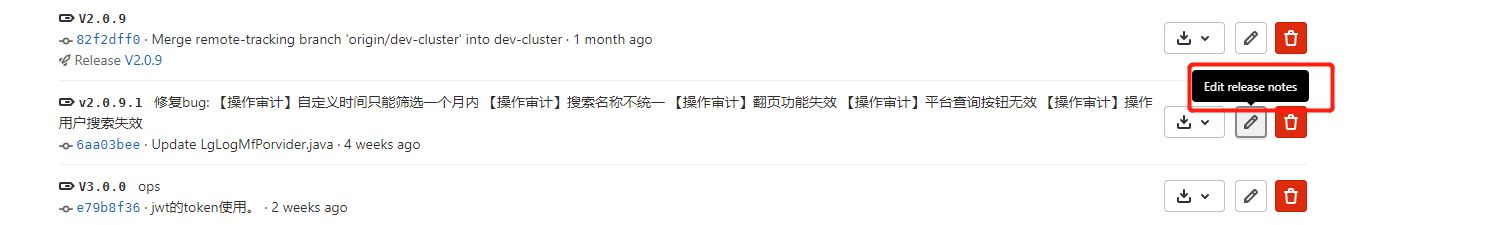


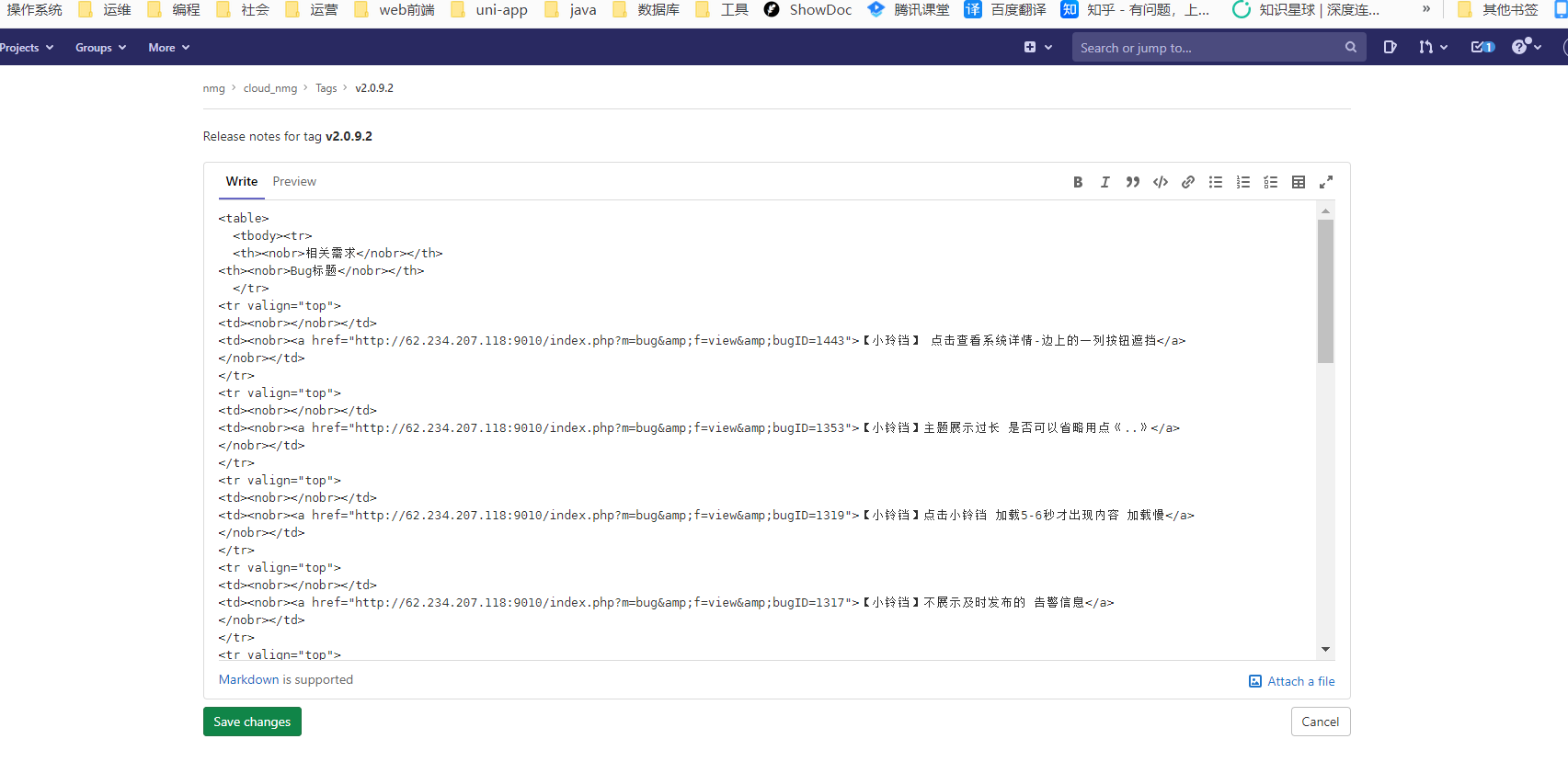
## Release的内容编写





把复制的table放到





## New Branch



# 对应CI/CD

1. 正常开发提交可以运行的代码 => 持续化集成
2. 打tag，提测的过程 => 持续部署
3. 发行release的过程 => 持续化发布
4. 提交给现场实施 => 持续化交付

# 视频指导