约定式提交

一种用于给提交信息增加人机可读含义的规范

阅读规范 GitHub

约定式提交 1.0.0

概述

约定式提交规范是一种基于提交信息的轻量级约定。 它提供了一组简单规则来创建清晰的提交历史； 这更有利于编写自动化工具。 通过在提交信息中描述功能、修复和破坏性变更， 使这种惯例与 **SemVer** 相互对应。

提交说明的结构如下所示：

原文：

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

译文：

```
<类型>[可选 范围]: <描述>

[可选 正文]

[可选 脚注]
```

提交说明包含了下面的结构化元素，以向类库使用者表明其意图：

1. **fix:** *类型* 为 `fix` 的提交表示在代码库中修复了一个 bug（这和语义化版本中的 `PATCH` 相对应）。
2. **feat:** *类型* 为 `feat` 的提交表示在代码库中新增了一个功能（这和语义化版本中的 `MINOR` 相对应）。
3. **BREAKING CHANGE:** 在脚注中包含 `BREAKING CHANGE:` 或 <类型>(范围) 后面有一个 `!` 的提交，表示引入了破坏性 API 变更（这和语义化版本中的 `MAJOR` 相对应）。 破坏性变更可以是任意 *类型* 提交的一部分。
4. 除 `fix:` 和 `feat:` 之外，也可以使用其它提交 *类型* ，例如 **@commitlint/config-conventional**（基于 **Angular 约定**）中推荐的 `build:` 、 `chore:` 、 `ci:` 、 `docs:` 、 `style:` 、 `refactor:` 、 `perf:` 、 `test:` ，等等。
   - build: 用于修改项目构建系统，例如修改依赖库、外部接口或者升级 Node 版本等；
   - chore: 用于对非业务性代码进行修改，例如修改构建流程或者工具配置等；
   - ci: 用于修改持续集成流程，例如修改 Travis、Jenkins 等工作流配置；
   - docs: 用于修改文档，例如修改 README 文件、API 文档等；
   - style: 用于修改代码的样式，例如调整缩进、空格、空行等；
   - refactor: 用于重构代码，例如修改代码结构、变量名、函数名等但不修改功能逻辑；
   - perf: 用于优化性能，例如提升代码的性能、减少内存占用等；

    - test: 用于修改测试用例，例如添加、删除、修改代码的测试用例等。
5. 脚注中除了 `BREAKING CHANGE: <description>` ，其它条目应该采用类似 **git trailer format** 这样的惯例。

其它提交类型在约定式提交规范中并没有强制限制，并且在语义化版本中没有隐式影响（除非它们包含 BREAKING CHANGE）。 可以为提交类型添加一个围在圆括号内的范围，以为其提供额外的上下文信息。例如 `feat(parser): adds ability to parse arrays.`。

示例

包含了描述并且脚注中有破坏性变更的提交说明

```
feat: allow provided config object to extend other configs

BREAKING CHANGE: `extends` key in config file is now used for extending other config files
```

包含了 `!` 字符以提醒注意破坏性变更的提交说明

```
feat!: send an email to the customer when a product is shipped
```

包含了范围和破坏性变更 `!` 的提交說明

```
feat(api)!: send an email to the customer when a product is shipped
```

包含了 `!` 和 BREAKING CHANGE 脚注的提交说明

```
chore!: drop support for Node 6

BREAKING CHANGE: use JavaScript features not available in Node 6.
```

不包含正文的提交说明

```
docs: correct spelling of CHANGELOG
```

包含范围的提交说明

```
feat(lang): add polish language
```

包含多行正文和多行脚注的提交说明

```
fix: prevent racing of requests

Introduce a request id and a reference to latest request. Dismiss
```

```
introduce a request id and a reference to latest request. Dismiss
incoming responses other than from latest request.

Remove timeouts which were used to mitigate the racing issue but are
obsolete now.

Reviewed-by: Z
Refs: #123
```

约定式提交规范

本文中的关键词 “必须（MUST）”、“禁止（MUST NOT）”、“必要（REQUIRED）”、“应当（SHALL）”、“不应当（SHALL NOT）”、“应该（SHOULD）”、“不应该（SHOULD NOT）”、“推荐（RECOMMENDED）”、“可以（MAY）” 和 “可选（OPTIONAL）”，其相关解释参考 **RFC 2119** 。

1. 每个提交都**必须**使用类型字段前缀，它由一个名词构成，诸如 `feat` 或 `fix` ， 其后接**可选的**范围字段，**可选的** `!`，以及**必要的**冒号（英文半角）和空格。
2. 当一个提交为应用或类库实现了新功能时，**必须**使用 `feat` 类型。
3. 当一个提交为应用修复了 bug 时，**必须**使用 `fix` 类型。
4. 范围字段**可以**跟随在类型字段后面。范围**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):`
5. 描述字段**必须**直接跟在 <类型>(范围) 前缀的冒号和空格之后。 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* 。
6. 在简短描述之后，**可以**编写较长的提交正文，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。
7. 提交的正文内容自由编写，并**可以**使用空行分隔不同段落。
8. 在正文结束的一个空行之后，**可以**编写一行或多行脚注。每行脚注都**必须**包含 一个令牌（token），后面紧跟 `:<space>` 或 `<space>#` 作为分隔符，后面再紧跟令牌的值（受 **git trailer convention** 启发）。
9. 脚注的令牌**必须**使用 `-` 作为连字符，比如 `Acked-by` (这样有助于 区分脚注和多行正文)。有一种例外情况就是 `BREAKING CHANGE` ，它**可以**被认为是一个令牌。
10. 脚注的值**可以**包含空格和换行，值的解析过程**必须**直到下一个脚注的令牌/分隔符出现为止。
11. 破坏性变更**必须**在提交信息中标记出来，要么在 <类型>(范围) 前缀中标记，要么作为脚注的一项。
12. 包含在脚注中时，破坏性变更**必须**包含大写的文本 `BREAKING CHANGE` ，后面紧跟着冒号、空格，然后是描述，例如： *BREAKING CHANGE: environment variables now take precedence over config files* 。
13. 包含在 <类型>(范围) 前缀时，破坏性变更**必须**通过把 `!` 直接放在 `:` 前面标记出来。 如果使用了 `!` ，那么脚注中**可以**不写 `BREAKING CHANGE:` ， 同时提交信息的描述中**应该**用来描述破坏性变更。
14. 在提交说明中，**可以**使用 `feat` 和 `fix` 之外的类型，比如：*docs: updated ref docs.* 。
15. 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。
16. BREAKING-CHANGE 作为脚注的令牌时**必须**是 BREAKING CHANGE 的同义词。

为什么使用约定式提交

- 自动化生成 CHANGELOG。
- 基于提交的类型，自动决定语义化的版本变更。
- 向同事、公众与其他利益关系者传达变化的性质。
- 触发构建和部署流程。
- 让人们探索一个更加结构化的提交历史，以便降低对你的项目做出贡献的难度。

FAQ

在初始开发阶段我该如何处理提交说明？

我们建议你按照假设你已发布了产品那样来处理。因为通常总 *有人* 使用你的软件，即便那是你软件开发的同事们。他们会希望知道诸如修复了什么、哪里不兼容等信息。

提交标题中的类型是大写还是小写？

大小写都可以，但最好是一致的。

如果提交符合多种类型我该如何操作？

回退并尽可能创建多次提交。约定式提交的好处之一是能够促使我们做出更有组织的提交和 PR。

这不会阻碍快速开发和迭代吗？

它阻碍的是以杂乱无章的方式快速前进。它助你能在横跨多个项目以及和多个贡献者协作时长期地快速演进。

约定式提交会让开发者受限于提交的类型吗（因为他们会想着已提供的类型）？

约定式提交鼓励我们更多地使用某些类型的提交，比如 `fixes`。除此之外，约定式提交的灵活性也允许你的团队使用自己的类型，并随着时间的推移更改这些类型。

这和 SemVer 有什么关联呢？

`fix` 类型提交应当对应到 `PATCH` 版本。 `feat` 类型提交应该对应到 `MINOR` 版本。带有 `BREAKING CHANGE` 的提交不管类型如何，都应该对应到 `MAJOR` 版本。

我对约定式提交做了形如 `@jameswomack/conventional-commit-spec` 的扩展，该如何版本化管理这些扩展呢？

我们推荐使用 SemVer 来发布你对于这个规范的扩展（并鼓励你创建这些扩展！）

如果我不小心使用了错误的提交类型，该怎么办呢？

当你使用了在规范中但错误的类型时，例如将 `feat` 写成了 `fix`

在合并或发布这个错误之前，我们建议使用 `git rebase -i` 来编辑提交历史。而在发布之后，根据你使用的工具和流程不同，会有不同的清理方案。

当使用了 *不在* 规范中的类型时，例如将 `feat` 写成了 `feet`

在最坏的场景下，即便提交没有满足约定式提交的规范，也不会是世界末日。这只意味着这个提交会被基于规范的工具错过而已。

所有的贡献者都需要使用约定式提交规范吗？

并不！如果你使用基于 squash 的 Git 工作流，主管维护者可以在合并时清理提交信息——这不会对普通提交者产生额外的负担。 有种常见的工作流是让 git 系统自动从 pull request 中 squash 出提交，并向主管维护者提供一份表单，用以在合并时输入合适的 git 提交信息。

约定式提交规范中如何处理还原（revert）提交？

还原提交（Reverting）会比较复杂：你还原的是多个提交吗？如果你还原了一个功能模块，下次发布的应该是补丁吗？

约定式提交不能明确的定义还原行为。所以我们把这个问题留给工具开发者， 基于 *类型* 和 *脚注* 的灵活性来开发他们自己的还原处理逻辑。

一种建议是使用 `revert` 类型，和一个指向被还原提交摘要的脚注：

```
revert: let us never again speak of the noodle incident

Refs: 676104e, a215868
```