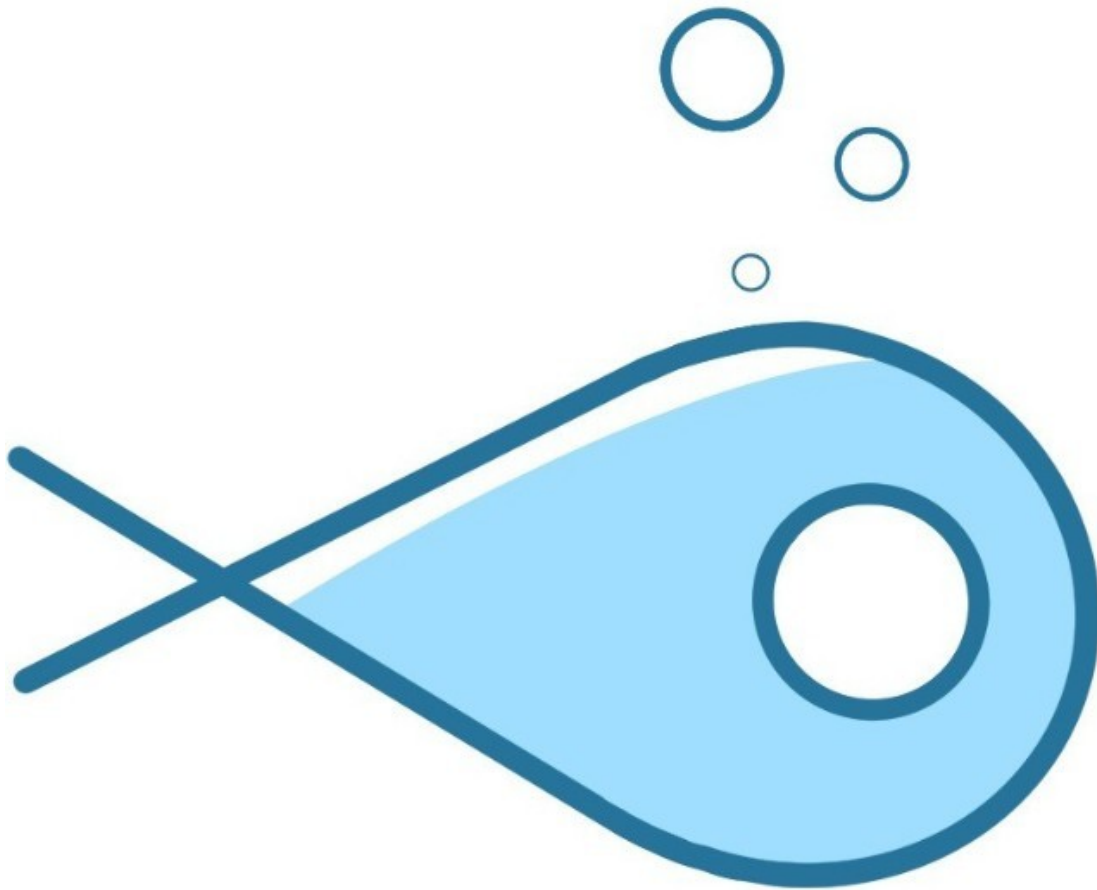


# ESLint 开始，说透我如何在团队项目中基于 Vue 做代码校验

 [mp.weixin.qq.com/s/f3WzR5lXTZ4FNuzMqlXQdA](https://mp.weixin.qq.com/s/f3WzR5lXTZ4FNuzMqlXQdA)

前端大全 2021-08-13 11:50

以下文章来源于大海我来了，作者布兰



大海我来了。

前端小小鱼儿，游啊游游啊游，朝着大海的方向~~~

↓推荐关注↓



大前端技术之路

最近遇到了一个老项目，比较有意思的是这个项目集前后端的代码于一起，而后端也会去修改前端代码，所以就出现了后端用 IntelliJ IDEA 来开发前端项目，而前端用 VSCode 来开发前端项目的情况。于是乎，出现了代码规范的问题，所以就有了这篇文章，整理了一下前端代码校验以及在 Vue 项目中的实践。

阅读完这篇文章，你可以收获：

- 能够自己亲手写出一套 ESLint 配置；
- 会知道业界都有哪些著名的 JS 代码规范，熟读它们可以让你写出更规范的代码；
- vue-cli 在初始化一个包含代码校验的项目时都做了什么；
- Prettier 是什么？为什么要使用它？如何与 ESLint 配合使用？
- EditorConfig 又是什么？如何使用？
- 如何在 VSCode 中通过插件来协助代码校验工作；
- 如何保证 push 到远程仓库的代码是符合规范的；

下面开始阅读吧，如果你对 ESLint 比较熟悉，可以直接跳过这个部分。

## ESLint 是什么

---

ESLint 是一个集代码审查和修复的工具，它的核心功能是通过配置一个个规则来限制代码的合法性和风格。

### 配置解析器和解析参数

---

ESLint 的解析器，早期的时候用的是 Esprima<sup>[1]</sup>，后面基于 Esprima v1.2.2 版本开发了一个新的解析器 Espree<sup>[2]</sup>，并且把它当做默认解析器。

除了使用 ESLint 自带的解析器外，还可以指定其他解析器：

- @babel/eslint-parser：使 Babel 和 ESLint 兼容，对一些 Babel 语法提供支持；
- @typescript-eslint/parser：TSLint 被弃用后，TypeScript 提供了此解析器用于将其与 ESTree 兼容，使 ESLint 对 TypeScript 进行支持；

为项目指定某个选择器的原则是什么？

- 如果你的项目用到了比较新的 ES 语法，比如 ES2021 的 Promise.any()，那就可以指定 @babel/eslint-parser 为解析器；
- 如果项目是基于 TS 开发的，那就使用 @typescript-eslint/parser；

除了指定解析器 parser 外，还可以额外配置解析器参数 parserOption：

```

{
  // ESLint 默认解析器，也可以指定成别的
  parser: "espreet",
  parserOption: {
    // 指定要使用的 ECMAScript 版本，默认值 5
    ecmaVersion: 5,
    // 设置为 script (默认) 或 module (如果你的代码是 ECMAScript 模块)
    sourceType: "script",
    // 这是个对象，表示你想使用的额外的语言特性，所有选项默认都是 false
    ecmaFeatures: {
      // 是否允许在全局作用域下使用 return 语句
      globalReturn: false,
      // 是否启用全局 strict 模式 (严格模式)
      impliedStrict: false,
      // 是否启用JSX
      jsx: false,
      // 是否启用对实验性的objectRest/sreadProperties的支持
      experimentalObjectRestSpread: false
    }
  }
}

```

## 指定环境 env

指定不同的环境可以给对应环境下提供预设的全局变量。比如说在 browser 环境下，可以使用 window 全局变量；在 node 环境下，可以使用 process 全局变量等；

ESLint 中可配置的环境比较多，这里有份完整的环境列表，下面列出几个比较常见的：

- browser：浏览器全局变量；
- node：Node.js 全局变量和作用域；
- es6：es6 中除了模块之外的其他特性，同时将自动设置 parserOptions.ecmaVersion 参数为 6；以此类推 ES2017 是 7，而 ES2021 是 12；
- es2017：parserOptions.ecmaVersion 为 8；
- es2020：parserOptions.ecmaVersion 为 11；
- es2021：parserOptions.ecmaVersion 为 12；

配置方式如下：

```

{
  env: {
    browser: true,
    node: true,
    es6: true,
    commonjs: true,
    mocha: true,
    jquery: true,
  }
}

```

可以指定多个环境并不意味着配置的环境越多越好，实际配置的时候还是得依据当前项目的环境来选择。

## 配置全局变量 globals

ESLint 的一些核心规则依赖于对代码在运行时可用的全局变量的了解。由于这些在不同环境之间可能会有很大差异，并且在运行时可能会进行修改，因此 ESLint 不会假设你的执行环境中存在哪些全局变量。

如果你想使用这些全局变量，那就可以通过 globals 来指定。比如在 react.eslintrc.js 里就把 spyOnDev、spyOnProd 等变量挂在了 global 下作为全局变量：

```
{
  globals: {
    spyOnDev: true,
    spyOnProd: true,
  }
}
```

对于它的值需要特别说明下：

- false、readable、readonly 这 3 个是等价的，表示变量只可读不可写；
- true、writeable、writable 这 3 个是等价的，表示变量可读可写；

## 配置扩展 extends

---

实际项目中配置规则的时候，不可能团队一条一条的去商议配置，太费精力了。通常的做法是使用业内大家普遍使用的、遵循的编码规范；然后通过 extends 去引入这些规范。extends 配置的时候接受字符串或者数组：

```
{
  extends: [
    'eslint:recommended',
    'plugin:vue/essential',
    'eslint-config-standard', // 可以缩写成 'standard'
    '@vue/prettier',
    './node_modules/coding-standard/.eslintrc-es6'
  ]
}
```

从上面的配置，可以知道 extends 支持的配置类型可以是以下几种

- eslint 开头的：是 ESLint 官方的扩展；
- plugin 开头的：是插件类型扩展，比如 plugin:vue/essential；
- eslint-config 开头的：来自 npm 包，使用时可以省略前缀 eslint-config-，比如上面的可以直接写成 standard；
- @开头的：扩展和 eslint-config 一样，只是在 npm 包上面加了一层作用域 scope；
- 一个执行配置文件的相对路径或绝对路径；

那有哪些常用的、比较著名扩展可以被 extends 引入呢

- eslint:recommended<sup>[3]</sup>：ESLint 内置的推荐规则，即 ESLint Rules 列表中打了钩的那些规则；
- eslint:all<sup>[4]</sup>：ESLint 内置的所有规则；
- eslint-config-standard<sup>[5]</sup>：standard 的 JS 规范；
- eslint-config-prettier<sup>[6]</sup>：关闭和 ESLint 中以及其他扩展中有冲突的规则；
- eslint-config-airbnb-base<sup>[7]</sup>：airbab 的 JS 规范；
- eslint-config-alloy<sup>[8]</sup>：腾讯 AlloyTeam 前端团队出品，可以很好的针对你项目的技术栈进行配置选择，比如可以选 React、Vue（现已支持 Vue 3.0）、TypeScript 等；

## 使用插件 plugins

---

### ESLint 提供插件是干嘛用的

ESLint 虽然可以定义很多的 rules，以及通过 extends 来引入更多的规则，但是说到底只是检查 JS 语法。如果需要检查 Vue 中的 template 或者 React 中的 jsx，就束手无策了。所以引入插件的目的就是为了增强 ESLint 的检查能力和范围。

### 如何配置插件

ESLint 相关的插件的命名形式有 2 种：不带命名空间的和带命名空间的，比如：

- eslint-plugin- 开头的可以省略这部分前缀；
- @/ 开头的；

```
{
  plugins: [
    'jquery',          // 是指 eslint-plugin-jquery
    '@jquery/jquery',  // 是指 @jquery/eslint-plugin-jquery
    '@foobar',         // 是指 @foobar/eslint-plugin
  ]
}
```

当需要基于插件进行 extends 和 rules 的配置的时候，需要加上插件的引用，比如：

```
{
  plugins: [
    'jquery',    // eslint-plugin-jquery
    '@foo/foo',  // @foo/eslint-plugin-foo
    '@bar',      // @bar/eslint-plugin
  ],
  extends: [
    'plugin:jquery/recommended',
    'plugin:@foo/foo/recommended',
    'plugin:@bar/recommended'
  ],
  rules: {
    'jquery/a-rule': 'error',
    '@foo/foo/some-rule': 'error',
    '@bar/another-rule': 'error'
  },
}
```

以上配置来自 ESLint plugins

## 配置规则 rules

ESLint 提供了大量内置的规则，这里是它的规则列表 ESLint Rules<sup>[9]</sup>，除此之外你还可以通过插件来添加更多的规则。

### 规则的校验说明，有 3 个报错等级

- off 或 0：关闭对该规则的校验；
- warn 或 1：启用规则，不满足时抛出警告，且不会退出编译进程；
- error 或 2：启用规则，不满足时抛出错误，且会退出编译进程；

通常规则只需要配置开启还是关闭即可；但是也有些规则可以传入属性，比如：

```
{
  rules: {
    'quotes': ['error', 'single'], // 如果不是单引号，则报错
    'one-var': ['error', {
      'var': 'always', // 每个函数作用域中，只允许 1 个 var 声明
      'let': 'never',  // 每个块作用域中，允许多个 let 声明
      'const': 'never', // 每个块作用域中，允许多个 const 声明
    }]
  }
}
```

如何知道某个扩展有哪些规则可以配置，以及每个规则具体限制？这里直接给出业内著名且使用比较多的规则列表的快速链接：

- ESLint rules，这个列表对应 eslint:all，而打钩 ✓ 的是 eslint:recommended；
- Prettier rules

- standard rules
- airbnb rules
- AlloyTeam vue rules

## 规则的优先级

- 如果 extends 配置的是一个数组，那么最终会将所有规则项进行合并，出现冲突的时候，后面的会覆盖前面的；
- 通过 rules 单独配置的规则优先级比 extends 高；

## 其他配置

---

### 配置当前目录为 root

ESLint 检测配置文件步骤：

在要检测的文件同一目录里寻找 .eslintrc.\* 和 package.json；  
紧接着在父级目录里寻找，一直到文件系统的根目录；  
如果在前两步发现有 root: true 的配置，停止在父级目录中寻找 .eslintrc；  
如果以上步骤都没有找到，则回退到用户主目录 ~/.eslintrc 中自定义的默认配置；

通常我们都习惯把 ESLint 配置文件放到项目根目录，因此可以为了避免 ESLint 校验的时候往父级目录查找配置文件，所以需要在配置文件中加上 root: true。

```
{
  root: true,
}
```

### 添加共享数据

ESLint 支持在配置文件添加共享设置，你可以添加 settings 对象到配置文件，它将提供给每一个将被执行的规则。如果你想添加的自定义规则而且使它们可以访问到相同的信息，这将会很有用，并且很容易配置：

```
{
  settings: {
    sharedData: 'Hello'
  },
}
```

参考：ESLint配置文件.eslintrc参数说明

### 针对个别文件设置新的检查规则

比如 webpack 的中包含了某些运行时的 JS 文件，而这些文件是只跑在浏览器端的，所以需要针对这部分文件进行差异化配置：

```
overrides: [
  {
    files: ["lib/**/*.runtime.js", "hot/*.js"],
    env: {
      es6: false,
      browser: true
    },
    globals: {
      Promise: false
    },
    parserOptions: {
      ecmaVersion: 5
    }
  }
]
```

以上配置来自 webpack.eslintrc.js

## 如何校验

---

上面细说了 ESLint 的各种配置项，以及针对 Vue 项目如何进行差异配置的说明。

现在我们知道了如何配置，但是你知道这些配置都是配置到哪里的吗？

## 配置方式

---

ESLint 支持 3 种配置方式：

- 命令行：不推荐，不做介绍；
- 单文件内注释：不推荐，不做介绍；
- 配置文件：配置文件的类型可以是好几种，比如：.js、.yaml、json 等。推荐使用 .eslintrc.js；

下面通过命令来生成一个配置文件：

```
# 安装 eslint
npm i eslint -D

# 初始化一个配置文件
npx eslint --init
```

最后会在当前目录生成一个 .eslintrc.js 文件。这里就不把代码贴出来了，没参考意义。

上面我们知道了可以将配置统一写到一个配置文件里，但是你知道该如何去触发这个配置文件的校验规则嘛？

## 校验单个文件

---

```
// 校验 a.js 和 b.js
npx eslint a.js b.js

// 校验 src 和 scripts 目录
npx eslint src scripts
```

## 校验别的类型的文件

---

通常 ESLint 只能校验 JS 文件。比如需要校验 .vue 文件，光配置 vue 插件和 vue-eslint-parser 解析器是不够的，还需要让 ESLint 在查找文件的时候找到 .vue 文件。

可以通过 --ext 来指定具体需要校验的文件：

```
npx eslint --ext .js,.jsx,.vue src
```

## 自动修复部分校验错误的代码

---

rules 列表项中标识了一个扳手  图案的规则就标识该规则是可以通过 ESLint 工具自动修复代码的。如何自动修复呢？通过 --fix 即可。比如对于 ESLint Rules 里的这个 semi 规则，它就是带扳手图案的。

对于如下的 a.js 代码：

```
const num = 12
```

当在配置文件配置了 'semi': [2, 'always'] 后，运行命令：

```
npx eslint --fix a.js
```

校验直接就通过了，且会自动修复代码，在代码末尾自动加上分号。

## 把校验命令加到 package.json

---

检验命令比较长，也难记，习惯上会把这些命名直接写到 package.json 里：

```
{
  "scripts": {
    "lint": "npx eslint --ext .js,.jsx,.vue src",
    "lint:fix": "npx eslint --fix --ext .js,.jsx,.vue src",
  }
}
```

## 过滤一些不需要校验的文件

---

对于一些公共的 JS、测试脚本或者是特定目录下的文件习惯上是不需要校验的，因此可以在项目根目录通过创建一个 .eslintignore 文件来配置，告诉 ESLint 校验的时候忽略它们：

```
public/
src/main.js
```

除了 .eslintignore 中指定的文件或目录，ESLint 总是忽略 /node\_modules/ 和 /bower\_components/ 中的文件；因此对于一些目前解决不了的规则报错，但是如果又急于打包上线，在不影响运行的情况下，我们就可以利用 .eslintignore 文件将其暂时忽略。

## 在 Vue 项目中的实践

---

上面把 ESLint 的几乎所有的配置参数和校验方式都详细的介绍了一遍，但是如果想在项目中落地，仅仅靠上面的知识还是不够的。下面将细说如何在 Vue 中落地代码校验。

关于如何在 Vue 中落地代码校验，一般是有 2 种情况：

- 通过 vue-cli 初始化项目的时候已经选择了对应的校验配置
- 对于一个空的 Vue 项目，想接入代码校验

其实这 2 种情况最终的校验的核心配置都是一样的，只是刚开始的时候安装的包有所区别。下面通过分析 vue-cli 配置的代码校验，来看看它到底做了哪些事情，通过它安装的包以及包的作用，我们就会知道如何在空项目中配置代码校验了。

## 通过 vue-cli 初始化的项目

---

如果你的项目最初是通过 vue-cli 新建的，那么在新建的时候会让你选



- 是否支持 eslint；
- 是否开启保存校验；
- 是否开启提交前校验；

如果都开启了话，会安装如下几个包：

- eslint：前面 2 大章节介绍的就是这玩意，ESLint 出品，是代码校验的基础包，且提供了很多内置的 Rules，比如 eslint:recommended 经常被作为项目的 JS 检查规范被引入；
- babel-eslint：一个对 Babel 解析器的包装，使其能够与 ESLint 兼容；
- lint-staged：请看后面 pre-commit 部分；
- @vue/cli-plugin-eslint
- eslint-plugin-vue

下面重点介绍 @vue/cli-plugin-eslint 和 eslint-plugin-vue，说下这 2 个包是干嘛的。

## @vue/cli-plugin-eslint

---

这个包它主要干了 2 件事情：

### 第一件事

往 package.json 里注册了一个命令：

```
{
  "scripts": {
    "lint": "vue-cli-service lint"
  }
}
```

执行这个命令之后，它会去检查和修复部分可以修复的问题。默认查找的文件是 src 和 tests 目录下所有的 .js, .jsx, .vue 文件，以及项目根目录下所有的 js 文件（比如，也会检查 .eslintrc.js）。

当然你也可以自定义的传入参数和校验文件：

```
vue-cli-service lint [options] [...files]
```

支持的参数如下：

- --no-fix: 不会修复 errors 和 warnings；
- --max-errors [limit]：指定导致出现 npm ERR 错误的最大 errors 数量；

### 第二件事

增加了代码保存触发校验的功能 lintOnSave，这个功能默认是开启的。如果想要关闭这个功能，可以在 vue.config.js 里配置，习惯上只开启 development 环境下的代码保存校验功能：

```
module.exports = {
  lintOnSave: process.env.NODE_ENV === 'development',
}
```

lintOnSave 参数说明：

- true 或者 warning：开启保存校验，会将 errors 级别的错误在终端中以 WARNING 的形式显示。默认的，WARNING 将不会导致编译失败；
- false：不开启保存校验；
- error：开启保存校验，会将 errors 级别的错误在终端中以 ERROR 的形式出现，会导致编译失败，同时浏览器页面变黑，显示 Failed to compile。

eslint-plugin-vue 是对 .vue 文件进行代码校验的插件。

针对这个插件，它提供了这几个扩展

- plugin:vue/base : 基础
- plugin:vue/essential : 预防错误的 (用于 Vue 2.x)
- plugin:vue/recommended : 推荐的, 最小化任意选择和认知开销 (用于 Vue 2.x) ;
- plugin:vue/strongly-recommended : 强烈推荐, 提高可读性 (用于 Vue 2.x) ;
- plugin:vue/vue3-essential : (用于 Vue 3.x)
- plugin:vue/vue3-strongly-recommended : (用于 Vue 3.x)
- plugin:vue/vue3-recommended : (用于 Vue 3.x)

各扩展规则列表: vue rules

看到这么一堆的扩展, 是不是都不知道选哪个了

代码规范的东西, 原则还是得由各自的团队去磨合商议出一套适合大家的规则。不过, 如果你用的是 Vue2, 我这里可以推荐 2 套 extends 配置:

```
{
  // Vue 官方示例上的配置
  extends: ['eslint:recommended', 'plugin:vue/recommended'],

  // 或者使用 AlloyTeam 团队那套
  extends: ['alloy', 'alloy/vue']
}
```

### 配置和插件对应的解析器

如果是 Vue 2.x 项目, 配置了 eslint-plugin-vue 插件和 extends 后, template 校验还是会失效, 因为不管是 ESLint 默认的解析器 Espree 还是 babel-eslint 都只能解析 JS, 无法解析 template 的内容。

而 vue-eslint-parser 只能解析 template 的内容, 但是不会解析 JS, 因此还需要对解析器做如下配置:

```
{
  parser: 'vue-eslint-parser',
  parseOptions: {
    parser: 'babel-eslint',
    ecmaVersion: 12,
    sourceType: 'module'
  },
  extends: [
    'eslint:recommended',
    'plugin:vue/recommended'
  ],
  plugins: ['vue']
}
```

参考: eslint-plugin-vue faq

## 让 Prettier 管控代码风格

---

针对 Prettier 不得不提出以下疑问?

- Prettier 是什么?
- 为什么有了 ESLint, 还需要引入 Prettier 呢? 它两之间有什么区别?
- 如何配置 Prettier?
- Prettier 如何和 ESLint 结合使用?

## Prettier 是什么

---

用它自己的话来说：我是一个自以为是的**代码格式化工具**，而且我支持的文件类型很多，比如：

- JavaScript（包括实验中的特性）
- JSX
- Vue
- TypeScript
- CSS、Less、SCSS
- HTML
- JSON
- Markdown

以及还有一些其他类型的文件。

## Prettier 对比 ESLint

---

我们知道 ESLint 负责了对代码的校验功能，并且主要提供了 2 类规则：

- 检查格式化的规则
- 检查代码质量的规则

说到底 ESLint 就是通过一条条的规则去限制代码的规范，但是这些规则毕竟是有限的，而且更重要的是这些规则的重点并不在代码风格上，所以单凭 ESLint 并不能完全的统一代码风格。

这个时候就需要引入 Prettier 了，因为它干的事就是只管代码格式化，不管代码质量。

┆ Prettier：在代码风格这一块，我一直拿捏的死死的。

## 如何配置 Prettier

---

初始化操作：

```
# 安装包
npm i prettier -D

# 新建 .prettierrc.js
echo module.exports = {} > .prettierrc.js

# 新建 .prettierignore
echo > .prettierignore
```

Prettier 支持可以配置参数不多，总共才 21 个，这里是所有参数的说明 prettier options

所有参数都有默认值，也就是说即使你没有配置 .prettierrc.js，当你用 Prettier 去格式化代码的时候全部都会走默认配置。针对个别参数，你不想用默认设置的话，就可以在 .prettierrc.js 配置具体想要的值。

如下，把项目中会用到的参数进行一个说明：

```

module.exports = {
  printWidth: 80,           // (默认值) 单行代码超出 80 个字符自动换行
  tabWidth: 2,              // (默认值) 一个 tab 键缩进相当于 2 个空格
  useTabs: true,            // 行缩进使用 tab 键代替空格
  semi: false,              // (默认值) 语句的末尾加上分号
  singleQuote: true,        // 使用单引号
  quoteProps: 'as-needed',  // (默认值) 仅仅当必须的时候才会加上双引号
  jsxSingleQuote: true,     // 在 JSX 中使用单引号
  trailingComma: 'all',     // 不用在多行的逗号分隔的句法结构的最后一行的末尾加上逗号
  bracketSpacing: true,     // (默认值) 在括号和对象的文字之间加上一个空格
  jsxBracketSameLine: true, // 把 > 符号放在多行的 JSX 元素的最后一行
  arrowParens: 'avoid',     // 当箭头函数中只有一个参数的时候可以忽略括弧
  vueIndentScriptAndStyle: false, // (默认值) 对于 .vue 文件, 不缩进 <script> 和 <style> 里的内容
  embeddedLanguageFormatting: 'off', // 不允许格式化内嵌的代码块, 比如 markdown 文件里的代码块
};

```

扩展阅读：关于 Trailing commas 你或许想了解更多。

然后通过命令来格式化代码：

```

# 将格式化当前目录及子目录下所有文件
npx prettier --write .

# 检查某个文件是否已经格式化
npx prettier --check src/main.js

```

如果有些文件不想被 Prettier 格式化，可以将其写入到 .prettierrignore 里：

```

build/
package.json
public/
test/*. *

```

## Prettier 和 ESLint 一起干活更配哦

上面介绍了 Prettier 的具体配置，这里主要介绍和 ESLint 结合使用的配置和注意事项。

和 ESLint 配合使用需要用到 eslint-plugin-prettier 这个插件：

```
npm i eslint-plugin-prettier -D
```

配置：

```

{
  plugins: ['prettier'],
  rules: {
    'prettier/prettier': 'error'
  }
}

```

这个插件的工作原理是先调用 Prettier 对你的代码进行格式化，然后会把格式化前后不一致的地方进行标记，通过配置 'prettier/prettier': 'error' 此条规则会将标记地方进行 error 级别的报错提示，然后可以通过 ESLint 的 --fix 自动修复功能将其修复。

## 冲突了怎么办

通过前面的介绍，我们知道 ESLint 也是会对代码风格做一些限制的，而 Prettier 主要就是规范代码风格，所以在把它们结合在一起使用的时候是存在会有一些问题的。对于个别规则，会使得双方在校验后出现代码格式不一致的问题。

那么当 Prettier 和 ESLint 出现冲突之后，该怎么办呢？

用 Prettier 的话来说很简单，只要使用 `eslint-config-prettier` 就可以了。解决冲突的思路就是通过将这个包提供的扩展放到 `extends` 最后面引入，依据 `rules` 生效的优先级，所以它会覆盖前面起冲突的规则，比如：

```
{
  extends: [
    'eslint:recommended',
    'prettier', // 必须放最后
  ],
}
```

除了能覆盖和 ESLint 中起冲突的规则之外，`eslint-config-prettier` 还能覆盖来自以下插件的规则（只列了部分）：

- `eslint-plugin-standard`
- `eslint-plugin-vue`

那 `eslint-config-prettier` 到底提供了哪些覆盖规则呢？直接看这个列表：`eslint-config-prettier rules`

如果想覆盖某些插件的规则，需要引入对应插件的扩展，比如：

```
{
  extends: [
    'standard',
    'plugin:vue/recommended',
    'prettier/standard', // 覆盖 eslint-config-standard
    'prettier/vue',      // 覆盖 eslint-plugin-vue
  ],
}
```

提示：在 `eslint-config-prettier 8.0.0` 版本后，`extends` 不再需要为单独的插件引入对应扩展来覆盖冲突了，统一引入 `'prettier'` 即可。

如果同时使用了 `eslint-plugin-prettier` 和 `eslint-config-prettier` 可以这么配置：

```
{
  extends: ['plugin:prettier/recommended'],
}
```

它其实和下面这些配置是等价的：

```
{
  extends: ['prettier'], // eslint-config-prettier 提供的，用于覆盖起冲突的规则
  plugins: ['prettier'], // 注册 eslint-plugin-prettier 插件
  rules: {
    'prettier/prettier': 'error',
    'arrow-body-style': 'off',
    'prefer-arrow-callback': 'off'
  }
}
```

所以如果是在 Vue 2 项目中配置 ESLint 和 Prettier 会这么配置：

```

{
  parser: 'vue-eslint-parser',
  parseOptions: {
    parser: 'babel-eslint',
    ecmaVersion: 12,
    sourceType: 'module'
  },
  extends: [
    'eslint:recommended',
    'plugin:vue/recommended',
    'plugin:prettier/recommended', // 在前面 Vue 配置的基础上加上这行
  ],
  plugins: ['vue']
}

```

其实如果你的项目是用 vue-cli 初始化的，且选择了 eslint + prettier 方案的话，生成的项目中，.eslintrc.js 配置文件中 extends 的配置是这样的：

```

{
  extends: [
    'plugin:vue/essential',
    'eslint:recommended',
    '@vue/prettier'
  ]
}

```

它的最后一项扩展是 @vue/prettier，这个对应的是 @vue/eslint-config-prettier 这个包，让我们看看这个包下面的 index.js 内容：

```

{
  plugins: ['prettier'],
  extends: [
    require.resolve('eslint-config-prettier'),
    require.resolve('eslint-config-prettier/vue')
  ],
  rules: {
    'prettier/prettier': 'warn'
  }
}

```

这个和我们上面配置的内容是相差无几的，而引入 eslint-config-prettier/vue 是因为这个 @vue/eslint-config-prettier 包依赖的 eslint-config-prettier 版本是 ^6.0.0 版本的，所以在处理冲突的时候需要特别指定和对应类型插件匹配的扩展。

## 让 EditorConfig 助力多编辑器开发吧

---

EditorConfig 是个啥玩意？它可以对多种类型的单文件进行简单的格式化，它提供的配置参数很少：

```

# 告诉 EditorConfig 插件，这是根文件，不用继续往上查找
root = true

# 匹配全部文件
[*]

# 设置字符集
charset = utf-8

# 缩进风格，可选 space、tab
indent_style = tab

# 缩进的空格数，当 indent_style = tab 将使用 tab_width
# 否则使用 indent_size
indent_size = 2
tab_width = 2

# 结尾换行符，可选 lf、cr、crlf
end_of_line = lf

# 在文件结尾插入新行
insert_final_newline = true

# 删除一行中的前后空格
trim_trailing_whitespace = true

# 匹配md结尾的文件
[* .md]
insert_final_newline = false
trim_trailing_whitespace = false

```

虽然它提供的格式化的配置参数很少，就 3 个，缩进风格、是否在文件末尾插入新行和是否删除一行中前后空格。但是它还是非常有必要存在的，理由有 3 个：

- 能够在不同的编辑器和 IDE 中保持一致的代码风格；
- 配合插件打开文件即自动格式化，非常方便
- 支持格式化的文件类型很多；

如果能让以上的配置生效，还得在 VSCode 里安装 EditorConfig for VS Code 这个插件配合使用。

## 重点来了

可以看到 EditorConfig 和 Prettier 会存在一些重复的配置，比如都提供了对缩进的配置参数，所以在实际使用的时候需要避免它们，或者把他们的参数设置为一致。

## 在 VSCode 中支持 ESLint

前面做的配置，都需要执行命令才能进行检查和修复代码，还是挺不方便的，如果我希望编辑完或者保存的时候去检查代码该如何做呢？可以直接在 IDE 里安装 ESLint 插件，因为我使用的是 VSCode，所以这里只介绍在 VSCode 中的配置。

在使用前，需要把 ESLint 扩展安装到 VSCode 里，这里我就不细说安装步骤了。安装完成后，需要在设置里写入配置：

- 在 VSCode 左下角找到一个齿轮 ⚙ 图标，点击后选择设置选项，这个时候打开了设置面板；
- 然后在 VSCode 右上角找到打开设置 (json) 的图标，点击后，会打开 settings.json 文件；
- 然后把以下配置贴进去即可；

```
{
  "eslint.alwaysShowStatus": true, // 总是在 VSCode 显示 ESLint 的状态
  "eslint.quiet": true,           // 忽略 warning 的错误
  "editor.codeActionsOnSave": {    // 保存时使用 ESLint 修复可修复错误
    "source.fixAll": true,
    "source.fixAll.eslint": true
  }
}
```

配置说明，在 ESLint 2.0.4 版本开始：

- 不需要通过 `eslint.validate` 来指定校验的文件类型了，已经自动支持了 `.vue` 文件；
- `editor.codeActionsOnSave` 开启保存自动修复功能；

当这样配置之后呢，每次编辑代码 ESLint 都会实时校验代码，且当保存的时候会自动 `fix`，是不是很方便呢。不过对于有些无法自动 `fix` 的代码就需要你手动去修改了，如果不想修改的话就可以配置 `rules` 把该条规则给关闭掉。

其实在团队开发的时候，最好把针对 VSCode 的配置，写一个文件跟随着项目，一起提交到远程仓库，这样的话就保证了项目成员都是用的这套配置。比如可以在项目根目录新建 `.vscode/settings.json`，然后写入上面的那串配置内容。

## 在提交前做校验 pre-commit

---

以上只是通过 ESLint 自动修复能够修复的错误以及通过 Prettier 进行代码的格式化，但是在实际开发的时候难免会遇到无法 `fix` 的错误，可能开发人员也忘记修改，如果这个时候把代码提交到远程仓库，那就把糟糕的代码给提交上去了。

那么如何杜绝把糟糕的代码提交上去呢？可以通过配置 `git hooks` 的 `pre-commit` 钩子来实现这个目的。主要是利用了 `husky`<sup>[10]</sup> 和 `lint-staged`<sup>[11]</sup> 这 2 个包。`husky` 就是用来配置 `git hooks` 的，而 `lint-staged` 则是对拿到的 `staged` 文件进行处理，比如执行 `npm run lint` 进行代码校验。

具体操作分两种情况：

- `package.json` 和 `.git` 在同一个目录下
- `package.json` 和 `.git` 在不同的目录下

### package.json 和 .git 在同一个目录下

---

1、执行以下命令：

```
npm install lint-staged
```

会自动安装 `lint-staged` 和 `husky` 并且在 `package.json` 里写入 `lint-staged`。

注意：`mrm` 是一个自动化工具，它将根据 `package.json` 依赖项中的代码质量工具来安装和配置 `husky` 和 `lint-staged`，因此请确保在此之前安装并配置所有代码质量工具，如 `Prettier` 和 `ESLint`。

如果上面顺利会在 `package.json` 里写入 `lint-staged`，可以自行修改让它支持 `.vue` 文件的校验：

```
{
  "lint-staged": {
    "**.{js,vue}": "eslint --cache --fix"
  }
}
```

可能你在别的地方有看过如下这种在 `lint-staged` 最后一项加了 `git add` 的配置，这样也是没问题的。需要说明的是，如果 `lint-staged` 是 11.x.x 的版本已经自动实现了将修改过的代码自动添加暂存区的功能，所以不需要再加 `git add`。



```
"lint-staged": {
  "**.{js,vue}": ["eslint --cache --fix", "git add"]
}
```

## 2、启动 git hooks

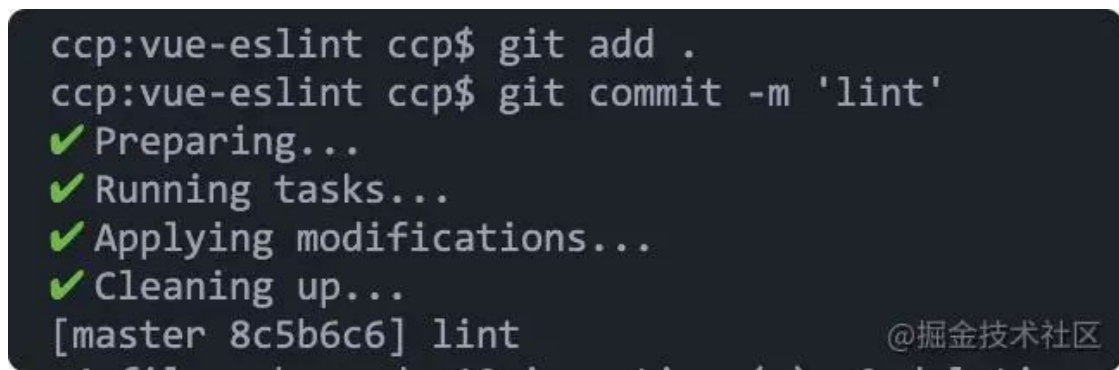
```
npx husky install
```

经过上面的命令后，v6 版本的 husky 会在项目根目录新建一个 .husky 目录。如果是 v4 版本的则会写入到 package.json 里。

## 3、创建 pre-commit 钩子

```
npx husky add .husky/pre-commit "npx lint-staged"
```

到这里后，git commit 前自动执行代码校验和修复的功能就算完成了。然后你可以试试修改文件，然后提交试试。



```
ccp:vue-eslint ccp$ git add .
ccp:vue-eslint ccp$ git commit -m 'lint'
✓ Preparing...
✓ Running tasks...
✓ Applying modifications...
✓ Cleaning up...
[master 8c5b6c6] lint
@掘金技术社区
```

## package.json 和 .git 在不同的目录下

如果你的项目不是纯前端项目，而是前后端代码放在同一个项目下，比如如下这个 shop-system 的项目结构：

```
├-.git           // .git 目录
├-shop-service  // 后端项目
└-shop-web      // 前端项目
```

如果 package.json 和 .git 在不同的目录下，那么通过第一种方式的 npx mrm lint-staged 来创建会失败，将无法创建 .husky 目录。这个时候就需要稍微麻烦点了。

在 shop-web 目录的终端下执行：

```
# 安装 husky
npm install husky --save-dev

# 启动 git hooks
cd .. && husky install shop-web/.husky

# 添加 pre-commit 钩子
npx husky add .husky/pre-commit "cd shop-web && npx lint-staged"

# 安装 lint-staged
npm install lint-staged -D
```

然后在 package.json 里配置 lint-staged，就可以大功告成。

```
{
  "lint-staged": {
    "*.{js,vue}": "eslint --cache --fix"
  }
}
```

参考：自定义目录启动 git hook

## 总结

---

这篇文章比较长，前前后后讲了很多代码校验的东西，现在我们来梳理下。

首先用 ESLint 来做代码校验，它自带的 rules 能提供 2 种类型的校验，分别是代码错误校验和代码格式校验，而 ESLint 本身的核心工作其实就是校验和修复错误的代码，而对格式化的规则提供的不多。

所以如果想要对代码格式化进行一个更加精细的配置则需要借助 Prettier，因为它是只负责风格的管控，所以用它再适合不过了。但是如果把 ESLint 和 Prettier 结合起来一起使用的话，就可能会出现规则的冲突了，毕竟它们两者都会对风格进行处理，所以这个时候就可以通过 eslint-config-prettier 这个扩展来把冲突的规则进行关闭，这个扩展不仅可以关闭和 ESLint 内置规则的冲突，还可以关闭实际项目中引用到的扩展规则的冲突，比如和 Vue、React、TypeScript、Flow 的冲突。

在把 ESLint 和 Prettier 结合的时候，我们希望让 ESLint 来检查代码错误，而 Prettier 校验代码风格，那么这个时候其实是有 2 个任务的，需要用 2 条命令来处理的。但是有了 eslint-plugin-prettier 这个插件后就可以很方便的把它们结合起来，当需要校验代码错误的时候 ESLint 会自动给你校验，当然前提是 VSCode 里必须按照 ESLint 插件，而当需要校验代码风格的时候 ESLint 就会调用 Prettier 的能力进行代码风格的检查。

文章的后面分别又细说了 EditorConfig 和提交代码前校验的处理，这里就不多讲了。

看到这里希望你对代码校验和规范有一个新的认识，不过我最希望的是你能够自己动手为你的项目配置一套校验规则，如果不能成功，一定是我的文章写的有问题，欢迎评论区留言指出不足之处。

## 参考资料

---

[1]  
<http://esprima.org/>: <https://link.juejin.cn?target=http%3A%2F%2Fesprima.org%2F>

[2]  
<https://github.com/eslint/esprez>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Feslint%2Fesprez>

[3]  
<https://eslint.org/docs/rules/>: <https://link.juejin.cn?target=https%3A%2F%2Feslint.org%2Fdocs%2Frules%2F>

[4]  
<https://eslint.org/docs/rules/>: <https://link.juejin.cn?target=https%3A%2F%2Feslint.org%2Fdocs%2Frules%2F>

[5]  
<https://github.com/standard/eslint-config-standard>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fstandard%2Feslint-config-standard>

[6]  
<https://github.com/prettier/eslint-config-prettier>: <https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fprettier%2Feslint-config-prettier>

[7]

[https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base:](https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base)

[https://link.juejin.cn?](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fairbnb%2Fjavascript%2Ftree%2Fmaster%2Fpackages%2Feslint-config-airbnb-base)

[target=https%3A%2F%2Fgithub.com%2Fairbnb%2Fjavascript%2Ftree%2Fmaster%2Fpackages%2Feslint-config-airbnb-base](https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base)

[8]

[https://github.com/AlloyTeam/eslint-config-alloy:](https://github.com/AlloyTeam/eslint-config-alloy) [https://link.juejin.cn?](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2FAlloyTeam%2Feslint-config-alloy)

[target=https%3A%2F%2Fgithub.com%2FAlloyTeam%2Feslint-config-alloy](https://github.com/AlloyTeam/eslint-config-alloy)

[9]

[https://eslint.org/docs/rules/:](https://eslint.org/docs/rules/) [https://link.juejin.cn?](https://link.juejin.cn?target=https%3A%2F%2Feslint.org%2Fdocs%2Frules%2F)

[target=https%3A%2F%2Feslint.org%2Fdocs%2Frules%2F](https://eslint.org/docs/rules/)

[10]

[https://typicode.github.io/husky/#/?id=install:](https://typicode.github.io/husky/#/?id=install) [https://link.juejin.cn?](https://link.juejin.cn?target=https%3A%2F%2Ftypicode.github.io%2Fhusky%2F%23%2F%3Fid%3Dinstall)

[target=https%3A%2F%2Ftypicode.github.io%2Fhusky%2F%23%2F%3Fid%3Dinstall](https://typicode.github.io/husky/#/?id=install)

[11]

[https://github.com/okonet/lint-staged:](https://github.com/okonet/lint-staged) [https://link.juejin.cn?](https://link.juejin.cn?target=https%3A%2F%2Fgithub.com%2Fokonet%2Flint-staged)

[target=https%3A%2F%2Fgithub.com%2Fokonet%2Flint-staged](https://github.com/okonet/lint-staged)

转自：大海我来了

<https://juejin.cn/post/6974223481181306888>

- EOF -

推荐阅读 点击标题可跳转

1、[深入理解 ESLint](#)

2、[Vue 组件设计：实现水波涟漪效果的点击反馈指令](#)

3、[从 Event Loop 角度解读 Vue NextTick 源码](#)

觉得本文对你有帮助？请分享给更多人

推荐关注「前端大全」，提升前端技能



**前端大全**

点击获取精选前端开发资源。「前端大全」日常分享 Web 前端相关的技术文章、实用案例、工具资源、精选课程、热点资讯。

201篇原创内容

公众号

点赞和在看就是最大的支持❤️