git 规范

git 规范一般包括两点：分支管理规范和 git commit 规范。

#分支管理规范

一个项目可以创建两个分支：master 和 dev。master 对应线上分支，不能直接在 master 分支上写代码，开发时需要从 master 上拉一个 dev 分支进行开发。

#开发新功能

当团队成员开发新功能时，需要从 dev 上拉一个 `feature-功能名称-开发姓名` 分支进行开发，例如：`feature-login-tgz`。开发完成后需要合并回 dev 分支。

#修改 bug

当团队成员修改 bug 时，需要从有 bug 的分支（环境）上拉一个 `bug-功能名称-开发姓名` 分支进行修复，例如：`bug-login-tgz`。修复完成后需要合并回原来出现 bug 的分支。

以 `feature` 或 `bug` 开始的分支都属于临时分支，在通过测试并上线后需要将临时分支进行删除。避免 git 上出现太多无用的分支。

#合并分支

在将一个分支合并到另一个分支时（例如将 `feature-*` 合并到 dev），需要查看自己的新分支中有没有多个重复提交或意义不明的 commit。如果有，则需要对它们进行合并（git rebase）。示例：

```
# 这两个 commit 可以合并成一个
chore: 修改按钮文字
chore: 修改按钮样式

# 合并后
chore: 修改按钮样式及文字
```

注意：在将 `feature-*` 合并到 dev 时，需要先将 dev 分支合并到 `feature-*` 分支，然后再将 `feature-*` 合并到 dev 分支，避免出现代码冲突的情况。同理，合并 `bug-*` 分支也一样。

#部署

当 dev 分支通过测试后，就可以合并到 master 进行发布了。

#发布时可能出现的意外情况

举个例子，假设程序要新增 a、b 两个功能，我们的操作流程是这样的：

1. 从 dev 分支拉两个新分支 `feature-a-tgz`、`feature-b-tgz`。
2. 开发完成合并回 dev。
3. dev 测试完毕后，合并到 master 进行发布。

如果这时突然被告知 b 功能不上，只上 a 功能。我们可以将 `feature-a-tgz` 分支重新部署到测试环境，这样就不用做任何的代码回滚。只要 `feature-a-tgz` 分支测试通过就可以直接合到 master 进行线上发布。

#git commit 规范

git 在每次提交时，都需要填写 commit message。

```
git commit -m 'this is a test'
```

commit message 就是对你这次的代码提交进行一个简单的说明，好的提交说明可以让人一眼就明白这次代码提交做了什么。

既然明白了 commit message 的重要性，那我们就更要好好的学习一下 commit message 规范。下面让我们看一下 commit message 的格式：

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

我们可以发现，commit message 分为三个部分(使用空行分割):

1. 标题行（subject）：必填, 描述主要修改类型和内容。
2. 主题内容（body）：描述为什么修改, 做了什么样的修改, 以及开发的思路等等。
3. 页脚注释（footer）：可以写注释，放 BUG 号的链接。

#type

commit 的类型：

- feat: 新功能、新特性
- fix: 修改 bug
- perf: 更改代码，以提高性能（在不影响代码内部行为的前提下，对程序性能进行优化）
- refactor: 代码重构（重构，在不影响代码内部行为、功能下的代码修改）
- docs: 文档修改
- style: 代码格式修改, 注意不是 css 修改（例如分号修改）
- test: 测试用例新增、修改
- build: 影响项目构建或依赖项修改
- revert: 恢复上一次提交
- ci: 持续集成相关文件修改
- chore: 其他修改（不在上述类型中的修改）
- release: 发布新版本

#scope

commit message 影响的功能或文件范围, 比如: route, component, utils, build...

#subject

commit message 的概述

#body

具体修改内容, 可以分为多行.

#footer

一些备注, 通常是 BREAKING CHANGE 或修复的 bug 的链接.

#约定式提交规范

以下内容来源于：https://www.conventionalcommits.org/zh-hans/v1.0.0-beta.4/

- 每个提交都必须使用类型字段前缀，它由一个名词组成，诸如 `feat` 或 `fix` ，其后接一个可选的作用域字段，以及一个必要的冒号（英文半角）和空格。
- 当一个提交为应用或类库实现了新特性时，必须使用 `feat` 类型。
- 当一个提交为应用修复了 `bug` 时，必须使用 `fix` 类型。
- 作用域字段可以跟随在类型字段后面。作用域必须是一个描述某部分代码的名词，并用圆括号包围，例如： `fix(parser):`
- 描述字段必须紧接在类型/作用域前缀的空格之后。描述指的是对代码变更的简短总结，例如：`fix: array parsing issue when multiple spaces were contained in string.`
- 在简短描述之后，可以编写更长的提交正文，为代码变更提供额外的上下文信息。正文必须起始于描述字段结束的一个空行后。
- 在正文结束的一个空行之后，可以编写一行或多行脚注。脚注必须包含关于提交的元信息，例如：关联的合并请求、Reviewer、破坏性变更，每条元信息一行。
- 破坏性变更必须标示在正文区域最开始处，或脚注区域中某一行的开始。一个破坏性变更必须包含大写的文本 `BREAKING CHANGE`，后面紧跟冒号和空格。
- 在 `BREAKING CHANGE:` 之后必须提供描述，以描述对 API 的变更。例如： `BREAKING CHANGE: environment variables now take precedence over config files.`
- 在提交说明中，可以使用 `feat` 和 `fix` 之外的类型。
- 工具的实现必须不区分大小写地解析构成约定式提交的信息单元，只有 `BREAKING CHANGE` 必须是大写的。
- 可以在类型/作用域前缀之后，: 之前，附加 `!` 字符，以进一步提醒注意破坏性变更。当有 `!` 前缀时，正文或脚注内必须包含 `BREAKING CHANGE: description`

#示例

#fix（修复BUG）

每次 git commit 最好加上范围描述。

例如这次 BUG 修复影响到全局，可以加个 global。如果影响的是某个目录或某个功能，可以加上该目录的路径，或者对应的功能名称。

```
// 示例1
fix(global):修复checkbox不能复选的问题
// 示例2 下面圆括号里的 common 为通用管理的名称
fix(common): 修复字体过小的BUG，将通用管理下所有页面的默认字体大小修改为 14px
// 示例3
fix(test): value.length -> values.length
```

#feat（添加新功能或新页面）

```
feat: 添加网站主页静态页面

这是一个示例，假设对任务静态页面进行了一些描述。

这里是备注，可以是放 BUG 链接或者一些重要性的东西。
```

#chore（其他修改）

chore 的中文翻译为日常事务、例行工作。顾名思义，即不在其他 commit 类型中的修改，都可以用 chore 表示。

```
chore: 将表格中的查看详情改为详情
```

其他类型的 commit 和上面三个示例差不多，在此不再赘述。