开发规范

- 命名规范
- UI 规范
- 项目结构规范
- workflow 规范
- git commit 规范

命名规范

> 代码的注释不是越详细越好。实际上好的代码本身就是注释，我们要尽量规范和美化自己的代码来减少不必要的注释。若编程语言足够有表达力，就不需要注释，尽量通过代码来阐述。——《Clean Code》

经典代码规范

- JS 代码规范
  - airbnb-中文版
  - standard (24.5k star) 中文版
  - 百度前端编码规范 3.9k
- CSS 代码规范
  - styleguide 2.3k
  - spec 3.9k

命名神器

Codelf ，该根据是从 Github、Bitbucket、Google 代码、Codeplex、Sourceforge、Fedora、GitLab 中搜索项目，查找实际使用变量名称，然后给你推荐返回结果。

并且还提供了 VS Code 插件，可以在写代码的时候直接右键选择“Codelf”就可以跳转到网页搜索。

项目结构规范

```JavaScript
├── build                        // 构建相关
├── config                       // 配置相关
├── src                          // 源代码
│   ├── api                      // 接口
│   ├── assets                   // 静态资源
│   ├── components               // 组件
│   ├── cnfig                    // axios的
│   ├── js                       // 公用js
│   ├── router                   // 路由
│   ├── store                    // vuex
│   │   ├ modules                // 状态库模
│   │   ├ actions                // 公用异步
│   │   ├ getters                // 获取公用
│   │   ├ index                  // 总和vue
│   │   ├ mutations              // 公用同步
│   │   ├ states                 // 公用状态
│   ├── styles                   // 公用样式
│   ├── router                   // 路由vie
│   ├── views                    // 页面
│   └── App.vue                  // 入口文件
├── package.json                 // packag
└── README.md                    // 文档
```

工作流程 workflow

WorkFlow 的字面意思，工作流，即工作流程。在分支篇里，有说过这样的话：因为有分支的存在，才构成了多工作流的特色。事实的确如此，因为项目开发中，多人协作，分支很多，虽然各自在分支上互不干扰，但是我们总归需要把分支合并到一起，而且真实项目中涉及到很多问题，例如版本迭代，版本发布，bug 修复等，为了更好的管理代码，需要制定一个工作流程，这就是我们说的工作流，也有人叫它分支管理策略。

使用度最高的工作流前三名

目前使用度最高的工作流前三名（排名不分先后哈）分别是以下三种：

- Git Flow
- GitHub Flow
- GitLab Flow

Git Flow

这个工作流，是 Vincent Driessen 2010 年发布出来的他自己的分支管理模型，到现在为止，使用度非常高，我自己管理项目用的就是这个，也可以说是比较熟悉的啦。

Git Flow 的分支结构很特别，按功能来说，可以分支为 5 种分支，从 5 种分支的生命时间上，又可以分别归类为长期分支和暂时分支，或者更贴切描述为，主要分支和协助分支。

主要分支：
在采用 Git Flow 工作流的项目中，代码的中央仓库会一直存在以下两个长期分支：

master
develop
其中 origin/master 分支上的最新代码永远是版本发布状态。origin / develop 分支则是最新的开发进度。

当 develop 上的代码达到一个稳定的状态，可以发布版本的时候，develop 上这些修改会以某种特别方式被合并到 master 分支上，然后标记上对应的版本标签。

协助分支：
除了主要分支，Git Flow 的开发模式还需要一系列的协助分支，来帮助更好的功能的并行开发，简化功能开发和问题修复。是的，就是下面的三类分支。这类分支是暂时分支非常无私奉献，在需要它们的时候，迫切地创建，用完它们的时候，又挥挥衣袖地彻底消失。
协助分支分为以下几类：

Feature Branch
Release Branch
Hotfix Branch
Feature 分支用来做分模块功能开发，命名看开发者喜好，不要和其他类型的分支命名弄混淆就好，举个坏例子，命名为 master 就是一个非常不妥当的举动。模块完成之后，会合并到 develop 分支，然后删除自己。

Release 分支用来做版本发布的预发布分支，建议命名为 release-xxx。例如在软件 1.0.0 版本的功能全部开发完成，提交测试之后，从 develop 检出 release-1.0.0 ,测试中出现的小问题，在 release 分支进行修改提交，测试完毕准备发布的时候，代码会合并到 master 和 develop，master 分支合并后会打上对应版本标签 v1.0.0, 合并后删除自己，这样做的好处是，在测试的时候，不影响下一个版本功能并行开发。

Hotfix 分支是用来做线上的紧急 bug 修复的分支,建议命名为 hotfix-xxx。当线上某个版本出现了问题，将检出对应版本的代码，创建 Hotfix 分支，问题修复后，合并回 master 和 develop ，然后删除自己。这里注意，合并到 master 的时候，也要打上修复后的版本标签。

Merge 加上 no-ff 参数
需要说明的是，Git Flow 的作者 Vincent Driessen 非常建议，合并分支的时候，加上 no-ff 参数，这个参数的意思是不要选择 Fast-Forward 合并方式，而是策略合并，策略合并会让我们多一个合并提交。这样做的好处是保证一个非常清晰的提交历史，可以看到被合并分支的存在。

下面是对比图，左侧是加上参数的，后者是普通的提交：

Git Flow 示意图
下面这张图，我在刚学习 Git 的时候，看到很多次这个图，然并卵，一直都没看懂过，也不知道这张图来自 Git Flow ，只能说，我当初学 Git 的方式的确不怎么认真和系统。好在，我现在已经能看明白了这个图，并且还写了个博客，不得不感叹，时光真是好神奇，让人都遇到不一样的自己。

图中画了 Git Flow 的五种分支，master，develop，feature branchs ,release branchs , hoxfixes，其中 master 和 develop 字体被加粗代表主要分支。master 分支每合并一个分支，无论是 hotfix 还是 release ,都会打一个版本标签。通过箭头可以清楚的看到分支的开始和结束走向，例如 feature 分支从 develop 开始，最终合并回 develop ，hoxfixes 从 master 检出创建，最后合并回 develop 和 master，master 也打上了标签。

看懂之后，觉着这个图画的还挺好看的，嗯，配色也不错。

GitHub Flow

GitHub Flow 是大型程序员交友社区 GitHub 制定并使用的工作流模型，由 scott chacon 在 2011 年 8 月 31 号正式发布。

文章中说，因为 Git Flow 对于大部分开发人员和团队来说，稍微有些复杂，而且没有 GUI 图形页面，只能命令行操作，所以为了更好的解决这些问题，GitHub Flow 应运而生了。

GitHub Flow 示意图
对比上面那张 Git flow 分支模型图，真的可以称得上简单明了啦，因为 GitHub Flow 推荐做法是只有一个主分支 master，团队成员们的分支代码通过 pull Request 来合并到 master 上。

GitHub Flow 模型简单说明
只有一个长期分支 master ,而且 master 分支上的代码，永远是可发布状态,一般 master 会设置 protected 分支保护，只有有权限的人才能推送代码到 master 分支。
如果有新功能开发，可以从 master 分支上检出新分支。
在本地分支提交代码，并且保证按时向远程仓库推送。
当你需要反馈或者帮助，或者你想合并分支时，可以发起一个 pull request。
当 review 或者讨论通过后，代码会合并到目标分支。
一旦合并到 master 分支，应该立即发布。
特色之 Pull Request
在我看来，GitHub Flow 最大的特色就是 Pull Request 的提出，这是一个伟大的发明，它的用处并不仅仅是合并分支，还有以下功能：

可以很好控制分支合并权限。

分支不是你想合并就合并，需要对方同意呐

问题讨论 或者 寻求其他小伙伴们的帮助。

和拉个讨论组差不多，可以选择相关的人参与，而且参与的人还可以向你的分支提交代码，可以说，是非常适合代码交流了。

代码 Review 。

如果代码写的很烂，有了 pull request 提供的评论功能支持，准备好接受来自 review 的实时吐槽吧。当然你如果写的很棒，肯定也能被双击 666 的。

特色之 issue tracking 问题追踪
日常开发中，会用到很多第三方库，然后使用过程中，出现了问题，是不是第一个反应是去这个第三方库的 GitHub 仓库去搜索一下 issue ，看没有人遇到过，项目维护者修复了没有，一般未解决的 issue 是 open 状态，已解决的会被标记为 closed。这就是 issue tracking。

如果你是一个项目维护者，除了标记 issue 的开启和关闭，还可以给它标记上不同的标签，来优化项目。当提交的时候，如果提交信息中有 fix #1 等字段，可以自动关闭对应编号的 issue。

issue tracking 真的是非常适合开源项目。

如果你想体验 GitHub Flow
GitHub 社区使用的就是这个工作流模型，而且帮助文档非常详细，可以建个项目，多耍耍。

GitLab Flow

这个工作流十分地年轻，是 GitLab 的 CEO Sytse Sijbrandij 在 2014 年 9 月 29 正式发布出来的。因为出现的比前面两种工作流稍微晚一些，所以它有个非常大的优势，集百家之长，补百家之短。

GitLab 既支持 Git Flow 的分支策略，也有 GitHub Flow 的 Pull Request（ Merge Request ） 和 issue tracking。

Git Flow & GitHub Flow 的瑕疵
当 Git Flow 出现后，它解决了之前项目管理的很让人头疼的分支管理，但是实际使用过程中，也暴露了很多问题：

默认工作分支是 develop，但是大部分版本管理工具默认分支都是 master，开始的时候总是需要切换很麻烦。
Hotfix 和 Release 分支在需要版本快速迭代的项目中，几乎用不到，因为刚开发完就直接合并到 master 发版，出现问题 develop 就直接修复发布下个版本了。
Hotfix 和 Release 分支，一个从 master 创建，一个从 develop 创建，使用完毕，需要合并回 develop 和 master。而且在实际项目管理中，很多开发者会忘记合并回 develop 或者 master。
GitHub Flow 的出现，非常大程度上简化了 Git Flow ，因为只有一个长期分支 master，并且提供 GUI 操作工具，一定程度上避免了上述的几个问题，然而在一些实际问题面前，仅仅使用 master 分支显然有点力不从心，例如：

版本的延迟发布（例如 iOS 应用审核到通过中间，可能也要在 master 上推送代码）
不同环境的部署 （例如：测试环境，预发环境，正式环境）
不同版本发布与修复 （是的，只有一个 master 分支真的不够用）
GitLab Flow 解决方案
为了解决上面那些毛茸茸的小问题，GitLab Flow 给出了以下的解决方法。

版本的延迟发布—Prodution Branch
master 分支不够，于是添加了一个 prodution 分支，专门用来发布版本。

不同环境的部署—Environment Branches & Upstream First
每个环境，都对应一个分支，例如下图中的 pre-production 和 prodution 分支都对应不同的环境，我觉得这个工作流模型比较适用服务端，测试环境，预发环境，正式环境，一个环境建一个分支。

这里要注意，代码合并的顺序，要按环境依次推送，确保代码被充分测试过,才会从上游分支合并到下游分支。除非是很紧急的情况，才允许跳过上游分支，直接合并到下游分支。这个被定义为一个规则，名字叫“upstream first”，翻译过来是“上游优先”。

版本发布分支—Release Branches & Upstream First
只有当对外发布软件的时候，才需要创建 release 分支。作为一个移动端开发来说，对外发布版本的记录是非常重要的，如果线上出现了一个问题，需要拿到问题出现对应版本的代码，才能准确定位问题。

在 Git Flow ,版本记录是通过 master 上的 tag 来记录。发现问题，创建 hotfix 分支，完成之后合并到 master 和 develop。

在 GitLab Flow ，建议的做法是每一个稳定版本，都要从 master 分支拉出一个分支，比如 2-3-stable、2-4-stable 等等。发现问题，就从对应版本分支创建修复分支，完成之后，先合并到 master，才能再合并到 release 分支，遵循“上游优先”原则。

git commit 规范

在我们使用 Git 提交代码的时候，会要求写一个 commit 信息，简述本次提交的代码的目的。试想一下，现在线上出问题了，需要排查问题，你打开 gtihub\gitlab，想要找到提交代码的记录，进行回滚操作。但是你发现 commit 的内容都是“。”、“提交”、“修改”这样的信息，时间紧迫，你需要用最快的时间恢复故障，面对这样的 commit 信息你会不会直接口吐芬芳呢？就算是你自己写的代码你也不能第一时间就知道问题出现在哪个提交记录里面，于是只好一次提交记录一次提交记录的点开看里面的改动内容。点开看完了之后回到列表，咦？我刚刚看的是哪一个呀？∧(;´Д｀∧) 如果随便提交 commit 信息，会直接提升我们修复 bug、查找问题的难度，写好 commit message 还是很有必要的。

commit message 作用

那么 commit message 有哪些作用呢？commit message 可以提供更多的历史信息，方便快速浏览，便于快速查找信息，高效合作。良好的 git commit 规范，可以让人只看描述就清楚这次提交的目的，可以提高解决 bug、查找问题的效率。所以作为开发者，我们有责任写好每一次的 commit message。

在一个团队中，团队中的每一个成员都共同维护一个或者多个产品，如果团队之间遵守同一套 commit message 规范，可以过滤某些 commit（比如文档改动），还可以使用一些脚本文件自动生成变更历史 CHANGELOG，这就会提高项目\产品的专业性，让其他开发者或者能够看到完整的产品功能迭代历史和发展线。所以，团队之间需要遵守同一套 commit message 规范。

commit message 的格式

说了那么多统一 commit message 的好处，那么如何来写 commit message 呢?

目前使用最多的是 Angular 规范，那么我们就来大致介绍下。一个 commit message 中包括 Header、Body、Footer 三个部分，并且一次可以提交多行信息。格式如下：

```
<type>(<scope>): <subject>

// 空一行

<body>

// 空一行

<footer>
```

其中，第一行`<type>(<scope>): <subject>`为 Header 部分，是必填字段，用于概括说明本次改动的内容，包括改动的类别(type)、影响范围(scope)、和简短的描述。后面两行分别为 Body 和 Footer，都是是选填字段，可以写一些对本次改动的详细描述和一些其他功能（比如关闭 Issue）。

Header

刚刚也说到了，在 Header 中又包括三个字段：type、scope、subject。

type

type 是用于说明 commit 的类别，只允许使用下面 7 个标识。

- feat：本次改动为新增功能；
- fix：本次改动为修补 bug；
- docs：本次改动为新增或修改文档（documentation）信息；
- style：本地改动为修改样式文件；
- refactor：本次改动为代码重构；
- test：本次改动为新增或修改测试用例；
- chore：构建过程或辅助工具的变动；

scope

scope 用于说明 commit 影响的范围，比如数据层、控制层、视图层或者会影响到那个功能或者改动的文件名称，视项目不同而不同。

subject

subject 是 commit 目的的简短描述，不超过 50 个字符。以动词开头，使用第一人称现在时，比如 change，而不是 changed 或 changes。第一个字母小写、结尾不加句号。

body

Body 部分是对本次 commit 的详细描述，可以分成多行。下面是一个范例。

> 产品新增功能：xxx
>
> PRD 链接：xxx
>
> 主要改动点如下：
>
> 1. 新增 xxx
> 2. 修改 xxx
> 3. 删除 xxx

Footer

Footer 部分只用于两种情况

- 不兼容变动：如果当前代码与上一个版本不兼容，则 Footer 部分以 BREAKING CHANGE 开头，后面是对变动的描述、以及变动理由和迁移方法。
- 关闭 Issue：如果当前 commit 针对某个 issue，那么可以在 Footer 部分关闭这个 issue

  > Closes #123, #456, #789

自动化

如果想更近一步，还可以用自动化的手段，方便生成、效验你的 commit message，也可以针对符合格式的 commit messag 使用脚本自动生成变更记录 CHANGELOG 文件。