# 一文彻底读懂ESLint

> 链接：https://zhuanlan.zhihu.com/p/370666720

在日常项目开发中，ESLint常常扮演者可有可无的角色，我们想让它来帮助我们检查代码，同时又害怕它带来的报错无法处理；本文带你深入的了解ESLint的配置以及原理。

ESLint是一个插件化的代码检测工具，正如它官网描述的slogan：

> 可组装的JavaScript和JSX检查工具

ESLint不仅可以检测JS，还支持JSX和Vue，它的高可扩展性让它能够支持更多的项目。

## ESLint的前辈们

提到ESLint，我们就不得不提及他的前辈们JSLint和JSHint，以及它们的区别；首先就是JSLint，它是由`Douglas Crockford`开发的；JSLint的灵感来源于C语言的检查工具`Lint`，Lint最初被发明用来扫描C语言源文件以便找到其中的错误，后来随着语言的成熟以及编译器能够更好的找到问题，Lint工具也逐渐不再被需要了。

JavaScript最开始被发明只是用来在网页上做一些简单的工作（点击事件、表单提交等），随着JS语言的发展完善以及项目复杂程度的增加，急需一个用来检查JS语法或者其他问题的校验工具，因此`JSLint`就诞生了；它是由`Douglas Crockford`在2010年开源的第一款针对JS的语法检测工具，它和Lint做着相同的事，扫描JS的源文件来找到错误；它内部也是通过`fs.readFile`来读取文件然后逐行来进行检查。


我们可以在全局安装jslint，然后`jslint source.js`对我们的代码进行检查；JSLint刚开始确实帮助很多JS开发者节省了不少排查错误的时间，但是JSLint的问题也很明显：所有的配置项都内置不可配置，因此你要用JSLint只能遵循`Douglas Crockford`老爷子自己定义的代码风格和规范；再加上他本身推崇`爱用不用`的传统，不像开发者开放配置或者修改他觉得对的规则，因此很多人也无法忍受他的规则。

由于JSLint让很多人无法忍受，所以`Anton Kovalyov`基于JSLint开发了JSHint，它的初衷就是为了能让开发者自定义规则`lint rules`，因此提供了丰富的配置项，给开发者极大的自由；同时它也提供了一套相当完善的编辑器插件，我们常用的VIM、Sublime、Atom、Vs Code等都有插件支持，方便开发。


JSHint一开始就保持了开源软件的风格，并且由社区来驱动，因此一推出就很快发展起来，我们熟知的一些项目或者公司也使用了JSHint，比如：Facebook、Google、Jquery、Disqus等。

JSHint相比于JSLint，最大的特点就是可配置，我们可以在项目中放入一个`.jshintrc`的配置文件，JSLint就会加载配置文件用于代码分析，配置文件的部分内容如下：

```
{
    // 禁止有未使用的变量
    "unused": true,
    // 禁止有未定义的变量
    "undef": true,
    // 无视没有加分号行尾
    "asi": true,
    // 全局变量
    "globals": {
        "jQuery": true
    }
}
```

由于JSHint是基于JSLint开发的，因此JSLint的一些问题也继承下来了，比如不易扩展以及不容易直接根据报错定位到具体的配置规则等；在2013年，`zakas`大佬发现JSHint无法满足自己定制化规则的需要，因此设想开发一个基于AST的Linter，可以动态执行额外的规则，同时可以很方面的扩展规则，于是在13年6月份开源推出了全新的ESLint。

ESLint号称下一代的JS Linter工具，它的灵感来源于PHP Linter，将源码解析成AST，然后检测AST是否符合规则；ESLint最开始使用`esprima`解析器将源码解析成AST，然后就可以使用任意规则来检测AST是否符合预期，这也是ESLint高可扩展的原因。

刚开始ESlint的推出并没有撼动JSHint的霸主地位，由于ESlint需要将源码转为AST，而JSHint直接检测源文件字符串，因此执行速度比JSHint慢很多；真正让ESLint实现弯道超车的是ES6的出现。

2015年，ES6规范发布后，由于大部分浏览器支持程度不高，因此需要Babel将代码转换编译成ES5或者更低版本；同时由于ES6变化很大，短期内JSHint无法完全支持，这时ESLint的高扩展性的优点显现出来了，不仅可以扩展规则，连默认的解析器也能替换；Babel团队就为ESLint开发了`babel-eslint`替换默认的解析器`esprima`，让ESLint率先支持ES6。

# 配置

ESLint被设计成完全可配置的，我们可以用多种方式配置它的规则，或者配置要检测文件的范围。

# 初始化

如果想在现有的项目中引入eslint，我们可以在项目中进行初始化：

```
npm i eslint --save-dev
npx eslint --init
```

在经过一系列问答后，会在项目根目录创建一个我们熟悉的`.eslintrc.js`配置文件；安装后就可以通过命令行对项目中的文件需要检测了：

```
# 检测单个文件
npx eslint file1.js file2.js
# 检测src和scripts目录
npx eslint src scripts
```

一般我们会把eslint命令行配置到`packages.json`中：

```
"scripts": {
    "lint": "npx eslint src scripts",
    "lint:fix": "npx eslint src scripts --fix",
    "lint:create": "npx eslint --init"
}
```

这里有一个`--fix`后缀，是ESLint提供自动修复基础错误的功能，我们运行`lint:fix`后发现有一些报错信息消失了，代码也改变了；不过它只能修复一些基础的不影响代码逻辑的错误，比如代码末尾加上分号、表达式的空格等等。

ESLint默认只会检测`.js`后缀的文件，如果我们想对更多类型的文件进行检测，比如.vue、.jsx，可以使用`--ext`选项，参数用逗号分隔：

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

对于一些公共的js，或者测试脚本，不需要进行检测，我们可以通过在项目根目录创建一个`.eslintignore`告诉ESLint去忽略特定的目录或者文件:

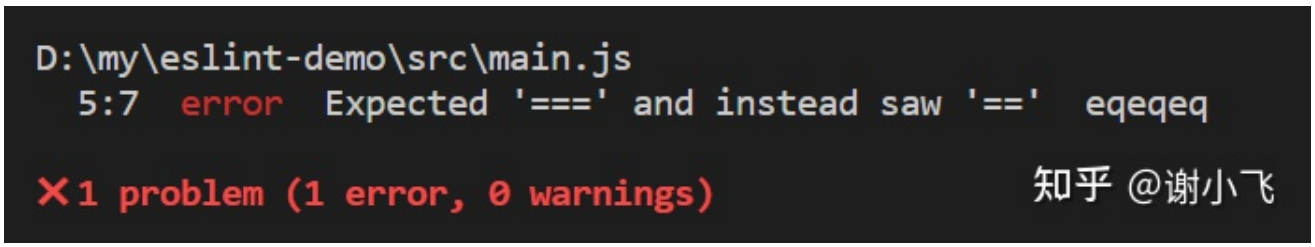
```
public/
src/main.js
```

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，但是我们需要打包上线，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。

ESLint一共有两种配置方式，第一种方式是直接把lint规则嵌入源代码中；

```
/* eslint eqeqeq: "error" */
var num = 1
num == '1'
```

`eqeqeq`代表eslint校验规则，error代表校验报错级别，后面会详细说明；这个eslint校验规则只会对该文件生效:




我们还可以使用其他注释，更精确地管理eslint对某个文件或某一行代码的校验:

```
/* eslint-disable */
alert('该注释放在文件顶部，eslint不会检查整个文件')

/* eslint-enable */
alert('重新启用eslint检查')

/* eslint-disable eqeqeq */
alert('只禁止某一个或多个规则')

/* eslint-disable-next-line */
alert('下一行禁止eslint检查')

alert('当前行禁止eslint检查') // eslint-disable-line
```

第二种方式是直接把lint规则放到我们的配置文件中，上面init初始化生成的`.eslintrc.js`就是一个配置文件，官方还提供了其他几种配置文件名称（优先级从上到下）：

```
.eslintrc.js
.eslintrc.yaml
.eslintrc.yml
.eslintrc.json
.eslintrc
package.json
```

一般情况下我们使用`.eslintrc.js`就可以了。

# 配置详解

我们详细看下`.eslintrc.js`文件内部有哪些配置选项：

```
module.exports = {
    "globals": {},
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": "eslint:recommended",
    "parse": "babel-eslint",
    "parserOptions": {
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "rules": {}
};
```

首先是我们的`globals`，ESLint会检测未声明的变量，并发出报错，比如node环境中的process，浏览器环境下的全局变量console，以及我们通过cdn引入的jQuery定义的$等；我们可以在`globals`中进行变量声明：

```
{
    "globals": {
        // true表示该变量可读写，false表示变量是只读
        "$": true,
        "console": false
    }
}
```

但是node或者浏览器中的全局变量很多，如果我们一个个进行声明显得繁琐，因此就需要用到我们的`env`，这是对环境定义的一组全局变量的预设：

```
{
    "env": {
        "browser": true,
        "node": true,
        "jquery": true
    }
}
```

更多的环境参数可以看ESLint声明环境。

然后就是我们的解析器`parse`和`parserOptions`；我们上面说到ESLint可以更换解析器，`"parse": "babel-eslint"`就是用来指定要使用的解析器，它有以下几个选择：

- esprima：ESLint最开始使用的解析器
- espree：默认，ESLint自己基于esprima v1.2.2开发的一个解析器
- babel-eslint：一个对Babel解析器的包装，使其能够与ESLint兼容。
- @typescript-eslint/parser：将TypeScript转换成与estree兼容的形式，以便在ESLint中使用。

那么这几个解析器怎么选择呢？如果你想使用一些先进的语法（ES6789），就使用babel-eslint（需要npm安装）；如果你想使用typescript，就使用@typescript-eslint/parser。

选好了解析器，我们可以通过`parserOptions`给解析器传入一些其他的配置参数：

```
{
  "parser": "babel-eslint",
  "parserOptions": {
    // 代码模块类型，可选script(默认)，module
    "sourceType": "module",
    // es版本号，默认为5，可以使用年份2015（同6）
    "ecamVersion": 6,
    // es 特性配置
    "ecmaFeatures": {
        "globalReturn": true, // 允许在全局作用域下使用 return 语句
        "impliedStrict": true, // 启用全局 strict mode
        "jsx": true // 启用 JSX
    }
  },
}
```

# 规则

ESLint可以配置大量的规则，我们可以在配置文件的`rules`属性自定义需要的规则：

```
{
    "rules":{
        // "semi": "off",
        "semi": 0,
        // "quotes": "warn",
        "quotes": 1,
        // "no-console": "error"
        "no-console": 2
    }
}
```

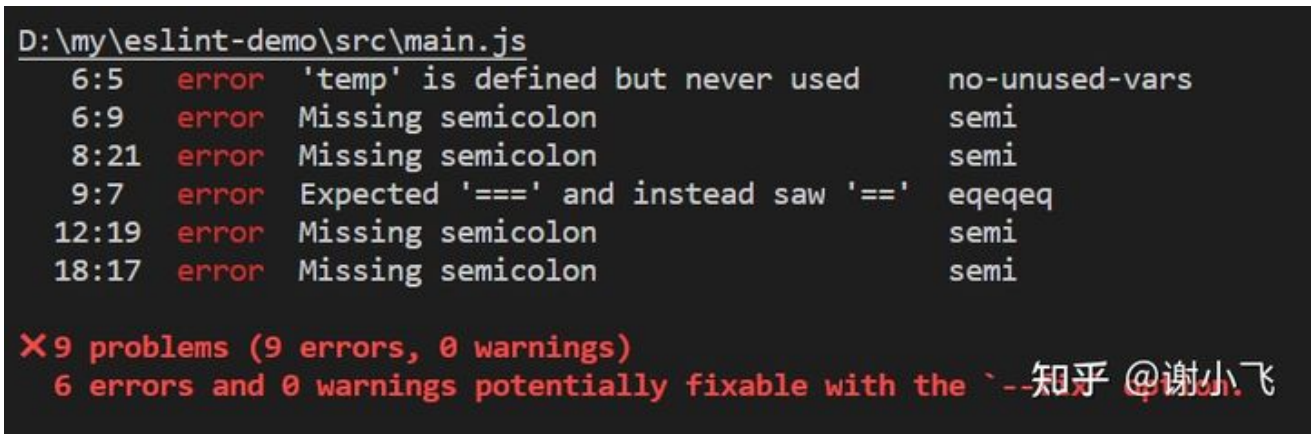
对于检验规则，有3个报错等级：

- "off" 或 0：关闭规则
- "warn" 或 1：开启规则，warn级别的错误 (不会导致程序退出)
- "error" 或 2：开启规则，error级别的错误(当被触发的时候，程序会退出)

有些规则没有属性，只需控制开启还是关闭；有些规则可以传入属性，我们通过数组的方式传入参数：

```
{
    "rules":{
        // 代码缩进，使用tab缩进，switch语句的case缩进级别，1表示2个空格
        "indent": ["error", "tab", { "SwitchCase": 1 }],
        // 引号，双引号
        "quotes": ["error", "double"],
        // 在语句末尾使用分号
        "semi": ["error", "always"]
    }
}
```

对于刚接触ESLint的同学，看到这么多的规则肯定很懵逼，难道要一条一条来记么？肯定不是的；项目的ESLint配置文件并不是一次性完成的，而是在项目开发中慢慢完善起来的，因为并不是所有的规则都是我们项目所需要的。因此我们可以先进行编码，在编码的过程中使用`npm run lint`校验代码规范，如果报错，可以通过报错信息去详细查看是那一条规范报错：




比如这里的报错`no-unused-vars`我们可以看到它来自第六行，再去文档查找，发现是我们在js中有一个定义了却未使用的变量；在团队协商后可以进一步来确定项目是否需要这条规范。

# 扩展

如果每条规则都需要团队协商配置还是比较繁琐的，在项目开始配置时，我们可以先使用一些业内已经成熟的、大家普遍遵循的编码规范（最佳实践）；我们可以通过`extends`字段传入一些规范，它接收String/Array：

```
{
    "extends": [
        "eslint:recommended",
        "plugin:vue/essential",
        "@vue/prettier",
        "eslint-config-standard"
    ]
}
```

extends可以使用以下几种类型的扩展：

- eslint：开头的ESLint官方扩展，有两个：`eslint:recommended`（推荐规范）和`eslint:all`（所有规范）。
- plugin：开头的扩展是插件类型扩展
- eslint-config：开头的来自npm包，使用时可以省略`eslint-config-`，比如上面的可以直接写成`standard`
- @：开头的扩展和eslint-config一样，是在npm包上面加了一层作用域scope

> 需要注意的是：多个扩展中有相同的规则，以后面引入的扩展中规则为准。

`eslint:recommended`推荐使用的规则在规则列表的右侧用`绿色√`标记。


插件类型的扩展一般先通过npm安装插件，以上面的vue为例，我们先来安装：

```
npm install --save-dev eslint eslint-plugin-vue
```

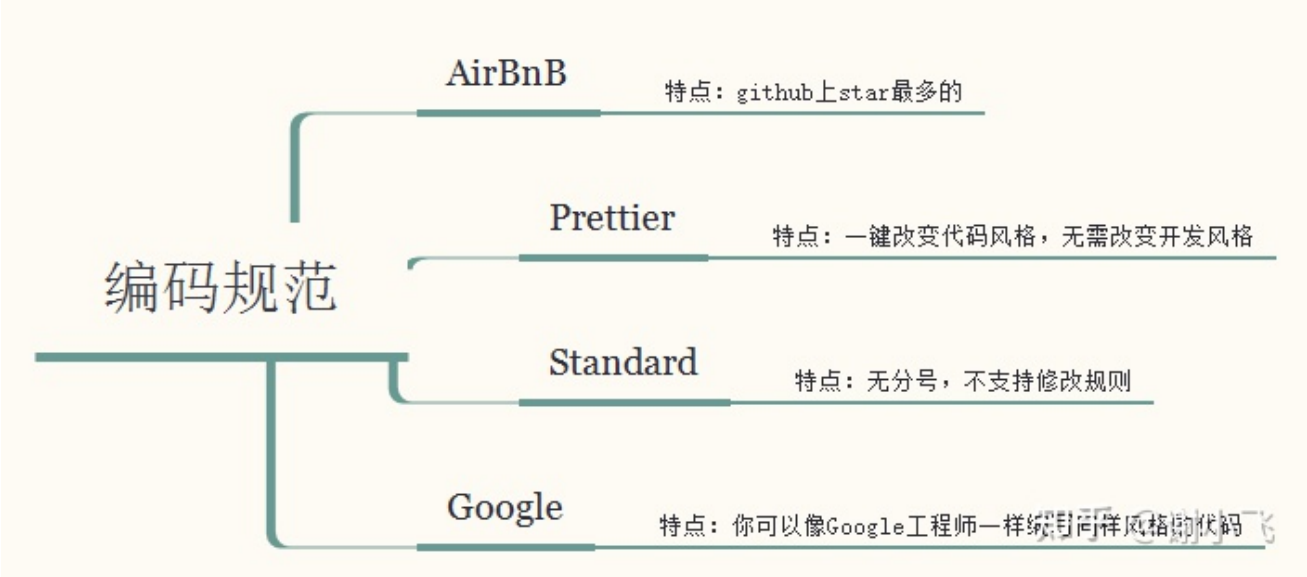
安装后一个插件中会有很多同类型扩展可供选择，比如vue就有以下几种扩展：

- plugin:vue/base：基础
- plugin:vue/essential：必不可少的
- plugin:vue/recommended：推荐的
- plugin:vue/strongly-recommended：强烈推荐

针对扩展中的规则，我们也能够通过rules来对它进行覆写：

```
{
  "extends": [
    "plugin:vue/recommended"
  ],
  "rules": {
    // 覆写规则
    "vue/no-unused-vars": "error"
  }
}
```

除了上面的`eslint-config-standard`，还有以下几个比较知名的编码规范：




不过需要注意的是，很多规范不仅需要安装扩展本身，还需要配合插件，比如`eslint-config-standard`，我们还需要安装下面几个插件才能有效：

```
npm i eslint-config-standard -D
npm i eslint-plugin-promise eslint-plugin-import eslint-plugin-node -D
```

# 插件

在Webpack中，插件是用来扩展功能，让其能够处理更多的文件类型以及功能，ESLint中的插件也是同样的作用；虽然ESLint提供了几百种规则可供选择，但是随着JS框架和语法的发展，这么多规则还是显得不够，因为官方的规则只能检查标准的JS语法；如果我们写的是vue或者react的jsx，那么ESLint就不能检测了。

这时就需要安装ESLint插件，用来定制一些特色的规则进行检测；eslint插件以`eslint-plugin-`开头，使用时可以省略；比如我们上面检测`.vue`文件就用到`eslint-plugin-vue`插件；需要注意的是，我们在配置`eslint-plugin-vue`这个插件时，如果仅配置`"plugins": ["vue"]`，vue文件中template内容还是会解析失败。

这是因为不管是默认的espree还是babel-eslint解析器都无法解析.vue中template的内容；`eslint-plugin-vue`插件依赖`vue-eslint-parser`解析器，而`vue-eslint-parser`解析器只会解析template内容，不会检测`script`中的JS内容，因此我们还需要指定一下解析器：

```
{
    "extends": ["eslint:recommended"],
    "plugins": ["vue"],
    "parser": "vue-eslint-parser",
    "parserOptions": {
        "parser": "babel-eslint",
        "ecmaVersion": 12,
        "sourceType": "module",
    },
}
```

上面`parserOptions.parser`不少同学肯定看的有点迷糊，这是由于外层的解析器只能有一个，我们已经用了`vue-eslint-parser`就不能再写其他的；因此`vue-eslint-parser`的做法是在解析器选项中再传入一个解析器选项用来处理`script`中的JS内容。

> 如果想让ESLint检测vue文件，确保将`.vue`后缀加入`--ext`选项中。

而react配置则较为简单了，引入插件，选择对应的扩展规则即可：

```
{
    "extends": [
        "eslint:recommended",
        "plugin:react/recommended"
    ],
    "parserOptions": {
        // 启用jsx语法支持
        "ecmaFeatures": {
            "jsx": true
        },
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "plugins": [
        "react"
    ],
}
```

# 配合prettier

虽然ESLint会对我们的代码格式进行一些检测（比如分号、单双引号等），但是并不能完全统一代码风格，我们还需要一个工具Prettier；Prettier是什么？Prettier是一个支持很多语言的代码格式化工具，官网用了一个“贬义”的单词来形容它`opinionated`，翻译过来就是固执己见的。

Prettier还有以下四个特点：

- An opinionated code formatter
- Supports many languages
- Integrates with most editors
- Has few options

那么为什么Prettier要用opinionated这个词呢？每个团队成员可能会用不同的编辑器或是不同的插件，每个插件也会有自己的格式化规范，这样就导致了我们在开发时代码风格极大的不统一，甚至造成不必要的冲突；Prettier就给我们定义好了风格，按照它的风格来（是不是很像SLint）；但是又没有完全封闭，开放了一些必要的设置，这也是最后一点`few options`的含义；因此我们只需要将代码的美化交给Prettier来做就好了。

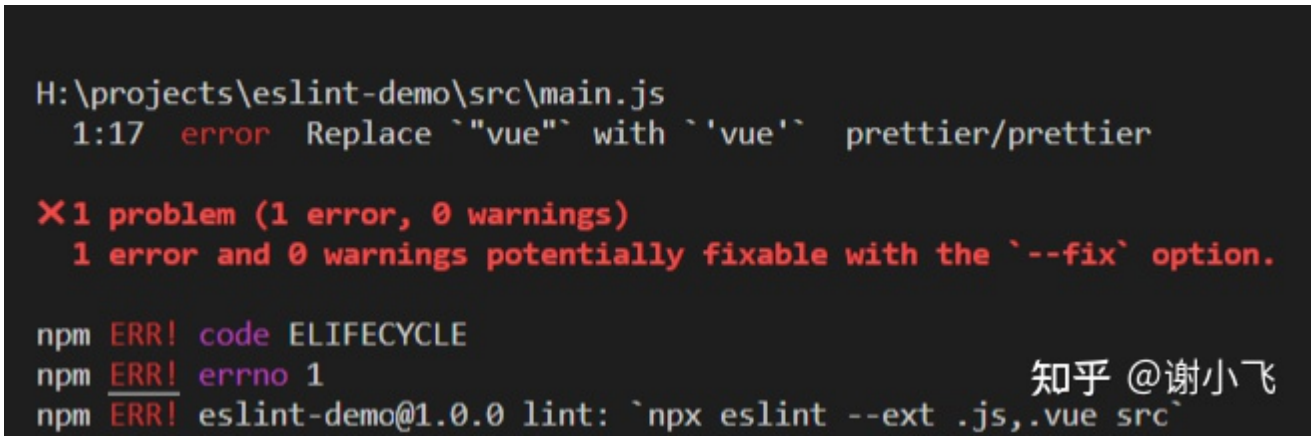
首先还是安装，我们将所需的插件进行安装，这里用到prettier的三个包：

```
npm i prettier
      eslint-plugin-prettier
      eslint-config-prettier
```

首先就是这个eslint-plugin-prettier插件，它会调用prettier对你的代码风格进行检查，其原理是先使用prettier对你的代码进行格式化，然后与格式化之前的代码进行对比，如果过出现了不一致，这个地方就会被prettier进行标记。

被标记后Prettier并不会有任何提示，我们还需要对标记后的代码进行报错处理，在`rules`中进行添加配置：

```
{
    "plugins": ["prettier"],
    "rules": {
        "prettier/prettier": "error",
    }
}
```




> 如果不希望Prettier影响项目打包，我们也可以将prettier的报错由error改为warn

借助ESLint的自动修复`--fix`，我们可以修复这种简单的样式问题；那如果我们想自定义一些样式怎么办呢？没关系，虽然Prettier是一个固执己见的工具，但是人家也是开放了一些配置可供我们进行自定义的，我们可以在项目中新建一个`.prettierrc.json`文件：

```
{
  // 尾逗号
  "trailingComma": "es5",
  // 缩进长度
  "tabWidth": 4,
  // 代码末尾分号
  "semi": false,
  // 单引号
  "singleQuote": true,
  // 单行代码最大长度
  "printWidth": 100,
  // 对象字面量的括号
  "bracketSpacing": true,
  // 箭头函数参数加括号
  "arrowParens": "always",
}
```

这里简单贴一些常用的，我们可以在官网选项配置找到更多的配置规则。

这样配置后虽然能修复代码了，但是如果遇到另一个也固执己见的扩展，比如我们引入`eslint-config-standard`这个扩展，它也有自己的代码风格；如果通过Prettier格式化，standard不干了；如果通过standard自动修复，那么Prettier又要报错了，两边都是大爷这可咋整呢？


机智的Prettier已经帮我们考虑到这个问题了，利用extends中最后一个覆盖前面扩展的特性，我们将`eslint-config-prettier`配置在extends最后，就能够关闭一些与Prettier的规则：

```
{
    "extends": ["standard", "prettier"],
    "plugins": ["prettier"],
    "rules": {
        "prettier/prettier": "error",
    }
}
```

另外eslint-plugin-prettier插件也附带有`plugin:prettier/recommended`扩展配置，可以同时启用插件和eslint-config-prettier扩展，因此我们可以只需要配置recommended就可以了：

```
{
    "extends": ["standard", "plugin:prettier/recommended"],
    "rules": {
        "prettier/prettier": "error",
    }
}
```

Vue中为了支持Prettier，也将eslint-plugin-prettier和eslint-config-prettier整合到一起，放到了`node_modules/@vue/eslint-config-prettier`目录中（加了一层作用域），因此我们在Vue脚手架生成的项目经常能看到`@vue/prettier`这个扩展，打开它的目录发现其本质是一样的：

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