# 项目规范

多人协作的应用在开发的时候需要有一个统一的规范

要求：

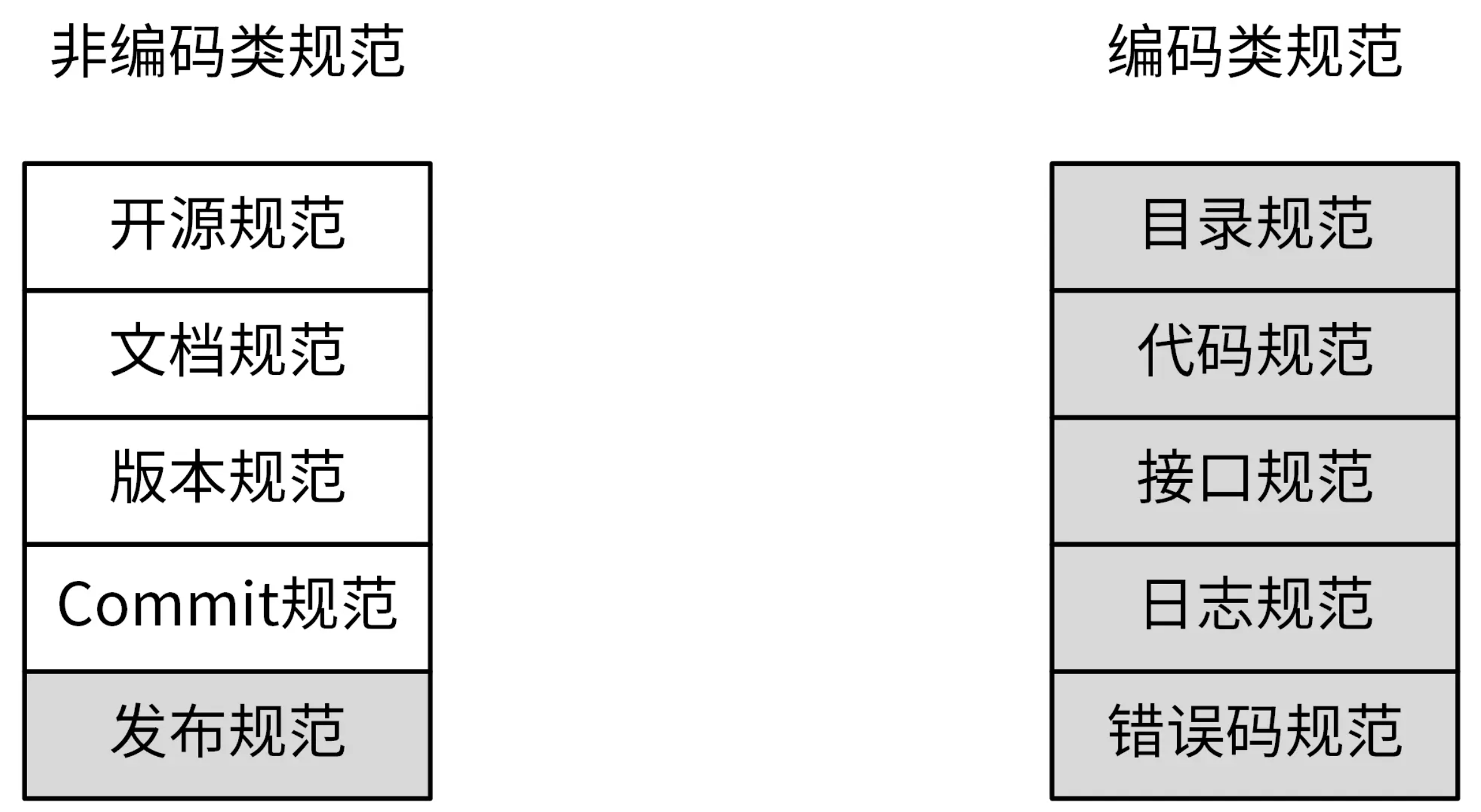
1. 代码风格统一，方便长期开发
2. 项目目录有序，可维护性
3. API接口统一，便于记忆
4. 错误码规范，减量聚类

规范需要在项目开始前一起来制定

# 规范分类

分为两大类规范

1. 非编码类规范
2. 编码类规范



# 非编码类规范

## 开源规范

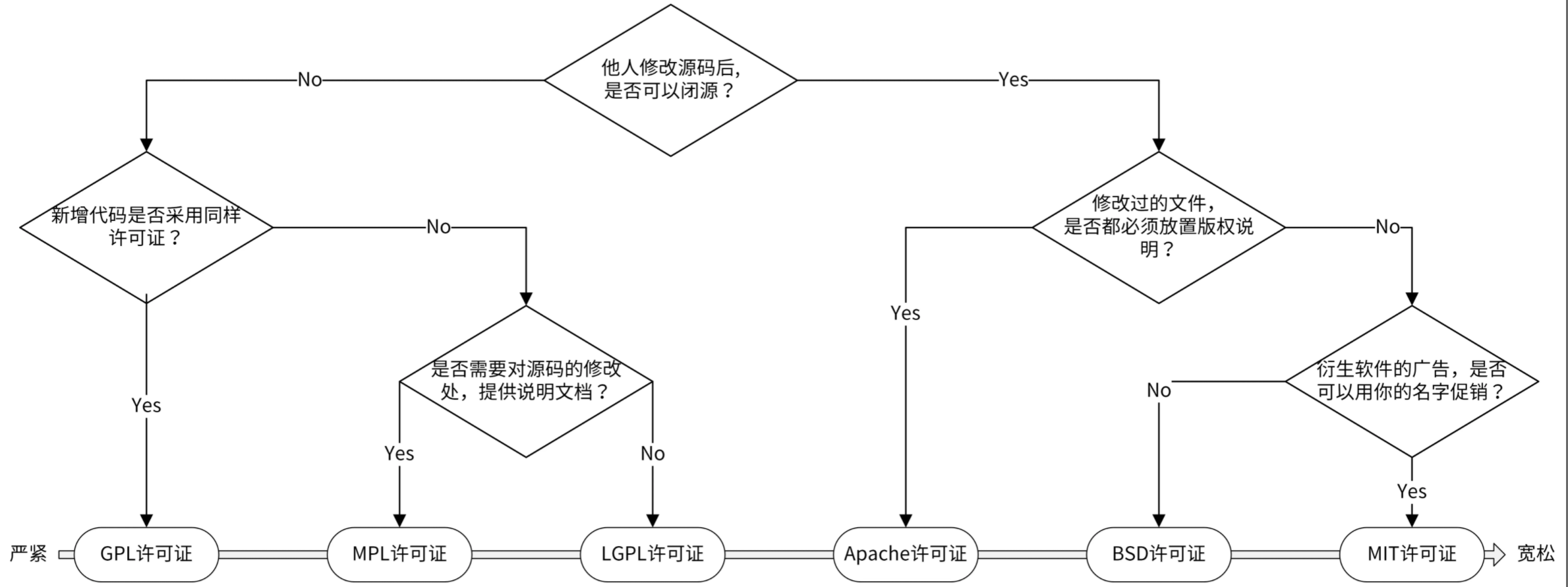
理由

1. 开源项目在代码质量、代码规范、文档等方面，要比非开源项目要求更高
2. 方便开源

开源协议

业界有上百种开源协议，没必要全都记住，只需要知道经常使用的 6 种开源协议，也就是**GPL、MPL、LGPL、Apache、BSD**和**MIT**

<https://github.com/marmotedu/geekbang-go/blob/master/%E5%BC%80%E6%BA%90%E5%8D%8F%E8%AE%AE%E4%BB%8B%E7%BB%8D.md>



因为 **Apache 是对商业应用友好的协议**，使用者也可以在需要的时候修改代码来满足需要，并作为开源或商业产品发布 / 销售，所以大型公司的开源项目通常会采用 Apache 2.0 开源协议。

开源规范特点

1. 第一，开源项目，应该有一个高的单元覆盖率。比如60%，保证代码质量
2. 第二，要确保整个代码库和提交记录中，不能出现内部 IP、内部域名、密码、密钥这类信息。
3. 第三，当我们的开源项目被别的开发者提交 pull request、issue、评论时，要及时处理
4. 第四，好的开源项目，应该能够持续地更新功能，修复 Bug。

<https://github.com/marmotedu/geekbang-go/blob/master/%E5%BC%80%E6%BA%90%E8%A7%84%E8%8C%83%E8%AF%A6%E7%BB%86%E5%88%97%E8%A1%A8.md>

开源项目提醒

第一件，如果有条件，你可以宣传、运营开源项目，让更多的人知道、使用、贡献代码。比如，你可以在掘金、简书等平台发表文章，也可以创建 QQ、微信交流群等，都是不错的方式。

第二件，如果你英文好、有时间，文档最好有中英文 2 份，优先使用英文，让来自全球的开发者都能了解、使用和参与你的项目。

## 文档规范

理由

1. 文档属于软件交付的一个重要组成部分，没有文档的项目很难理解、部署和使用。
2. 我认为项目中最需要的 3 类文档是**README文档**、**项目文档**和**API接口文档**。

### README 规范

README 文档是项目的门面，它主要是用来介绍项目的功能、安装、部署和使用的，所以它是可以规范化的。

规范参考模板

# 项目名称

<!-- 写一段简短的话描述项目 -->

## 功能特性

<!-- 描述该项目的核心功能点 -->

## 软件架构(可选)

<!-- 可以描述下项目的架构 -->

## 快速开始

### 依赖检查

<!-- 描述该项目的依赖，比如依赖的包、工具或者其他任何依赖项 -->

### 构建

<!-- 描述如何构建该项目 -->

### 运行

<!-- 描述如何运行该项目 -->

## 使用指南

<!-- 描述如何使用该项目 -->

## 如何贡献

<!-- 告诉其他开发者如果给该项目贡献源码 -->

## 社区(可选)

<!-- 如果有需要可以介绍一些社区相关的内容 -->

## 关于作者

<!-- 这里写上项目作者 -->

## 谁在用(可选)

<!-- 可以列出使用本项目的其他有影响力的项目，算是给项目打个广告吧 -->

## 许可证

<!-- 这里链接上该项目的开源许可证 -->

一个其他参考：<https://raw.githubusercontent.com/marmotedu/iam/master/README.md>

有个在线的 README 生成工具，你也可以参考下：**readme.so**。

<https://readme.so/>

### 项目文档规范

理由

项目文档包括一切需要文档化的内容，它们通常集中放在 /docs 目录下。当我们在创建团队的项目文档时，通常会预先规划并创建好一些目录，用来存放不同的文档。

两类文档

1. 开发文档：用来说明该项目的开发流程，比如如何搭建开发环境、构建二进制文件、测试、部署等。
2. 用户文档：软件的使用文档，对象一般是软件的使用者，内容可根据需要添加。比如，可以包括 API 文档、SDK 文档、安装文档、功能介绍文档、最佳实践、操作指南、常见问题等。

为了方便全球开发者和用户使用，开发文档和用户文档，可以预先规划好英文和中文 2 个版本。



### API接口文档规范

理由

1. 接口文档又称为 API 文档，一般由后台开发人员编写，用来描述组件提供的 API 接口，以及如何调用这些 API 接口。
2. 在项目初期，接口文档可以解耦前后端，让前后端并行开发：前端只需要按照接口文档实现调用逻辑，后端只需要按照接口文档提供功能。
3. 在项目后期，接口文档可以提供给使用者，不仅可以降低组件的使用门槛，还能够减少沟通成本。

接口文档有四种编写方式，包括编写 **Word 格式文档**、**借助工具编写**、**通过注释生成**和**编写Markdown格式**文档。

通过**注释生成**和编写**Markdown格式**文档这2种方式用得最多

接口文档拆分为以下几个 Markdown 文件

1. README.md ：API 接口介绍文档，会分类介绍 IAM 支持的 API 接口，并会存放相关 API 接口文档的链接，方便开发者查看。
2. CHANGELOG.md ：API 接口文档变更历史，方便进行历史回溯，也可以使调用者决定是否进行功能更新和版本更新。
3. generic.md ：用来说明通用的请求参数、返回参数、认证方法和请求方法等。
4. struct.md ：用来列出接口文档中使用的数据结构。这些数据结构可能被多个 API 接口使用，会在 user.md、secret.md、policy.md 文件中被引用。
5. user.md 、 secret.md 、 policy.md ：**API接口文档**，相同 REST 资源的接口会存放在一个文件中，以 REST 资源名命名文档名。
6. error\_code.md ：错误码描述，通过程序自动生成。

API接口文档例子

<https://github.com/marmotedu/iam/tree/v1.0.0/docs/guide/zh-CN/api>

接口文档

1. 接口描述：描述接口实现了什么功能。
2. 请求方法：接口的请求方法，格式为 HTTP 方法 请求路径，例如 POST /v1/users。在 通用说明中的请求方法部分，会说明接口的请求协议和请求地址。
3. 输入参数：接口的输入字段，它又分为 Header 参数、Query 参数、Body 参数、Path 参数。每个字段通过：参数名称、必选、类型 和 描述 4 个属性来描述。如果参数有限制或者默认值，可以在描述部分注明。
4. 输出参数：接口的返回字段，每个字段通过 参数名称、类型 和 描述 3 个属性来描述。
5. 请求示例：一个真实的 API 接口请求和返回示例。

## 版本规范

**语义化版本规范（SemVer，Semantic Versioning）**是 GitHub 起草的一个具有指导意义的、统一的版本号表示规范。它规定了版本号的表示、增加和比较方式，以及不同版本号代表的含义。

<https://semver.org/lang/zh-CN/>

语义化版本格式为：**主版本号.次版本号.修订号（X.Y.Z）**，其中 X、Y 和 Z 为非负的整数，且禁止在数字前方补零。

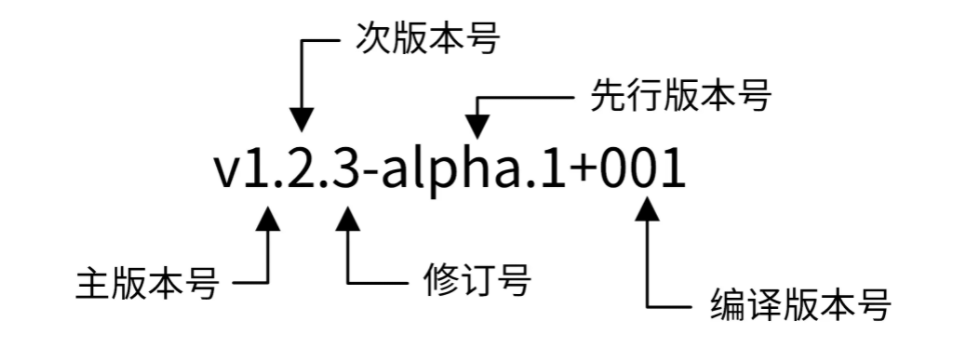
版本号可按以下规则递增：

1. 主版本号（MAJOR）：当做了不兼容的 API 修改。
2. 次版本号（MINOR）：当做了向下兼容的功能性新增及修改。这里有个不成文的约定需要你注意，偶数为稳定版本，奇数为开发版本。
3. 修订号（PATCH）：当做了向下兼容的问题修正。

例如，v1.2.3 是一个语义化版本号

先行版本号和版本编译元数据

作为延伸加到了主版本号.次版本号.修订号的后面，格式为 **X.Y.Z[-先行版本号][+版本编译元数据]**



1. **先行版本号**意味着，该版本不稳定，可能存在兼容性问题，格式为：X.Y.Z-[一连串以句点分隔的标识符]。
2. **编译版本号**，一般是编译器在编译过程中自动生成的，我们只定义其格式，并不进行人为控制。
3. 先行版本号和编译版本号只能是字母、数字，且不可以有空格。

例如：

1.0.0-alpha

1.0.0-alpha.1

1.0.0-0.3.7

1.0.0-x.7.z.92

1.0.0-alpha+001

1.0.0+20130313144700

1.0.0-beta+exp.sha.5114f85

如何确定版本号

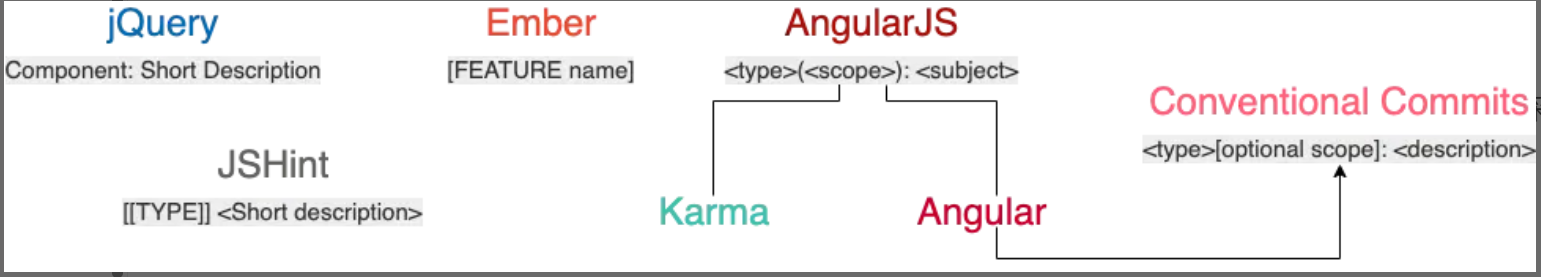
1. 第一，在实际开发的时候，我建议你使用 0.1.0 作为第一个开发版本号，并在后续的每次发行时递增次版本号。
2. 第二，当我们的版本是一个稳定的版本，并且第一次对外发布时，版本号可以定为 1.0.0。
3. 第三，当我们严格按照 Angular commit message 规范提交代码时，版本号可以这么来确定：
4. fix 类型的 commit 可以将修订号 +1。
5. feat 类型的 commit 可以将次版本号 +1。
6. 带有 BREAKING CHANGE 的 commit 可以将主版本号 +1。

## Commit规范

要求

1. 知道每个commit的变更内容
2. 基于commit能够进行过滤查找，比如只查找某个版本新增的功能：git log --oneline --grep "^feat|^fix|^perf"。
3. 基于commit能够生成change log
4. 基于commit能够触发构建或者发布流程，CI/CD
5. 基于commit确定语义化版本号

社区有多种 Commit Message 的规范，例如 jQuery、Angular 等



在这些规范中，Angular 规范在功能上能够满足开发者 commit 需求，在格式上清晰易读，目前也是用得最多的

Angular语义化的提交规范

1. Commit Message 是语义化的：Commit Message 都会被归为一个有意义的类型，用来说明本次 commit 的类型。
2. Commit Message 是规范化的：Commit Message 遵循预先定义好的规范，比如 Commit Message 格式固定、都属于某个类型，这些规范不仅可被开发者识别也可以被工具识别。

在 Angular 规范中，Commit Message 包含三个部分，分别是 Header、Body 和 Footer，格式如下

<type>[optional scope]: <description>

// 空行

[optional body]

// 空行

[optional footer(s)]

Header 是必需的，Body 和 Footer 可以省略

### Header

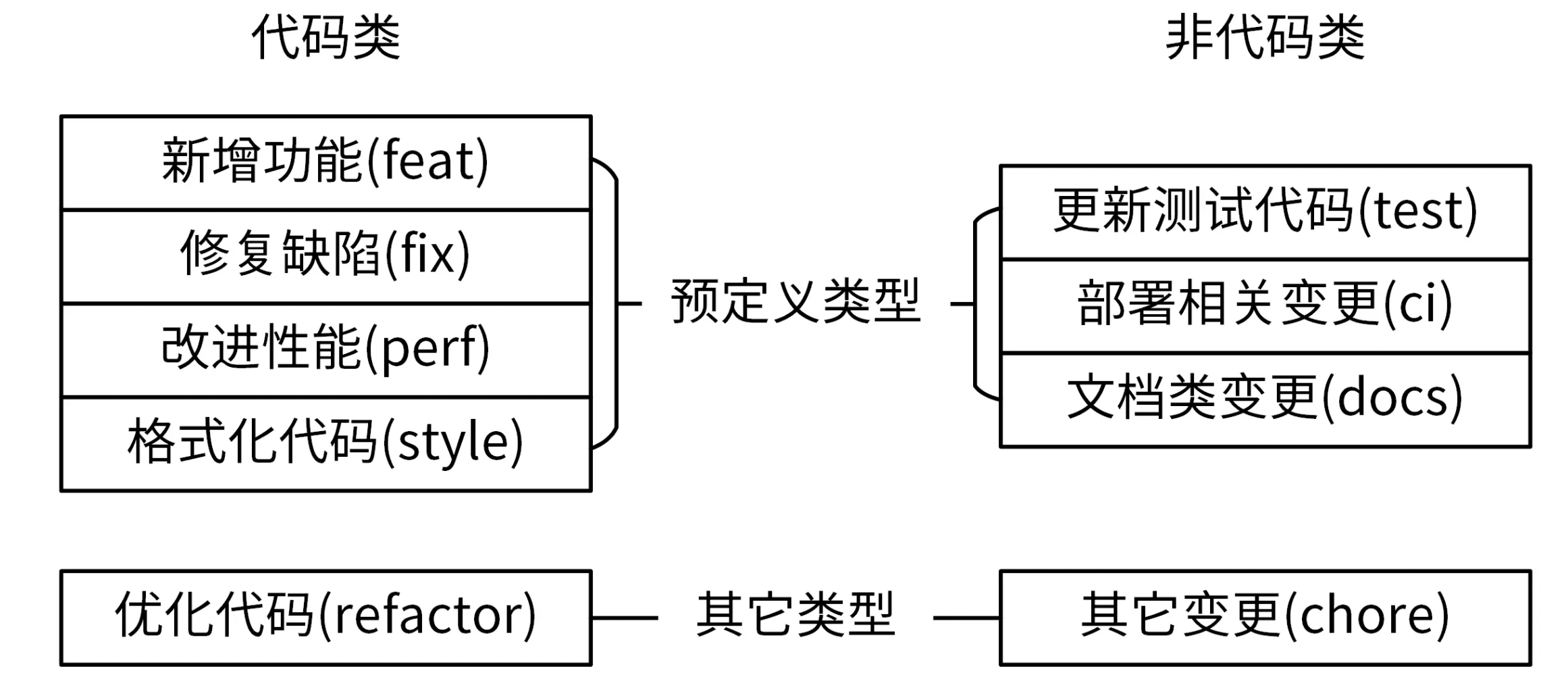
Header 部分只有一行，包括三个字段：type（必选）、scope（可选）和 subject（必选）。

#### Type

1. 主要可以归为 Development 和 Production 共两类
2. Production需要慎重



确定commit的类型



#### Scope

1. scope 是用来说明 commit 的影响范围的，它必须是名词。
2. 这里强调的是，scope 不适合设置太具体的值，提供大类上的划分。
3. 比如根据组件名和功能来设置，一个项目的scope可以enum在项目文档规范中

#### subject

1. subject 是 commit 的简短描述，必须以动词开头、使用现在时。
2. 我们可以明确地知道 commit 所执行的操作。此外我们还要注意，subject 的结尾不能加英文句号。

### Body

1. 对本次 commit 的更详细描述，是可选的。
2. 和 Header 里的一样，它也要以动词开头，使用现在时。此外，它还必须要包括修改的动机，以及和跟上一版本相比的改动点。

### Footer

1. Footer 部分不是必选的，可以根据需要来选择，主要用来说明本次 commit 导致的后果。
2. 在实际应用中，**Footer 通常用来说明不兼容的改动和关闭的 Issue** 列表

BREAKING CHANGE: <breaking change summary>

// 空行

<breaking change description + migration instructions>

// 空行

// 空行

Fixes #<issue number>

1. 不兼容的改动：如果当前代码跟上一个版本不兼容，需要在 Footer 部分，以 BREAKING CHANG: 开头，后面跟上不兼容改动的摘要。Footer 的其他部分需要说明变动的描述、变动的理由和迁移方法
2. 关闭的 Issue 列表：关闭的 Bug 需要在 Footer 部分新建一行，并以 Closes 开头列出，例如：Closes #123。如果关闭了多个 Issue，可以这样列出：Closes #123, #432, #886。

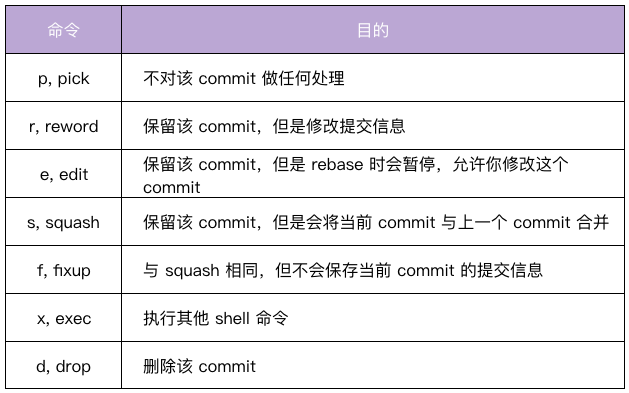
### Revert Commit

一种特殊情况，当前commit是还原了先前的commit，那么以revert:开头，后面跟还原的commit的Header，Body中写This reverts commit <hash>，其中hash是要还原的commit的SHA标识。

**为了更好地遵循 Angular 规范，建议你在提交代码时养成不用 git commit -m，即不用 -m 选项的习惯，而是直接用 git commit 或者 git commit -a 进入交互界面编辑 Commit Message。这样可以更好地格式化 Commit Message。**

### commit相关的3个重要内容

1. 提交频率
   1. 不能太高，不专业且难以理解
   2. 一种提交依据：对项目进行修改，通过测试后提交。比如修复一个bug、开发完一个小功能、开发完一个完整的功能，测试通过后就提交
   3. 另一种提交依据：规定一个代码提交周期，建议代码下班前提交一次，确保本地未提交代码延期不超过1天
2. 合并commit
   1. 可以等开发完完整的功能之后，放在一个commit中提交，那么可以在最后合并代码或者提交Pull Request前，执行git rebase -i合并之前的所有commit
   2. 将多个commit合并为一个commit提交，建议把新的commit合并到主干时，只保留2~3个commit记录
   3. **git rebase**命令
      1. 命令的作用在于重写历史
      2. git rebase -i <commit ID> 其中-i参数表示通过交互的方法，也就是vim编辑器，在界面中可以对里面的commit做一些操作
      3. git rebase支持的变更操作如下：

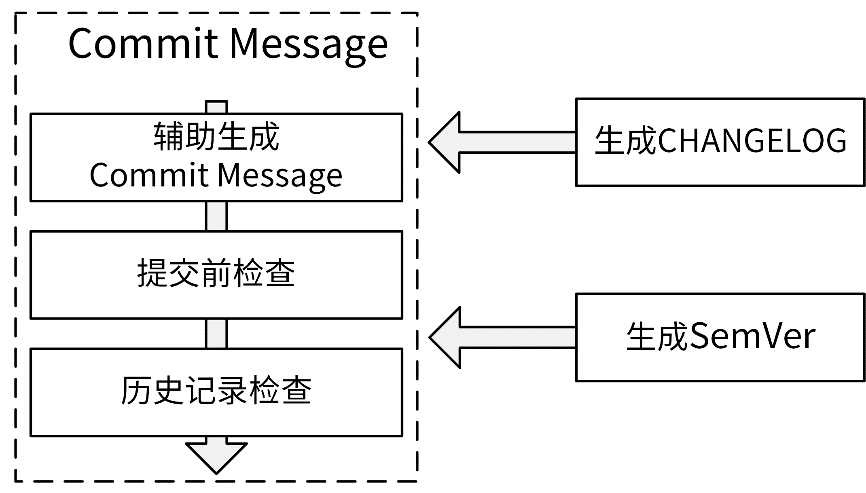


* + 1. squash指令，合并commit的时候，也合并commit的文字
    2. fixup指令，合并commit，但是会将f后面的commit的文字注释掉
    3. 删除某个commit，该commit的代码修改也会丢失
    4. 可以对commit进行排序，git会从上到下进行合并

1. **项目开发git流程示例**
   1. 首先，我们新建一个分支。我们需要先基于 master 分支新建并切换到 feature 分支（git checkout -b feature/user）
   2. 查看当前所有commit（git checkout -b feature/user），越上面越新
   3. 修改代码后，发现修改代码的commit太多，将这个功能开发的所有commit合并，那么找到当前功能commit之前一条commit，执行（git rebase -i <last commit id>），此时rebase的是当前开发的所有的commit（越下面越新）
   4. 修改指令为squash，然后，修改合并后的最终commit message信息
   5. 检查合并情况（git log –oneline）
   6. 将分支合并到主干分支（git checkout master，git merge feature/user，git log –oneline）
2. commit message修改
   1. git commit --amend，修改最近一次commit的message
   2. git rebase -i，修改某次commit的message
      1. git log --oneline
      2. git rebase -i <父commit ID>命令
      3. 使用r，reword命令
      4. git log --oneline
      5. 注意，commit message变化，那么commit id一定会变；--amend只变化最近一次，rebase会修改<父commit ID>之后所有的ID
      6. 如果当前分支有未 commit 的代码，需要先执行 git stash 将工作状态进行暂存，当修改完成后再执行 git stash pop 恢复之前的工作状态。

### commit message规范自动化

规范如果靠文档去约束，那么对于开发者的素养要求太高。



如下5个自动工具

1. commitizen-go，提供交互模式，根据提示生成commit message，再提交
2. commit-msg：githooks，再commit-msg中制定检查规则，是一个脚本，根据需要编写，或者调用其他工具比如go-gitlink进行检查
3. go-gitlint：检查历史提交的commit message是否符合angular规范，可以将工具添加再CI流程中，确保commit message都是符合规范的
4. gsemver：语义化版本自动生成工具
5. git-chglog：根据commit message生成changelog