*标准化大厂编程规范解决方案之ESLint + Git Hooks   
## 01：为什么需要编程规范？*\*\*工欲善其事，必先利其器\*\*   
对于一些大型的企业级项目而言，通常情况下我们都是需要一个团队来进行开发的。而又因为团队人员对技术理解上的参差不齐，所以就会导致出现一种情况，那就是《\*\*一个项目无法具备统一的编程规范，导致项目的代码像多个不同材质的补丁拼接起来一样\*\*》  
设想一下，下面的这段代码有一个团队进行开发，因为没有具备统一的代码标准，所以生成了下面的代码：

文本

描述已自动生成  
这段代码可以正常运行没有问题，但是整体的代码结构却非常的难看。  
  
> 有的地方有空格进行分割，有的地方却没有  
>  
> 有的地方是单引号，有的地方却是双引号  
>  
> 有的地方有分号，有的地方没有分号  
>  
> ....  
  
这样的项目虽然可以正常运行，但是如果把它放到大厂的项目中，确实 \*\*不及格\*\* 的，它会被认为是 \*\*不可维护、不可扩展的代码内容\*\*  
  
那么所谓的大厂标准的代码结构应该是什么样子的呢？  
  
我们把上面的代码进行一下修正，做一个对比：

文本

描述已自动生成

修改之后的代码具备了统一的规范之后，是不是看起来就舒服多了！  
  
并且以上所列举出来的只是《编程规范》中的一小部分内容！  
  
那么有些同学可能就会说了，你列举出来这些编程规范有什么用啊！  
  
哪怕你写上一部书，我们一个团队这么多人，总不能指望所有人都看一遍，并且严格的遵守你所说的规范吧！  
  
说的没错！指望人主动的遵守这些规范不太现实  
  
那怎么办呢？  
  
那么我们可不可以另辟蹊径，让程序自动处理规范化的内容呢？  
  
答案是：可以的！  
  
这些也是我们本章节所需要讲解的重点内容！  
  
本章节中我们会为大家讲解，如何自动化的对代码进行规范，其中主要包括：  
  
1. 编码规范  
2. git 规范  
  
两大类  
  
那么明确好了我们的范围之后，接下来就让我们创建一个项目，开始我们的代码规范之旅吧！  
*## 02：使用 vue-cli 创建项目（图文）*本章节为 \*\*图文节\*\*，请点击 [这里](./图文课程/2-02：使用 vue-cli 创建项目.md) 查看对应文档  
*## 03：升级最新的 vue 版本以支持 script setup 语法（图文）*本章节为 \*\*图文\*\*，请点击 [这里](./图文课程/2-03：升级最新的 vue 版本以支持 script setup 语法 .md) 查看对应文档  
*## 04：大厂编程规范一：代码检测工具 ESLint 你了解多少？*在我们去创建项目的时候，脚手架工具已经帮助我们安装了 `ESLint` 代码检测工具。  
对于 `ESLint` 的大名，同学们或多或少的应该都听说过，只不过有些同学可能了解的多一些，有些同学了解的少一些。  
  
那么本小节我们就先来聊一下，这个赫赫有名的代码检测工具 `ESLint`  
  
首先 `ESLint` 是 `2013年6月` 创建的一个开源项目，它的目标非常简单，只有一个，那就是 \*\*提供一个插件化的 `javascript` 代码检测工具\*\* ，说白了就是做 \*\*代码格式检测使用的\*\*  
  
在咱们当前的项目中，包含一个 `.eslintrc.js` 文件，这个文件就是 `eslint` 的配置文件。  
  
随着大家对代码格式的规范性越来越重视，`eslint` 也逐渐被更多的人所接收，同时也有很多大厂在原有的 `eslint` 规则基础之上进行了一些延伸。  
  
我们在创建项目时，就进行过这样的选择：  
  
```js  
? Pick a linter / formatter config:   
 ESLint with error prevention only // 仅包含错误的 ESLint  
 ESLint + Airbnb config // Airbnb 的 ESLint 延伸规则  
 ESLint + Standard config // 标准的 ESLint 规则  
```  
  
我们当前选择了 \*\*标准的 ESLint 规则\*\* ，那么接下来我们就在该规则之下，看一看 `ESLint` 它的一些配置都有什么？  
  
打开项目中的 `.eslintrc.js` 文件  
  
```js  
// ESLint 配置文件遵循 commonJS 的导出规则，所导出的对象就是 ESLint 的配置对象  
// 文档：https://eslint.bootcss.com/docs/user-guide/configuring  
module.exports = {  
 // 表示当前目录即为根目录，ESLint 规则将被限制到该目录下  
 root: true,  
 // env 表示启用 ESLint 检测的环境  
 env: {  
 // 在 node 环境下启动 ESLint 检测  
 node: true  
 },  
 // ESLint 中基础配置需要继承的配置  
 extends: ["plugin:vue/vue3-essential", "@vue/standard"],  
 // 解析器  
 parserOptions: {  
 parser: "babel-eslint"  
 },  
 // 需要修改的启用规则及其各自的错误级别  
 /\*\*  
 \* 错误级别分为三种：  
 \* "off" 或 0 - 关闭规则  
 \* "warn" 或 1 - 开启规则，使用警告级别的错误：warn (不会导致程序退出)  
 \* "error" 或 2 - 开启规则，使用错误级别的错误：error (当被触发的时候，程序会退出)  
 \*/  
 rules: {  
 "no-console": process.env.NODE\_ENV === "production" ? "warn" : "off",  
 "no-debugger": process.env.NODE\_ENV === "production" ? "warn" : "off"  
 }  
};  
  
```  
  
那么到这里咱们已经大致的了解了`.eslintrc.js` 文件，基于 `ESLint` 如果我们出现不符合规范的代码格式时，那么就会得到一个对应的错误。  
  
比如：  
  
> 我们可以把 `Home.vue` 中的 `name` 属性值，由单引号改为双引号  
  
此时，只要我们一保存代码，那么就会得到一个对应的错误  
文本

描述已自动生成  
  
  
这个错误表示：   
  
1. 此时我们触发了一个 《错误级别的错误》   
2. 触发该错误的位置是 在 `Home.vue` 的第 13 行 第九列 中  
3. 错误描述为：字符串必须使用单引号  
4. 错误规则为：`quotes`  
  
那么想要解决这个错误，通常情况下我们有两种方式：  
  
1. 按照 `ESLint` 的要求修改代码  
2. 修改 `ESLint` 的验证规则  
  
\*\*按照 `ESLint` 的要求修改代码：\*\*  
  
> 在 `Home.vue` 的第 13 行中把双引号改为单引号  
  
\*\*修改 `ESLint` 的验证规则：\*\*  
  
1. 在 `.eslintrc.js` 文件中，新增一条验证规则  
  
 ```json  
 "quotes": "error" // 默认  
 "quotes": "warn" // 修改为警告  
 "quotes": "off" // 修改不校验  
 ```  
  
那么这一小节，我们了解了 `vue-cli` 创建 `vue3` 项目时，`Standard config` 的 `ESLint` 配置，并且知道了如何解决 `ESLint` 报错的问题。  
  
但是一个团队中，人员的水平高低不齐，大量的 `ESLint` 规则校验，会让很多的开发者头疼不已，从而大大影响了项目的开发进度。  
  
试想一下，在你去完成项目代码的同时，还需要时时刻刻注意代码的格式问题，这将是一件多么痛苦的事情！  
  
那么有没有什么办法，既可以保证 `ESLint` 规则校验，又可以解决严苛的格式规则导致的影响项目进度的问题呢？  
  
欲知后事如何，请听下一节《`Prettier` ，让你的代码变得更漂亮！》  
  
  
  
*## 05：大厂编程规范二：你知道代码格式化 Prettier 吗？*在上一小节中，我们知道了 `ESLint` 可以让我们的代码格式变得更加规范，但是同样的它也会带来开发时编码复杂度上升的问题。  
  
那么有没有办法既可以保证 `ESLint` 规则校验，又可以让开发者无需关注格式问题来进行顺畅的开发呢？  
  
答案是：有的！  
  
而解决这个问题的关键就是 `prettier`！（点击 [这里](*https://www.prettier.cn/*) 进入 `prettier` 中文官网！）  
  
\*\*`prettier` 是什么？\*\*  
  
1. 一个代码格式化工具  
2. 开箱即用  
3. 可以直接集成到 `VSCode` 之中  
4. 在保存时，让代码直接符合 `ESLint` 标准（需要通过一些简单配置）  
  
那么这些简单配置具体指的是什么呢？  
  
请看下一小节《ESLint 与 Prettier 配合解决代码格式问题》  
  
*## 06：ESLint 与 Prettier 配合解决代码格式问题*在上一小节中，我们提到《`prettier` 可以在保存代码时，让我们的代码直接符合 `ESLint` 标准》但是想要实现这样的功能需要进行一些配置。  
  
那么这一小节，我们就来去完成这个功能：  
  
1. 在 `VSCode` 中安装 `prettier` 插件（搜索 `prettier`），这个插件可以帮助我们在配置 `prettier` 的时候获得提示  
 图形用户界面, 应用程序

描述已自动生成  
2. 在项目中新建 `.prettierrc` 文件，该文件为 `perttier` 默认配置文件  
  
3. 在该文件中写入如下配置：  
  
 ```json  
 {  
 // 不尾随分号  
 "semi": false,  
 // 使用单引号  
 "singleQuote": true,  
 // 多行逗号分割的语法中，最后一行不加逗号  
 "trailingComma": "none"  
 }  
 ```  
  
4. 打开 `VSCode` 《设置面板》

图形用户界面, 应用程序

描述已自动生成  
  
  
5. 在设置中，搜索 `save` ，勾选 `Format On Save`  
 文本

描述已自动生成  
至此，你即可在 \*\*`VSCode` 保存时，自动格式化代码！\*\*  
  
\*\*但是！\*\* 你只做到这样还不够！  
  
> 1. VSCode 而言，默认一个 tab 等于 4 个空格，而 ESLint 希望一个 tab 为两个空格  
>  
> 2. 如果大家的 VSCode 安装了多个代码格式化工具的化  
>  
> 3. ESLint 和 prettier 之间的冲突问题  
  
我们尝试在 `Home.vue` 中写入一个 `created` 方法，写入完成之后，打开我们的控制台我们会发现，此时代码抛出了一个 `ESLint` 的错误  
  
文本

描述已自动生成  
  
这个错误的意思是说：\*\*`created` 这个方法名和后面的小括号之间，应该有一个空格！\*\*  
  
但是当我们加入了这个空格之后，只要一保存代码，就会发现 `prettier` 会自动帮助我们去除掉这个空格。  
  
那么此时的这个问题就是 `prettier` 和 `ESLint` 的冲突问题。  
  
针对于这个问题我们想要解决也非常简单：  
  
1. 打开 `.eslintrc.js` 配置文件  
  
2. 在 `rules` 规则下，新增一条规则  
  
 ```json  
 'space-before-function-paren': 'off'  
 ```  
  
3. 该规则表示关闭《方法名后增加空格》的规则  
  
4. 重启项目  
  
至此我们整个的 `perttier` 和 `ESLint` 的配合使用就算是全部完成了。  
  
在之后我们写代码的过程中，只需要保存代码，那么 `perttier` 就会帮助我们自动格式化代码，使其符合 `ESLint` 的校验规则。而无需我们手动进行更改了。  
  
  
  
*## 07：大厂编程规范三：约定式提交规范*在前面我们通过 `prettier + ESLint` 解决了代码格式的问题，但是我们之前也说过 \*\*编程规范\*\* 指的可不仅仅只是 \*\*代码格式规范\*\* 。  
  
除了 \*\*代码格式规范\*\* 之外，还有另外一个很重要的规范就是 \*\*`git` 提交规范！\*\*  
  
在现在的项目开发中，通常情况下，我们都会通过 `git` 来管理项目。只要通过 `git` 来管理项目，那么就必然会遇到使用 `git` 提交代码的场景  
  
当我们执行 `git commit -m "描述信息"` 的时候，我们知道此时必须添加一个描述信息。但是中华文化博大精深，不同的人去填写描述信息的时候，都会根据自己的理解来进行描述。  
  
而很多人的描述 “天马行空” ，这样就会导致别人在看你的提交记录时，看不懂你说的什么意思？不知道你当前的这次提交到底做了什么事情？会不会存在潜在的风险？  
  
比如说，我们来看这几条提交记录：  
  
文本

描述已自动生成  
  
你能够想象得到它们经历了什么吗？  
  
所以 \*\*`git` 提交规范\*\* 势在必行。  
  
对于 \*\*`git` 提交规范\*\* 来说，不同的团队可能会有不同的标准，那么咱们今天就以目前使用较多的 [Angular团队规范](*https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#-git-commit-guidelines*) 延伸出的 [Conventional Commits specification（约定式提交）](*https://www.conventionalcommits.org/zh-hans/v1.0.0/*) 为例，来为大家详解 \*\*`git` 提交规范\*\*  
  
约定式提交规范要求如下：  
  
```js  
<type>[optional scope]: <description>  
  
[optional body]  
  
[optional footer(s)]  
  
-------- 翻译 -------------  
   
<类型>[可选 范围]: <描述>  
  
[可选 正文]  
  
[可选 脚注]  
```  
  
其中 `<type>` 类型，必须是一个可选的值，比如：  
  
1. 新功能：`feat`  
2. 修复：`fix`  
3. 文档变更：`docs`  
4. ....  
  
也就是说，如果要按照 \*\*约定式提交规范\*\* 来去做的化，那么你的一次提交描述应该式这个样子的：  
  
文本, 信件

描述已自动生成  
  
我想大家看到这样的一个提交描述之后，心里的感觉应该和我一样是崩溃的！要是每次都这么写，写到猴年马月了！  
  
如果你有这样的困惑，那么 ”恭喜你“ ，接下来我们将一起解决这个问题！  
  
欲知后事如何，请看下一节《Commitizen助你规范化提交代码》  
  
  
  
*## 08：Commitizen助你规范化提交代码*在上一小节我们讲述了 [约定式提交规范](*https://www.conventionalcommits.org/zh-hans/v1.0.0/*) ，我们知道如果严格安装 \*\*约定式提交规范\*\*， 来手动进行代码提交的话，那么是一件非常痛苦的事情，但是 \*\*git 提交规范的处理\*\* 又势在必行，那么怎么办呢？  
  
你遇到的问题，也是其他人所遇到的！  
  
经过了很多人的冥思苦想，就出现了一种叫做 \*\*git 提交规范化工具\*\* 的东西，而我们要学习的 `commitizen` 就是其中的佼佼者！  
  
`commitizen` 仓库名为 [cz-cli](*https://github.com/commitizen/cz-cli*) ，它提供了一个 `git cz` 的指令用于代替 `git commit`，简单一句话介绍它：  
  
> 当你使用 `commitizen` 进行代码提交（git commit）时，`commitizen` 会提交你在提交时填写所有必需的提交字段！  
  
这句话怎么解释呢？不用着急，下面我们就来安装并且使用一下 `commitizen` ，使用完成之后你自然就明白了这句话的意思！  
  
1. 全局安装`Commitizen`  
  
 ```js  
 npm install -g commitizen@4.2.4  
 ```  
  
2. 安装并配置 `cz-customizable` 插件  
 1. 使用 `npm` 下载 `cz-customizable`  
  
 ```node  
 npm i cz-customizable@6.3.0 --save-dev  
 ```  
  
 2. 添加以下配置到 `package.json ` 中  
  
 ```json  
 ...  
 "config": {  
 "commitizen": {  
 "path": "node\_modules/cz-customizable"  
 }  
 }  
 ```  
  
3. 项目根目录下创建 `.cz-config.js` 自定义提示文件  
  
 ```js  
 module.exports = {  
 // 可选类型  
 types: [  
 { value: 'feat', name: 'feat: 新功能' },  
 { value: 'fix', name: 'fix: 修复' },  
 { value: 'docs', name: 'docs: 文档变更' },  
 { value: 'style', name: 'style: 代码格式(不影响代码运行的变动)' },  
 {  
 value: 'refactor',  
 name: 'refactor: 重构(既不是增加feature，也不是修复bug)'  
 },  
 { value: 'perf', name: 'perf: 性能优化' },  
 { value: 'test', name: 'test: 增加测试' },  
 { value: 'chore', name: 'chore: 构建过程或辅助工具的变动' },  
 { value: 'revert', name: 'revert: 回退' },  
 { value: 'build', name: 'build: 打包' }  
 ],  
 // 消息步骤  
 messages: {  
 type: '请选择提交类型:',  
 customScope: '请输入修改范围(可选):',  
 subject: '请简要描述提交(必填):',  
 body: '请输入详细描述(可选):',  
 footer: '请输入要关闭的issue(可选):',  
 confirmCommit: '确认使用以上信息提交？(y/n/e/h)'  
 },  
 // 跳过问题  
 skipQuestions: ['body', 'footer'],  
 // subject文字长度默认是72  
 subjectLimit: 72  
 }  
 ```  
  
4. 使用 `git cz` 代替 `git commit`  
 使用 `git cz` 代替 `git commit`，即可看到提示内容  
  
那么到这里我们就已经可以使用`git cz` 来代替了 `git commit` 实现了规范化的提交诉求了，但是当前依然存在着一个问题，那就是我们必须要通过 `git cz` 指令才可以完成规范化提交！  
  
那么如果有马虎的同事，它们忘记了使用 `git cz` 指令，直接就提交了怎么办呢？  
  
那么有没有方式来限制这种错误的出现呢？  
  
答案是有的！  
  
下一节我们来看 《什么是 Git Hooks》  
  
*## 09：什么是 Git Hooks*上一小节中我们使用了 `git cz` 来代替了 `git commit` 实现了规范化的提交诉求，但是依然存在着有人会忘记使用的问题。  
  
那么这一小节我们就来看一下这样的问题，我们应该如何去进行解决。  
  
先来明确一下我们最终要实现的效果：  
  
> 我们希望：  
>  
> 当《提交描述信息》不符合 [约定式提交规范](https://www.conventionalcommits.org/zh-hans/v1.0.0/) 的时候，阻止当前的提交，并抛出对应的错误提示  
  
而要实现这个目的，我们就需要先来了解一个概念，叫做 `Git hooks（git 钩子 || git 回调方法）`   
  
也就是：\*\*`git` 在执行某个事件之前或之后进行一些其他额外的操作\*\*  
  
而我们所期望的 \*\*阻止不合规的提交消息\*\*，那么就需要使用到 `hooks` 的钩子函数。  
  
下面是我整理出来的所有的 `hooks` ，大家可以进行一下参考，其中加粗的是常用到的 `hooks`：  
  
| Git Hook | 调用时机 | 说明 |  
| :------------------------------ | -------------------------------------------------- | ------------------------------------------------------------ |  
| pre-applypatch | `git am`执行前 | |  
| applypatch-msg | `git am`执行前 | |  
| post-applypatch | `git am`执行后 | 不影响`git am`的结果 |  
| \*\*pre-commit\*\* | `git commit`执行前 | 可以用`git commit --no-verify`绕过 |  
| \*\*commit-msg\*\* | `git commit`执行前 | 可以用`git commit --no-verify`绕过 |  
| post-commit | `git commit`执行后 | 不影响`git commit`的结果 |  
| pre-merge-commit | `git merge`执行前 | 可以用`git merge --no-verify`绕过。 |  
| prepare-commit-msg | `git commit`执行后，编辑器打开之前 | |  
| pre-rebase | `git rebase`执行前 ||  
| post-checkout | `git checkout`或`git switch`执行后 | 如果不使用`--no-checkout`参数，则在`git clone`之后也会执行。 |  
| post-merge | `git commit`执行后 | 在执行`git pull`时也会被调用 |  
| pre-push | `git push`执行前 | |  
| pre-receive | `git-receive-pack`执行前 | |  
| update | | |  
| post-receive | `git-receive-pack`执行后 | 不影响`git-receive-pack`的结果 |  
| post-update | 当 `git-receive-pack`对 `git push` 作出反应并更新仓库中的引用时 | |  
| push-to-checkout | 当``git-receive-pack`对`git push`做出反应并更新仓库中的引用时，以及当推送试图更新当前被签出的分支且`receive.denyCurrentBranch`配置被设置为`updateInstead`时 | |  
| pre-auto-gc | `git gc --auto`执行前 | |  
| post-rewrite | 执行`git commit --amend`或`git rebase`时 | |  
| sendemail-validate | `git send-email`执行前 | |  
| fsmonitor-watchman | 配置`core.fsmonitor`被设置为`.git/hooks/fsmonitor-watchman`或`.git/hooks/fsmonitor-watchmanv2`时 | |  
| p4-pre-submit | `git-p4 submit`执行前 | 可以用`git-p4 submit --no-verify`绕过 |  
| p4-prepare-changelist | `git-p4 submit`执行后，编辑器启动前 | 可以用`git-p4 submit --no-verify`绕过 |  
| p4-changelist | `git-p4 submit`执行并编辑完`changelist message`后 | 可以用`git-p4 submit --no-verify`绕过 |  
| p4-post-changelist | `git-p4 submit`执行后 | |  
| post-index-change | 索引被写入到`read-cache.c do\_write\_locked\_index`后 | |  
  
PS：详细的 `HOOKS介绍` 可点击[这里](https://git-scm.com/docs/githooks)查看  
  
整体的 `hooks` 非常多，当时我们其中用的比较多的其实只有两个：  
  
| Git Hook | 调用时机 | 说明 |  
| :------------- | ------------------------------------------------------------ | ---------------------------------- |  
| \*\*pre-commit\*\* | `git commit`执行前<br />它不接受任何参数，并且在获取提交日志消息并进行提交之前被调用。脚本`git commit`以非零状态退出会导致命令在创建提交之前中止。 | 可以用`git commit --no-verify`绕过 |  
| \*\*commit-msg\*\* | `git commit`执行前<br />可用于将消息规范化为某种项目标准格式。<br />还可用于在检查消息文件后拒绝提交。 | 可以用`git commit --no-verify`绕过 |  
  
简单来说这两个钩子：  
  
1. `commit-msg`：可以用来规范化标准格式，并且可以按需指定是否要拒绝本次提交  
2. `pre-commit`：会在提交前被调用，并且可以按需指定是否要拒绝本次提交  
  
而我们接下来要做的关键，就在这两个钩子上面。  
  
  
  
*## 10：使用 husky + commitlint 检查提交描述是否符合规范要求*在上一小节中，我们了解了 `git hooks` 的概念，那么接下来我们就使用 `git hooks` 来去校验我们的提交信息。  
  
要完成这么个目标，那么我们需要使用两个工具：  
  
1. [commitlint](https://github.com/conventional-changelog/commitlint)：用于检查提交信息  
  
2. [husky](https://github.com/typicode/husky)：是`git hooks`工具  
  
注意：\*\*`npm` 需要在 7.x 以上版本！！！！！\*\*  
  
那么下面我们分别来去安装一下这两个工具：  
  
*### commitlint*1. 安装依赖：  
  
 ```  
 npm install --save-dev @commitlint/config-conventional@12.1.4 @commitlint/cli@12.1.4  
 ```  
  
2. 创建 `commitlint.config.js` 文件  
  
 ```  
 echo "module.exports = {extends: ['@commitlint/config-conventional']}" > commitlint.config.js  
 ```  
  
3. 打开 `commitlint.config.js` ， 增加配置项（ [config-conventional 默认配置点击可查看](*https://github.com/conventional-changelog/commitlint/blob/master/@commitlint/config-conventional/index.js*) ）：  
  
 ```js  
 module.exports = {  
 // 继承的规则  
 extends: ['@commitlint/config-conventional'],  
 // 定义规则类型  
 rules: {  
 // type 类型定义，表示 git 提交的 type 必须在以下类型范围内  
 'type-enum': [  
 2,  
 'always',  
 [  
 'feat', // 新功能 feature  
 'fix', // 修复 bug  
 'docs', // 文档注释  
 'style', // 代码格式(不影响代码运行的变动)  
 'refactor', // 重构(既不增加新功能，也不是修复bug)  
 'perf', // 性能优化  
 'test', // 增加测试  
 'chore', // 构建过程或辅助工具的变动  
 'revert', // 回退  
 'build' // 打包  
 ]  
 ],  
 // subject 大小写不做校验  
 'subject-case': [0]  
 }  
 }  
   
 ```  
  
\*\*注意：确保保存为 `UTF-8` 的编码格式\*\*，否则可能会出现以下错误：  
  
文本

描述已自动生成  
  
接下来我们来安装 `husky`  
  
*### husky*1. 安装依赖：  
  
 ```  
 npm install husky@7.0.1 --save-dev  
 ```  
  
2. 启动 `hooks` ， 生成 `.husky` 文件夹  
  
 ```  
 npx husky install  
 ```  
  
 图形用户界面, 文本

描述已自动生成  
3. 在 `package.json` 中生成 `prepare` 指令（ \*\*需要 npm > 7.0 版本\*\* ）  
  
 ```  
 npm set-script prepare "husky install"  
 ```  
  
 文本

描述已自动生成  
4. 执行 `prepare` 指令  
  
 ```  
 npm run prepare  
 ```  
  
5. 执行成功，提示  
 文本

描述已自动生成  
6. 添加 `commitlint` 的 `hook` 到 `husky`中，并指令在 `commit-msg` 的 `hooks` 下执行 `npx --no-install commitlint --edit "$1"` 指令  
  
 ```  
 npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1"'  
 ```  
  
7. 此时的 `.husky` 的文件结构  
 图形用户界面, 文本, 应用程序

描述已自动生成  
至此， 不符合规范的 commit 将不再可提交：  
  
```  
PS F:\xxxxxxxxxxxxxxxxxxxxx\imooc-admin> git commit -m "测试"  
⧗ input: 测试  
✖ subject may not be empty [subject-empty]  
✖ type may not be empty [type-empty]  
  
✖ found 2 problems, 0 warnings  
ⓘ Get help: https://github.com/conventional-changelog/commitlint/#what-is-commitlint  
  
husky - commit-msg hook exited with code 1 (error)  
```  
  
  
  
那么至此，我们就已经可以处理好了 \*\*强制规范化的提交要求\*\*，到现在 \*\*不符合规范的提交信息，将不可在被提交！\*\*  
  
那么到这里我们的 \*\*规范化目标\*\* 就完成了吗？  
  
当然没有！  
  
现在我们还缺少一个 \*\*规范化的处理\*\* ，那就是 \*\*代码格式提交规范处理！\*\*  
  
有同学看到这里可能说，咦！ 这个怎么看着这么眼熟啊？这个事情我们之前不是做过了吗？还需要在处理什么？  
  
欲知后事如何，请看下一节《通过 pre-commit 处理提交时代码规范》  
  
## 11：通过 pre-commit 检测提交时代码规范  
  
在 \*\*`ESLint` 与 `Prettier` 配合解决代码格式问题\*\* 的章节中，我们讲解了如何处理 \*\*本地！代码格式问题。\*\*  
  
但是这样的一个格式处理问题，他只能够在本地进行处理，并且我们还需要 \*\*手动在 `VSCode` 中配置自动保存\*\* 才可以。那么这样就会存在一个问题，要是有人忘记配置这个东西了怎么办呢？他把代码写的乱七八糟的直接就提交了怎么办呢？  
  
所以我们就需要有一种方式来规避这种风险。  
  
那么想要完成这么一个操作就需要使用 `husky` 配合 `eslint` 才可以实现。  
  
我们期望通过 \*\*`husky` 监测 `pre-commit` 钩子，在该钩子下执行 `npx eslint --ext .js,.vue src`\*\* 指令来去进行相关检测：  
  
1. 执行 `npx husky add .husky/pre-commit "npx eslint --ext .js,.vue src"` 添加 `commit` 时的 `hook` （`npx eslint --ext .js,.vue src` 会在执行到该 hook 时运行）  
  
2. 该操作会生成对应文件 `pre-commit`：  
 图形用户界面, 文本

描述已自动生成  
3. 关闭 `VSCode` 的自动保存操作  
  
4. 修改一处代码，使其不符合 `ESLint` 校验规则  
  
5. 执行 \*\*提交操作\*\* 会发现，抛出一系列的错误，代码无法提交  
  
 ```  
 PS F:\xxxxxxxxxxxxxxxxxxx\imooc-admin> git commit -m 'test'  
   
 F:\xxxxxxxxxxxxxxxx\imooc-admin\src\views\Home.vue  
 13:9 error Strings must use singlequote quotes  
   
 ✖ 1 problem (1 error, 0 warnings)  
 1 error and 0 warnings potentially fixable with the `--fix` option.  
   
 husky - pre-commit hook exited with code 1 (error)  
 ```  
  
   
  
6. 想要提交代码，必须处理完成所有的错误信息  
  
那么到这里位置，我们已经通过 `pre-commit` 检测到了代码的提交规范问题。  
  
那么到这里就万事大吉了吗？  
  
在这个世界上从来不缺的就是懒人，错误的代码格式可能会抛出很多的 `ESLint` 错误，让人看得头皮发麻。严重影响程序猿的幸福指数。  
  
那么有没有办法，让程序猿在 0 配置的前提下，哪怕代码格式再乱，也可以 \*\*”自动“\*\* 帮助他修复对应的问题，并且完成提交呢？  
  
你别说，还真有！  
  
那么咱们来看下一节《lint-staged 自动修复格式错误》  
  
## 12：lint-staged 自动修复格式错误  
  
在上一章中我们通过 `pre-commit` 处理了 \*\*检测代码的提交规范问题，当我们进行代码提交时，会检测所有的代码格式规范\*\* 。  
  
但是这样会存在两个问题：  
  
1. 我们只修改了个别的文件，没有必要检测所有的文件代码格式  
2. 它只能给我们提示出对应的错误，我们还需要手动的进行代码修改  
  
那么这一小节，我们就需要处理这两个问题  
  
那么想要处理这两个问题，就需要使用另外一个插件 [lint-staged](https://github.com/okonet/lint-staged) ！  
  
[lint-staged](https://github.com/okonet/lint-staged) 可以让你当前的代码检查 \*\*只检查本次修改更新的代码，并在出现错误的时候，自动修复并且推送\*\*  
  
[lint-staged](https://github.com/okonet/lint-staged) 无需单独安装，我们生成项目时，`vue-cli` 已经帮助我们安装过了，所以我们直接使用就可以了  
  
1. 修改 `package.json` 配置  
  
 ```js  
 "lint-staged": {  
 "src/\*\*/\*.{js,vue}": [  
 "eslint --fix",  
 "git add"  
 ]  
 }  
 ```  
  
2. 如上配置，每次它只会在你本地 `commit` 之前，校验你提交的内容是否符合你本地配置的 `eslint`规则(这个见文档 [ESLint](https://panjiachen.github.io/vue-element-admin-site/zh/guide/advanced/eslint.html) )，校验会出现两种结果：  
  
 1. 如果符合规则：则会提交成功。  
 2. 如果不符合规则：它会自动执行 `eslint --fix` 尝试帮你自动修复，如果修复成功则会帮你把修复好的代码提交，如果失败，则会提示你错误，让你修好这个错误之后才能允许你提交代码。  
  
3. 修改 `.husky/pre-commit` 文件  
  
 ```js  
 #!/bin/sh  
 . "$(dirname "$0")/\_/husky.sh"  
   
 npx lint-staged  
   
 ```  
  
4. 再次执行提交代码  
  
5. 发现 \*\*暂存区中\*\* 不符合 `ESlint` 的内容，被自动修复  
  
  
  
## 13：关于 `vetur` 检测 `template` 的单一根元素的问题（图文）  
  
本章节为 \*\*图文\*\*，请点击 [这里](./图文课程/2-13：关于vetur检测 template的单一根元素的问题.md) 查看对应文档  
  
## 14：总结  
  
本章中我们处理了 \*\*编程格式规范的问题\*\*，整个规范大体可以分为两大类：  
  
1. 代码格式规范  
2. `git` 提交规范  
  
\*\*代码格式规范：\*\*  
  
对于 \*\*代码格式规范\*\* 而言，我们通过 `ESLint` + `Prettier` + `VSCode 配置` 配合进行了处理。  
  
最终达到了在保存代码时，自动规范化代码格式的目的。  
  
\*\*`git` 提交规范：\*\*  
  
对于 \*\*`git` 提交规范\*\* 而言我们使用了 `husky` 来监测 `Git hooks` 钩子，并且通过以下插件完成了对应的配置：  
  
1. [约定式提交规范](https://www.conventionalcommits.org/zh-hans/v1.0.0/)  
2. [commitizen](https://github.com/commitizen/cz-cli)：git 提交规范化工具  
3. [commitlint](https://github.com/conventional-changelog/commitlint)：用于检查提交信息  
4. `pre-commit`： `git hooks` 钩子  
5. [lint-staged](https://github.com/okonet/lint-staged)：只检查本次修改更新的代码，并在出现错误的时候，自动修复并且推送  
  
那么处理完成这些规范操作之后，在下一章我们将会正式进入到咱们的项目开发之中！