前端编码规范

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文档版本修改记录 | | | | | |
| 版本 | 修改日期 | 修改摘要 | 修改人 | 审批人 | 审批日期 |
| V 1.0 | 2018-10-10 | 初版 | 张印 | YYY | yyyy-mm-dd |
| V 1.1 | 2018-10-18 | 已有项目fix失败 | 张印 |  |  |
| V 1.2 | 2018-10-23 | 1.7描述错误更改 | 张印 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

目录

第1章 Vue风格规范 4

1.1 组件名为多个单词 4

1.2 组件数据 4

1.3 Prop 定义 5

1.4 为 v-for 设置键值 6

1.5 避免 v-if 和 v-for 用在一起 7

1.6 为组件样式设置作用域 8

1.7 私有属性名 10

1.8 组件文件 10

1.9 单文件组件文件的大小写 11

1.10 基础组件名 12

1.11 单例组件名 13

1.12 紧密耦合的组件名 13

1.13 组件名中的单词顺序 14

1.14 自闭合组件 15

1.15 组件数据 16

1.16 模板中的组件名大小写 16

1.17 完整单词的组件名 18

1.18 Prop 名大小写 18

1.19 多个特性的元素 19

1.20 模板中简单的表达式 20

1.21 简单的计算属性 21

1.22 带引号的特性值 22

1.23 指令缩写 23

1.24 组件/实例的选项的顺序 24

1.25 元素特性的顺序 24

1.26 组件数据 25

1.27 单文件组件的顶级元素的顺序 25

1.28 没有在 v-if/v-else-if/v-else 中使用 key 26

1.29 scoped 中的元素选择器 27

1.30 隐性的父子组件通信 28

1.31 非 Flux 的全局状态管理 30

第2章 JS代码规范 33

2.1 逻辑错误 33

2.2 最佳实践(避免一些问题) 44

2.3 变量声明 50

2.4 风格指南 54

2.5 ECMAScript 6 80

第3章 附录 A 操作手册 82

3.1 开发环境搭建 82

3.2 编辑器 93

3.3 Vue安装 93

1 创建vue项目 94

2 webpack项目中ESLint的使用 96

# Vue风格规范

## 组件名为多个单词

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：组件名应该始终是多个单词的，根组件 App 除外。**

说明：

这样做可以避免跟现有的以及未来的 HTML 元素相冲突，因为所有的 HTML 元素名称都是单个单词的。

**正例：**

Vue.component(' todo-item', {})

**反例：**

Vue.component('todo', {})

## 组件数据

**自动检测：是**

**自动修复：是**

**检测规则：[no-shared-component-data](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/no-shared-component-data.md)**

**概述：组件的 data 必须是一个函数**。

说明：

当在组件中使用 data 属性的时候 (除了 new Vue 外的任何地方)，它的值必须是返回一个对象的函数。

**正例：**

export default {

data () {

return {

foo: 'bar'

}

}

}

**反例：**

export default {

data: {

foo: 'bar'

}

}

**例外：**

// 在一个 Vue 的根实例上直接使用对象是可以的，

// 因为只存在一个这样的实例。

new Vue({

data: {

foo: 'bar'

}

})

## Prop 定义

**自动检测：是**

**自动修复：否**

**检测规则：[require-prop-types](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/require-prop-types.md)**

**概述：Prop 定义应该尽量详细。**

说明：

在你提交的代码中，prop 的定义应该尽量详细，至少需要指定其类型。

**正例：**

props: {

status: String

}

// 更好的做法！

props: {

status: {

type: String,  
 required: true,  
 validator: function (value) {  
 return [  
 'syncing',  
 'synced',  
 'version-conflict',  
 'error'  
 ].indexOf(value) !== -1  
 }  
 }  
}

**反例：**

props: ['status']

## 为 v-for 设置键值

**自动检测：是**

**自动修复：否**

**检测规则：[require-v-for-key](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/require-v-for-key.md)**

**概述：总是用 key 配合 v-for**。

说明：

在组件上总是必须用 key 配合 v-for，以便维护内部组件及其子树的状态。甚至在元素上维护可预测的行为，比如动画中的对象固化 (object constancy)，也是一种好的做法。

**正例：**

<ul>

<li  
 v-for="todo in todos"  
 :key="todo.id"  
 >  
 {{ todo.text }}  
 </li>  
</ul>

**反例：**

<ul>

<li v-for="todo in todos">  
 {{ todo.text }}  
 </li>  
</ul>

## 避免 v-if 和 v-for 用在一起

**自动检测：是**

**自动修复：否**

**检测规则：[no-use-v-if-with-v-for](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/no-use-v-if-with-v-for.md)**

**概述：永远不要把 v-if 和 v-for 同时用在同一个元素上**。

说明：

当 Vue 处理指令时，v-for 比 v-if 具有更高的优先级, 因此哪怕我们只渲染出一小部分用户的元素，也得在每次重渲染的时候遍历整个列表，不论活跃用户是否发生了变化。

**正例：**

<ul>

<li  
 v-for="user in activeUsers"  
 :key="user.id"  
 >  
 {{ user.name }}  
 </li>  
</ul>

**反例：**

<ul>

<li  
 v-for="user in users"  
 v-if="user.isActive"  
 :key="user.id"  
 >  
 {{ user.name }}  
 </li>  
</ul>

## 为组件样式设置作用域

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：对于应用来说，顶级 App 组件和布局组件中的样式可以是全局的，但是其它所有组件都应该是有作用域的**。

说明：

这条规则只和单文件组件有关。你不一定要使用 scoped 特性。设置作用域也可以通过 CSS Modules，那是一个基于 class 的类似 BEM 的策略，当然你也可以使用其它的库或约定。

**正例：**

<template>

<button class="button button-close">X</button>  
</template>  
  
<!-- 使用 `scoped` 特性 -->  
<style scoped>  
.button {  
 border: none;  
 border-radius: 2px;  
}  
  
.button-close {  
 background-color: red;  
}  
</style>

**反例：**

<template>

<button class="btn btn-close">X</button>  
</template>  
  
<style>  
.btn-close {  
 background-color: red;  
}  
</style>

## 私有属性名

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：私有属性名**。

说明：

在插件、混入等扩展中始终为自定义的私有属性使用 $\_ 前缀。并附带一个命名空间以回避和其它作者的冲突 (比如 $\_yourPluginName\_)。

**正例：**

var myGreatMixin = {

methods: {  
 $\_myGreatMixin\_update: function () {

}  
 }  
}

**反例：**

var myGreatMixin = {

methods: {  
 update: function () {  
 }  
 }  
}

## 组件文件

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：只要有能够拼接文件的构建系统，就把每个组件单独分成文件**。

说明：

当你需要编辑一个组件或查阅一个组件的用法时，可以更快速的找到它。

**正例：**

components/

|- TodoList.vue  
|- TodoItem.vue

**反例：**

Vue.component('TodoList', {

// ...  
})  
Vue.component('TodoItem', {  
 // ...  
})

## 单文件组件文件的大小写

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：单文件组件的文件名应该要么始终是单词大写开头 (PascalCase)，要么始终是横线连接 (kebab-case)**。

说明：

单词大写开头对于代码编辑器的自动补全最为友好，因为这使得我们在 JS(X) 和模板中引用组件的方式尽可能的一致。然而，混用文件命名方式有的时候会导致大小写不敏感的文件系统的问题，这也是横线连接命名同样完全可取的原因。

**正例：**

components/

|- MyComponent.vue

components/  
|- my-component.vue

**反例：**

components/

|- mycomponent.vue

components/

|- myComponent.vue

## 基础组件名

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：应用特定样式和约定的基础组件 (也就是展示类的、无逻辑的或无状态的组件) 应该全部以一个特定的前缀开头，比如 Base、App 或 V**。

说明：

这些组件为你的应用奠定了一致的基础样式和行为但是它们**绝不会**包括全局状态 (比如来自 Vuex store)。

**正例：**

components/

|- AppButton.vue  
|- AppTable.vue  
|- AppIcon.vue

**反例：**

components/

|- MyButton.vue  
|- VueTable.vue  
|- Icon.vue

## 单例组件名

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：只应该拥有单个活跃实例的组件应该以 The 前缀命名，以示其唯一性**。

说明：

这不意味着组件只可用于一个单页面，而是每个页面只使用一次。这些组件永远不接受任何 prop，因为它们是为你的应用定制的，而不是它们在你的应用中的上下文。如果你发现有必要添加 prop，那就表明这实际上是一个可复用的组件，只是目前在每个页面里只使用一次。

**正例：**

components/

|- TheHeading.vue  
|- TheSidebar.vue

**反例：**

components/

|- Heading.vue  
|- MySidebar.vue

## 紧密耦合的组件名

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：和父组件紧密耦合的子组件应该以父组件名作为前缀命名**。

说明：

如果一个组件只在某个父组件的场景下有意义，这层关系应该体现在其名字上。因为编辑器通常会按字母顺序组织文件，所以这样做可以把相关联的文件排在一起。

**正例：**

components/

|- TodoList.vue  
|- TodoListItem.vue  
|- TodoListItemButton.vue

**反例：**

components/

|- TodoList.vue  
|- TodoItem.vue  
|- TodoButton.vue

## 组件名中的单词顺序

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：组件名应该以高级别的 (通常是一般化描述的) 单词开头，以描述性的修饰词结尾**。

说明：

**正例：**

components/

|- SearchButtonClear.vue  
|- SearchButtonRun.vue  
|- SearchInputQuery.vue  
|- SearchInputExcludeGlob.vue  
|- SettingsCheckboxTerms.vue  
|- SettingsCheckboxLaunchOnStartup.vue

**反例：**

components/

|- ClearSearchButton.vue  
|- ExcludeFromSearchInput.vue  
|- LaunchOnStartupCheckbox.vue  
|- RunSearchButton.vue  
|- SearchInput.vue  
|- TermsCheckbox.vue

## 自闭合组件

**自动检测：是**

**自动修复：是**

**检测规则：[html-self-closing](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/html-self-closing.md)**

**概述：在单文件组件、字符串模板和 JSX 中没有内容的组件应该是自闭合的——但在 DOM 模板里永远不要这样做**。

说明：

自闭合组件表示它们不仅没有内容，而且刻意没有内容。其不同之处就好像书上的一页白纸对比贴有“本页有意留白”标签的白纸。而且没有了额外的闭合标签，你的代码也更简洁。

不幸的是，HTML 并不支持自闭合的自定义元素——只有官方的“空”元素。所以上述策略仅适用于进入 DOM 之前 Vue 的模板编译器能够触达的地方，然后再产出符合 DOM 规范的 HTML。

**正例：**

<!-- 在单文件组件、字符串模板和 JSX 中 -->

<MyComponent/>

<!-- 在 DOM 模板中 -->  
<my-component></my-component>

**反例：**

<!-- 在单文件组件、字符串模板和 JSX 中 -->  
<MyComponent></MyComponent>

<!-- 在 DOM 模板中 -->  
<my-component/>

## 组件数据

**自动检测：是**

**自动修复：是否**

**检测规则：[no-shared-component-data](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/require-prop-types.md)**

**概述：组件的 data 必须是一个函数**。

说明：

当在组件中使用 data 属性的时候 (除了 new Vue 外的任何地方)，它的值必须是返回一个对象的函数。

**正例：**

**反例：**

## 模板中的组件名大小写

**自动检测：是**

**自动修复：是**

**检测规则：[name-property-casing](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/name-property-casing.md)**

**概述：对于绝大多数项目来说，在单文件组件和字符串模板中组件名应该总是 PascalCase 的——但是在 DOM 模板中总是 kebab-case 的**。

说明：

PascalCase 相比 kebab-case 有一些优势：

编辑器可以在模板里自动补全组件名，因为 PascalCase 同样适用于 JavaScript。

<MyComponent> 视觉上比 <my-component> 更能够和单个单词的 HTML 元素区别开来，因为前者的不同之处有两个大写字母，后者只有一个横线。

如果你在模板中使用任何非 Vue 的自定义元素，比如一个 Web Component，PascalCase 确保了你的 Vue 组件在视觉上仍然是易识别的。

不幸的是，由于 HTML 是大小写不敏感的，在 DOM 模板中必须仍使用 kebab-case。

还请注意，如果你已经是 kebab-case 的重度用户，那么与 HTML 保持一致的命名约定且在多个项目中保持相同的大小写规则就可能比上述优势更为重要了。在这些情况下，在所有的地方都使用 kebab-case 同样是可以接受的。。

**正例：**

|  |
| --- |
| <!-- 在单文件组件和字符串模板中 -->  <MyComponent/> |

|  |
| --- |
| <!-- 在 DOM 模板中 -->  <my-component></my-component> |

或者

|  |
| --- |
| <!-- 在所有地方 -->  <my-component></my-component> |

**反例：**

|  |
| --- |
| <!-- 在单文件组件和字符串模板中 -->  <mycomponent/> |

|  |
| --- |
| <!-- 在单文件组件和字符串模板中 -->  <myComponent/> |

|  |
| --- |
| <!-- 在 DOM 模板中 -->  <MyComponent></MyComponent> |

## 完整单词的组件名

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：组件名应该倾向于完整单词而不是缩写**。

说明：

编辑器中的自动补全已经让书写长命名的代价非常之低了，而其带来的明确性却是非常宝贵的。不常用的缩写尤其应该避免。

**正例：**

components/

|- StudentDashboardSettings.vue  
|- UserProfileOptions.vue

**反例：**

components/

|- SdSettings.vue  
|- UProfOpts.vue

## Prop 名大小写

**自动检测：是**

**自动修复：是**

**检测规则：[prop-name-casing](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/prop-name-casing.md)**

**概述：在声明 prop 的时候，其命名应该始终使用 camelCase，而在模板和 JSX 中应该始终使用 kebab-case**。

说明：

我们单纯的遵循每个语言的约定。在 JavaScript 中更自然的是 camelCase。而在 HTML 中则是 kebab-case。

**正例：**

props: {

greetingText: String  
}

<WelcomeMessage greeting-text="hi"/>

**反例：**

props: {

'greeting-text': String  
}

<WelcomeMessage greetingText="hi"/>

## 多个特性的元素

**自动检测：是**

**自动修复：是**

**检测规则：[max-attributes-per-line](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/max-attributes-per-line.md)**

**概述：多个特性的元素应该分多行撰写，每个特性一行**。

说明：

在 JavaScript 中，用多行分隔对象的多个属性是很常见的最佳实践，因为这样更易读。模板和 JSX 值得我们做相同的考虑。

**正例：**

<img

src="https://vuejs.org/images/logo.png"  
 alt="Vue Logo"  
>

<MyComponent  
 foo="a"  
 bar="b"  
 baz="c"  
/>

**反例：**

<img src="https://vuejs.org/images/logo.png" alt="Vue Logo">

<MyComponent foo="a" bar="b" baz="c"/>

## 模板中简单的表达式

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：组件模板应该只包含简单的表达式，复杂的表达式则应该重构为计算属性或方法**。

说明：

复杂表达式会让你的模板变得不那么声明式。我们应该尽量描述应该出现的是什么，而非如何计算那个值。而且计算属性和方法使得代码可以重用。

**正例：**

<!-- 在模板中 -->

{{ normalizedFullName }}

// 复杂表达式已经移入一个计算属性  
computed: {  
 normalizedFullName: function () {  
 return this.fullName.split(' ').map(function (word) {  
 return word[0].toUpperCase() + word.slice(1)  
 }).join(' ')  
 }  
}

**反例：**

{{

fullName.split(' ').map(function (word) {  
 return word[0].toUpperCase() + word.slice(1)  
 }).join(' ')  
}}

## 简单的计算属性

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：应该把复杂计算属性分割为尽可能多的更简单的属性**。

说明：

更简单、命名得当的计算属性是这样的:

**易于测试**

当每个计算属性都包含一个非常简单且很少依赖的表达式时，撰写测试以确保其正确工作就会更加容易

**易于阅读**

简化计算属性要求你为每一个值都起一个描述性的名称，即便它不可复用。这使得其他开发者 (以及未来的你) 更容易专注在他们关心的代码上并搞清楚发生了什么。

**更好的“拥抱变化”**

任何能够命名的值都可能用在视图上。举个例子，我们可能打算展示一个信息，告诉用户他们存了多少钱；也可能打算计算税费，但是可能会分开展现，而不是作为总价的一部分。

小的、专注的计算属性减少了信息使用时的假设性限制，所以需求变更时也用不着那么多重构了

。

**正例：**

computed: {

basePrice: function () {  
 return this.manufactureCost / (1 - this.profitMargin)  
 },  
 discount: function () {  
 return this.basePrice \* (this.discountPercent || 0)  
 },  
 finalPrice: function () {  
 return this.basePrice - this.discount  
 }  
}

**反例：**

computed: {

price: function () {  
 var basePrice = this.manufactureCost / (1 - this.profitMargin)  
 return (  
 basePrice -  
 basePrice \* (this.discountPercent || 0)  
 )  
 }  
}

## 带引号的特性值

**自动检测：是**

**自动修复：是**

**检测规则：[html-quotes](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/html-quotes.md)**

**概述：非空 HTML 特性值应该始终带引号 (单引号或双引号，选你 JS 里不用的那个)**。

说明：

在 HTML 中不带空格的特性值是可以没有引号的，但这鼓励了大家在特征值里不写空格，导致可读性变差。

**正例：**

<input type="text">

<AppSidebar :style="{ width: sidebarWidth + 'px' }">

**反例：**

<input type=text>

<AppSidebar :style={width:sidebarWidth+'px'}>

## 指令缩写

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：指令缩写 (用 : 表示 v-bind: 和用 @ 表示 v-on:) 应该要么都用要么都不用**。

说明：

当在组件中使用 data 属性的时候 (除了 new Vue 外的任何地方)，它的值必须是返回一个对象的函数。

**正例：**

<input

:value="newTodoText"  
 :placeholder="newTodoInstructions"  
>

<input  
 @input="onInput"  
 @focus="onFocus"  
>

**反例：**

<input

v-bind:value="newTodoText"  
 :placeholder="newTodoInstructions"  
>

<input  
 v-on:input="onInput"  
 @focus="onFocus"  
>

## 组件/实例的选项的顺序

**自动检测：是**

**自动修复：是**

**检测规则：[order-in-components](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/order-in-components.md)**

**概述：组件/实例的选项应该有统一的顺序**。

说明：

这是我们推荐的组件选项默认顺序。它们被划分为几大类，所以你也能知道从插件里添加的新属性应该放到哪里。

**正例：**

**反例：**

## 元素特性的顺序

**自动检测：是**

**自动修复：是**

**检测规则：[attributes-order](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/attributes-order.md)**

**概述：元素 (包括组件) 的特性应该有统一的顺序**。

说明：

这是我们为组件选项推荐的默认顺序。它们被划分为几大类，所以你也能知道新添加的自定义特性和指令应该放到哪里。

**正例：**

**反例：**

## 组件数据

**自动检测：是**

**自动修复：是否**

**检测规则：[no-shared-component-data](https://github.com/vuejs/eslint-plugin-vue/blob/v4.7.1/docs/rules/require-prop-types.md)**

**概述：组件的 data 必须是一个函数**。

说明：

当在组件中使用 data 属性的时候 (除了 new Vue 外的任何地方)，它的值必须是返回一个对象的函数。

**正例：**

**反例：**

## 单文件组件的顶级元素的顺序

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：组件的 data 必须是一个函数**。

说明：

单文件组件应该总是让 <script>、<template> 和 <style> 标签的顺序保持一致。且 <style> 要放在最后，因为另外两个标签至少要有一个。

**正例：**

<template>...</template>

<script>/\* ... \*/</script>  
<style>/\* ... \*/</style>

**反例：**

<style>/\* ... \*/</style>

<script>/\* ... \*/</script>  
<template>...</template>

## 没有在 v-if/v-else-if/v-else 中使用 key

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：如果一组 v-if + v-else 的元素类型相同，最好使用 key (比如两个 <div> 元素)**。

说明：

默认情况下，Vue 会尽可能高效的更新 DOM。这意味着其在相同类型的元素之间切换时，会修补已存在的元素，而不是将旧的元素移除然后在同一位置添加一个新元素。如果本不相同的元素被识别为相同，则会出现意料之外的副作用。

**正例：**

<div

v-if="error"  
 key="search-status"  
>  
 错误：{{ error }}  
</div>  
<div  
 v-else  
 key="search-results"  
>  
 {{ results }}  
</div>

<p v-if="error">  
 错误：{{ error }}  
</p>  
<div v-else>  
 {{ results }}  
</div>

**反例：**

<div v-if="error">

错误：{{ error }}  
</div>  
<div v-else>  
 {{ results }}  
</div>

## scoped 中的元素选择器

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：元素选择器应该避免在 scoped 中出现**。

说明：

在 scoped 样式中，类选择器比元素选择器更好，因为大量使用元素选择器是很慢的。

**正例：**

<template>

<button class="btn btn-close">X</button>  
</template>  
  
<style scoped>  
.btn-close {  
 background-color: red;  
}  
</style>

**反例：**

<template>

<button>X</button>  
</template>  
  
<style scoped>  
button {  
 background-color: red;  
}  
</style>

## 隐性的父子组件通信

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：应该优先通过 prop 和事件进行父子组件之间的通信，而不是 this.$parent 或改变 prop**。

说明：

一个理想的 Vue 应用是 prop 向下传递，事件向上传递的。遵循这一约定会让你的组件更易于理解。然而，在一些边界情况下 prop 的变更或 this.$parent 能够简化两个深度耦合的组件。

问题在于，这种做法在很多简单的场景下可能会更方便。但请当心，不要为了一时方便 (少写代码) 而牺牲数据流向的简洁性 (易于理解)。

**正例：**

Vue.component('TodoItem', {

props: {  
 todo: {  
 type: Object,  
 required: true  
 }  
 },  
 template: `  
 <input  
 :value="todo.text"  
 @input="$emit('input', $event.target.value)"  
 >  
 `  
})

Vue.component('TodoItem', {  
 props: {  
 todo: {  
 type: Object,  
 required: true  
 }  
 },  
 template: `  
 <span>  
 {{ todo.text }}  
 <button @click="$emit('delete')">  
 X  
 </button>  
 </span>  
 `  
})

**反例：**

Vue.component('TodoItem', {

props: {  
 todo: {  
 type: Object,  
 required: true  
 }  
 },  
 template: '<input v-model="todo.text">'  
})

Vue.component('TodoItem', {  
 props: {  
 todo: {  
 type: Object,  
 required: true  
 }  
 },  
 methods: {  
 removeTodo () {  
 var vm = this  
 vm.$parent.todos = vm.$parent.todos.filter(function (todo) {  
 return todo.id !== vm.todo.id  
 })  
 }  
 },  
 template: `  
 <span>  
 {{ todo.text }}  
 <button @click="removeTodo">  
 X  
 </button>  
 </span>  
 `  
})

## 非 Flux 的全局状态管理

**自动检测：否**

**自动修复：否**

**检测规则：无**

**概述：应该优先通过 Vuex 管理全局状态，而不是通过 this.$root 或一个全局事件总线**。

说明：

通过 this.$root 和/或全局事件总线管理状态在很多简单的情况下都是很方便的，但是并不适用于绝大多数的应用。Vuex 提供的不仅是一个管理状态的中心区域，还是组织、追踪和调试状态变更的好工具。

**正例：**

// store/modules/todos.js

export default {  
 state: {  
 list: []  
 },  
 mutations: {  
 REMOVE\_TODO (state, todoId) {  
 state.list = state.list.filter(todo => todo.id !== todoId)  
 }  
 },  
 actions: {  
 removeTodo ({ commit, state }, todo) {  
 commit('REMOVE\_TODO', todo.id)  
 }  
 }  
}

<!-- TodoItem.vue -->

<template>  
 <span>  
 {{ todo.text }}  
 <button @click="removeTodo(todo)">  
 X  
 </button>  
 </span>  
</template>  
  
<script>  
import { mapActions } from 'vuex'  
  
export default {  
 props: {  
 todo: {  
 type: Object,  
 required: true  
 }  
 },  
 methods: mapActions(['removeTodo'])  
}  
</script>

**反例：**

// main.js

new Vue({  
 data: {  
 todos: []  
 },  
 created: function () {  
 this.$on('remove-todo', this.removeTodo)  
 },  
 methods: {  
 removeTodo: function (todo) {  
 var todoIdToRemove = todo.id  
 this.todos = this.todos.filter(function (todo) {  
 return todo.id !== todoIdToRemove  
 })  
 }  
 }  
})

# JS代码规范

## 逻辑错误

### 不要定义冗余的函数参数

**自动修复：否**

**检测规则：[no-dupe-args](https://eslint.org/docs/rules/no-dupe-args)**

**概述：禁止 function 定义中出现重名参数**。

说明：

**正例：**

**function** foo(a, b, c) {

console.log(a, b, c);

}

**反例：**

**function** foo(a, b, a) {

console.log("value of the second a:", a);

}

### 类中不要定义冗余的属性

**自动修复：是**

**检测规则：[no-dupe-class-members](https://eslint.org/docs/rules/no-dupe-class-members)**

**概述：不允许类成员中有重复的名称**。

说明：如果类成员中有同名的声明，最后一个声明将会默默地覆盖其它声明。 它可能导致意外的行为

**正例：**

**class** Foo {

bar() { }

qux() { }

}

**class** Foo {

**get** bar() { }

**set** bar(value) { }

}

**class** Foo {

**static** bar() { }

bar() { }

}

**反例：**

**class** Foo {

bar() { }

bar() { }

}

**class** Foo {

bar() { }

**get** bar() { }

}

**class** Foo {

**static** bar() { }

**static** bar() { }

}

### 对象中不要定义重复的属性

**自动修复：否**

**检测规则：[no-dupe-keys](https://eslint.org/docs/rules/no-dupe-keys)**

**概述：禁止在对象字面量中出现重复的键**。

说明：如果对象字面量中出现多个属性有同样的键可能会到导致意想不到的情况出现

**正例：**

**var** foo **=** {

bar: "baz",

quxx: "qux"

};

**反例：**

**var** foo **=** {

bar: "baz",

bar: "qux"

};

**var** foo **=** {

"bar": "baz",

bar: "qux"

};

**var** foo **=** {

0x1: "baz",

1: "qux"

};

### 禁止使用稀疏数组

**自动修复：否**

**检测规则：[no-sparse-arrays](https://eslint.org/docs/rules/no-sparse-arrays)**

**概述：禁用稀疏数组**。

说明：稀疏数组包括很多空位置，经常是由于在数组字面量中使用多个逗号造成的

**正例：**

**var** items **=** [];

**var** items **=** **new** Array(23);

**反例：**

**var** items **=** [,];

**var** colors **=** [ "red",, "blue" ];

### 禁止条件表达式中出现赋值操作符

**自动修复：否**

**检测规则：[no-return-assign](https://eslint.org/docs/rules/no-return-assign)**

**概述：禁止在条件语句中出现赋值操作符**。

说明：在条件语句中使用赋值操作符是有效的。然而，很难判断某个特定的赋值是否是有意为之。

**正例：**

**var** x;

**if** (x **===** 0) {

**var** b **=** 1;

}

*// Practical example that wraps the assignment in parentheses*

**function** setHeight(someNode) {

"use strict";

**do** {

someNode.height **=** "100px";

} **while** ((someNode **=** someNode.parentNode));

}

*// Practical example that wraps the assignment and tests for 'null'*

**function** setHeight(someNode) {

"use strict";

**do** {

someNode.height **=** "100px";

} **while** ((someNode **=** someNode.parentNode) **!==** **null**);

}

**反例：**

**var** x;

**if** (x **=** 0) {

**var** b **=** 1;

}

*// Practical example that is similar to an error*

**function** setHeight(someNode) {

"use strict";

**do** {

someNode.height **=** "100px";

} **while** (someNode **=** someNode.parentNode);

}

### 条件中使用常量表达式

**自动修复：否**

**检测规则：[no-constant-condition](https://eslint.org/docs/rules/no-constant-condition)**

**概述：禁止在条件中使用常量表达式**。

说明：

**正例：**

**if** (x **===** 0) {

doSomething();

}

**for** (;;) {

doSomethingForever();

}

**while** (**typeof** x **===** "undefined") {

doSomething();

}

**do** {

doSomething();

} **while** (x);

**var** result **=** x **!==** 0 ? a : b;

**反例：**

**if** (**false**) {

doSomethingUnfinished();

}

**if** (**void** x) {

doSomethingUnfinished();

}

**for** (;**-**2;) {

doSomethingForever();

}

**while** (**typeof** x) {

doSomethingForever();

}

**do** {

doSomethingForever();

} **while** (x **=** **-**1);

**var** result **=** 0 ? a : b;

### 禁止不必要的布尔转换

**自动修复：是**

**检测规则：[no-extra-boolean-cast](https://eslint.org/docs/rules/no-extra-boolean-cast)**

**概述：禁止不必要的布尔类型转换**。

说明：在上下文中比如 if 语句的测试表达式的结果已经被强制转化成了一个布尔值，再通过双重否定（!!）或 Boolean 转化是不必要的

**正例：**

**var** foo **=** **!!**bar;

**var** foo **=** Boolean(bar);

**function** foo() {

**return** **!!**bar;

}

**var** foo **=** bar ? **!!**baz : **!!**bat;

**反例：**

**var** foo **=** **!!!**bar;

**var** foo **=** **!!**bar ? baz : bat;

**var** foo **=** Boolean(**!!**bar);

**var** foo **=** **new** Boolean(**!!**bar);

**if** (**!!**foo) {

*// ...*

}

**if** (Boolean(foo)) {

*// ...*

}

**while** (**!!**foo) {

*// ...*

}

**do** {

*// ...*

} **while** (Boolean(foo));

**for** (; **!!**foo; ) {

*// ...*

}

### 不可达代码

**自动修复：否**

**检测规则：[no-unreachable](https://eslint.org/docs/rules/no-unreachable)**

**概述：禁止在 return、throw、continue 和 break 语句后出现****不可达代码**。

说明：

**正例：**

**function** foo() {

**return** bar();

**function** bar() {

**return** 1;

}

}

**function** bar() {

**return** x;

**var** x;

}

**switch** (foo) {

**case** 1:

**break**;

**var** x;

}

**反例：**

**function** foo() {

**return** **true**;

console.log("done");

}

**function** bar() {

**throw** **new** Error("Oops!");

console.log("done");

}

**while**(value) {

**break**;

console.log("done");

}

**throw** **new** Error("Oops!");

console.log("done");

**function** baz() {

**if** (Math.random() **<** 0.5) {

**return**;

} **else** {

**throw** **new** Error();

}

console.log("done");

}

**for** (;;) {}

console.log("done");

## 最佳实践(避免一些问题)

### 使用 === 和 !==

**自动修复：是**

**检测规则：[eqeqeq](https://eslint.org/docs/rules/eqeqeq)**

**概述：要求使用 === 和 !==**。

说明：

使用类型安全的 === 和 !== 操作符代替 == 和 != 操作符是一个很好的实践。

这样做的原因是，== 和 != 遵循 [Abstract Equality Comparison Algorithm](https://www.ecma-international.org/ecma-262/5.1/" \l "sec-11.9.3) 作强制转型。例如，以下语句被认为是 true。

* [] == false
* [] == ![]
* 3 == "03"

如果它们中的任何一个出现在一个看上去无害的语句中，比如 a == b ，那么实际的问题是很难被发现的。

**正例：**

a **===** b

foo **===** **true**

bananas **!==** 1

value **===** **undefined**

**typeof** foo **===** 'undefined'

'hello' **!==** 'world'

0 **===** 0

**true** **===** **true**

foo **===** **null**

**反例：**

a **==** b

foo **==** **true**

bananas **!=** 1

value **==** **undefined**

**typeof** foo **==** 'undefined'

'hello' **!=** 'world'

0 **==** 0

**true** **==** **true**

foo **==** **null**

### 一致的括号风格

**自动修复：是**

**检测规则：[curly](https://eslint.org/docs/rules/curly)**

**概述：要求遵循大括号约定**。

说明：当代码块只有一条语句时，JavaScript 允许省略大括号。然而，在块区域前后时刻保留大括号是一种最佳实践，即使他们是可有可无的，因为省略大括号会导致错误，并且降低代码的清晰度

**正例：**

**if** (foo) {

foo**++**;

}

**while** (bar) {

baz();

}

**if** (foo) {

baz();

} **else** {

qux();

}

**反例：**

**if** (foo) foo**++**;

**while** (bar)

baz();

**if** (foo) {

baz();

} **else** qux();

### 点操作符

**自动修复：是**

**检测规则：[dot-location](https://eslint.org/docs/rules/dot-location)**

**概述：点号操作符须与属性需在同一行**。

说明：

**正例：**

**var** foo **=** object

.property;

**var** bar **=** object.property;

**反例：**

**var** foo **=** object.

property;

### 禁止扩展原生类型

**自动修复：否**

**检测规则：[no-extend-native](https://eslint.org/docs/rules/no-extend-native)**

**概述：禁止扩展原生对象**。

说明：在 JavaScript 中，你可以扩展任何对象，包括内置或者”原生”对象。有时人们改变这些原生对象的行为，会影响到代码中的其它部分

**正例：**

**反例：**

Object.prototype.a **=** "a";

Object.defineProperty(Array.prototype, "times", { value: 999 });

### 禁止多次声明同一变量

**自动修复：否**

**检测规则：[no-redeclare](https://eslint.org/docs/rules/no-redeclare)**

**概述：禁止重新声明变量**。

说明：在 JavaScript 中，使用 var 可以对同一个变量再次声明。这会使变量实际声明和定义的位置混乱不堪

**正例：**

**var** a **=** 3;

a **=** 10;

**反例：**

**var** a **=** 3;

**var** a **=** 10;

### 禁止在 return 语句中使用赋值语句

**自动修复：否**

**检测规则：[no-return-assign](https://eslint.org/docs/rules/no-return-assign)**

**概述：禁止在返回语句中赋值**。

说明：

**正例：**

**function** doSomething() {

**return** foo **==** bar **+** 2;

}

**function** doSomething() {

**return** foo **===** bar **+** 2;

}

*// return 语句中的赋值必需有括号包裹*

**function** doSomething() {

**return** (foo **=** bar **+** 2);

}

**反例：**

**function** doSomething() {

**return** foo **=** bar **+** 2;

}

**function** doSomething() {

**return** foo **+=** 2;

}

### 禁止自我赋值

**自动修复：否**

**检测规则：[no-self-assign](https://eslint.org/docs/rules/no-self-assign)**

**概述：避免将变量赋值给自己。**。

说明：自身赋值不起任何作用，可能是由于不完整的重构造成的错误。也表明你的工作还没做完

**正例：**

foo **=** bar;

[a, b] **=** [b, a];

*// This pattern is warned by the `no-use-before-define` rule.*

**let** foo **=** foo;

*// The default values have an effect.*

[foo **=** 1] **=** [foo];

**反例：**

foo **=** foo;

[a, b] **=** [a, b];

[a, ...b] **=** [x, ...b];

({a, b} **=** {a, x});

## 变量声明

### 未使用的变量

**自动修复：否**

**检测规则：[no-unused-vars](https://eslint.org/docs/rules/no-unused-vars)**

**概述：禁止定义未使用的变量**。

说明：已声明的变量在代码里未被使用过，就像是由于不完整的重构而导致的遗漏错误。这样的变量增加了代码量，并且混淆读者

**正例：**

**var** x **=** 10;

alert(x);

*// foo 在这里被视为已使用*

myFunc(**function** foo() {

*// ...*

}.bind(**this**));

(**function**(foo) {

**return** foo;

})();

**var** myFunc;

myFunc **=** setTimeout(**function**() {

*// myFunc 被视为已使用*

myFunc();

}, 50);

*// 只有析构的第二个参数被视为已使用.*

**function** getY([, y]) {

**return** y;

}

**反例：**

**var** x;

*// 只写变量不被视为已使用.*

**var** y **=** 10;

y **=** 5;

*// 对本身修改不被视为已使用.*

**var** z **=** 0;

z **=** z **+** 1;

*// 在默认情况下，未使用的参数也会引发警告.*

(**function**(foo) {

**return** 5;

})();

*// 未使用的递归函数也会引起警告.*

**function** fact(n) {

**if** (n **<** 2) **return** 1;

**return** n **\*** fact(n **-** 1);

}

*// 当函数定义对数组进行析构时，数组中未使用的项也会引发警告.*

**function** getY([x, y]) {

**return** y;

}

### 禁用未声明的变量

**自动修复：否**

**检测规则：[no-undef](https://eslint.org/docs/rules/no-undef)**

**概述：禁用未声明的变量，除非它们在** /\*global \*/ **注释中被提到**。

说明：对任何未声明的变量的引用都会引起一个警告，除非显式地在 /\*global ...\*/  注释中指定，或在 [globals key in the configuration file](https://eslint.org/docs/user-guide/configuring" \l "specifying-globals)  中指定。另一个常见的用例是，你有意使用定义在其他地方的全局变量(例如来自 HTML 的脚本)

**正例：**

*/\*global someFunction b:true\*/*

**var** a **=** someFunction();

b **=** 10;

**反例：**

**var** a **=** someFunction();

b **=** 10;

### 禁止删除变量

**自动修复：否**

**检测规则：[no-delete-var](https://eslint.org/docs/rules/no-delete-var)**

**概述：不要对变量使用 delete 操作。**。

说明：delete的目的是删除对象的属性。使用 delete操作删除一个变量可能会导致意外情况发生。

**正例：**

**反例：**

**var** x;

**delete** x;

**反例：**

## 风格指南

### 禁用 tab

**自动修复：否**

**检测规则：[no-tabs](https://eslint.org/docs/rules/no-tabs)**

**概述：禁用 tab**。

说明：不允许使用 tab 字符，包括在注释内

**正例：**

**var** a **=** 2;

*/\*\**

*\* it's a test function*

*\*/*

**function** test(){}

**var** x **=** 1; *// test*

**反例**

**var** a */\*tab\*/***=** 2;

*/\*\**

*\* /\*tab\*/ it's a test function*

*\*/*

**function** test(){}

**var** x **=** 1; *// /\*tab\*/ test*

### 缩进

**自动修复：是**

**检测规则：[indent](https://eslint.org/docs/rules/indent)**

**概述：使用两个空格进行缩进**。

说明：强制使用一致的缩进

**正例：**

**if**(a) {

b **=** c;

**function** foo(d) {

e **=** f;

}

}

**反例：**

**if**(a) {

b **=** c;

**function** foo(d) {

e **=** f;

}

}

### 禁止空格和 tab 的混合缩进

**自动修复：否**

**检测规则：[no-mixed-spaces-and-tabs](https://eslint.org/docs/rules/no-mixed-spaces-and-tabs)**

**概述：禁止使用 空格 和 tab 混合缩进**。

说明：

**正例：**

**反例**

### 禁止使用多个空格

**自动修复：是**

**检测规则：[no-multi-spaces](https://eslint.org/docs/rules/no-multi-spaces)**

**概述：禁止出现多个空格**。

说明：在某行中，出现多个空格而且不是用来作缩进的，通常是个错误

**正例：**

**var** a **=** 1;

**if**(foo **===** "bar") {}

a **<<** b

**var** arr **=** [1, 2];

a ? b: c

**反例**

**var** a **=** 1;

**if**(foo **===** "bar") {}

a **<<** b

**var** arr **=** [1, 2];

a ? b: c

### 禁用行尾空格

**自动修复：是**

**检测规则：[no-trailing-spaces](https://eslint.org/docs/rules/no-trailing-spaces)**

**概述：禁用行尾空白**。

说明：

该规则禁止使用行尾空白（空格、tab 和其它 Unicode 空白字符）。

有时在编辑文件的过程中，你可以在行的末尾以额外的空格作为结束。这些空格差异可以被源码控制系统识别出并被标记为差异，给开发人员带来挫败感。虽然这种额外的空格并不会造成功能性的问题

**正例：**

**var** foo **=** 0;

**var** baz **=** 5;

**反例**

**var** foo **=** 0;*//•••••*

**var** baz **=** 5;*//••*

*//•••••*

### 引号

**自动修复：是**

**检测规则：[quotes](https://eslint.org/docs/rules/quotes)**

**概述：除需要转义的情况外，字符串统一使用单引号**。

说明：强制使用一致的反勾号、双引号或单引号

**正例：**

**var** single **=** 'single';

**var** backtick **=** `back${x}tick`;

**反例：**

**var** **double** **=** "double";

**var** unescaped **=** "a string containing 'single' quotes";

### 关键字周围的空格

**自动修复：是**

**检测规则：[keyword-spacing](https://eslint.org/docs/rules/keyword-spacing)**

**概述：强制关键字周围空格的一致性**。

说明：关键字是 JavaScript 的语法元素，比如  function  和  if  。这些标识符对 这门语言具有特殊的意义，所以，在代码编辑器内以不同的颜色显示。作为该语言很重要的一部分，风格指南常有关键字周围应该有空格的约定

**特别说明：** 该规则在项目中做了更改

在 .eslintrc.js中添加

**"keyword-spacing"**: **[**"error"**,** **{** **"overrides"**: **{**

**"if"**: **{ "after"**: false **},**

**"for"**: **{ "after"**: false **},**

**"while"**: **{ "after"**: false **}**

**}}]**

**正例：**

**if**(foo) {

*//...*

} **else** **if**(bar) {

*//...*

} **else** {

*//...*

}

**for**(;;);

**while**(**true**) {

*//...*

}

**反例：**

**if** (foo) {

*//...*

}**else** **if** (bar) {

*//...*

}**else** {

*//...*

}

**for** (;;){

}

**while** (**true**){

}

### function左括号前的空格

**自动修复：是**

**检测规则：[space-before-function-paren](https://eslint.org/docs/rules/space-before-function-paren)**

**概述：强制在 function的左括号之前使用一致的空格**。

说明：

* anonymous 针对匿名函数表达式 (比如 function () {})。
* named 针对命名的函数表达式 (比如 function foo () {})。
* asyncArrow 针对异步的箭头函数表达式 (比如 async () => {})。

**特别说明：** 该规则在项目中做了更改

在 .eslintrc.js中添加

**"space-before-function-paren"**: **[**

"error"**,**

**{**

**"anonymous"**: "always"**,**

**"named"**: "never"**,**

**"asyncArrow"**: "always"

**}**

**]**

**正例：**

**function** foo() {

*// ...*

}

**var** bar **=** **function** () {

*// ...*

};

**class** Foo {

**constructor**() {

*// ...*

}

}

**var** foo **=** {

bar() {

*// ...*

}

};

**var** foo **=** **async** (a) **=>** **await** a

**反例：**

**function** foo () {

*// ...*

}

**var** bar **=** **function**() {

*// ...*

};

**class** Foo {

**constructor** () {

*// ...*

}

}

**var** foo **=** {

bar () {

*// ...*

}

};

**var** foo **=** **async**(a) **=>** **await** a

### 操作符周围空格

**自动修复：是**

**检测规则：[space-infix-ops](https://eslint.org/docs/rules/space-infix-ops)**

**概述：要求中缀操作符周围有空格**。

说明：虽然格式化首选项都非常个人化，但大量的风格指南要求运算符周围有空格，这些额外空格的支持者认为它将使代码易于阅读，可以更轻易地突出潜在的错误。

**正例：**

a **+** b

a **+** b

a ? b : c

**const** a **=** {b: 1};

**var** {a **=** 0} **=** bar;

**function** foo(a **=** 0) { }

**反例：**

a**+**b

a**+** b

a **+**b

a?b:c

**const** a**=**{b:1};

**var** {a**=**0}**=**bar;

**function** foo(a**=**0) { }

### 逗号周围的空格

**自动修复：是**

**检测规则：[comma-spacing](https://eslint.org/docs/rules/comma-spacing)**

**概述：强制在逗号周围使用空格**。

说明：逗号前后的空格可以提高列表项的可读性。在逗号之后而不是之前添加一个空格。

**正例：**

**var** foo **=** 1, bar **=** 2

, baz **=** 3;

**var** arr **=** [1, 2];

**var** arr **=** [1,, 3]

**var** obj **=** {"foo": "bar", "baz": "qur"};

foo(a, b);

**new** Foo(a, b);

**function** foo(a, b){}

a, b

**反例：**

**var** foo **=** 1 ,bar **=** 2;

**var** arr **=** [1 , 2];

**var** obj **=** {"foo": "bar" ,"baz": "qur"};

foo(a ,b);

**new** Foo(a ,b);

**function** foo(a ,b){}

a ,b

### 大括号位置要求

**自动修复：是**

**检测规则：[brace-style](https://eslint.org/docs/rules/brace-style)**

**概述：强制在代码块中使用一致的大括号风格**。

说明：在编程过程中，大括号风格与缩进风格紧密联系，用来描述大括号相对控制语句和代码块的位置，少说也有十几种。

在 Javascript 中，one true brace style也是最常见的一种，它将大括号放在控制语句或声明语句同一行的位置

**正例：**

**function** foo() {

**return** **true**;

}

**if** (foo) {

bar();

}

**if** (foo) {

bar();

} **else** {

baz();

}

**try** {

somethingRisky();

} **catch**(e) {

handleError();

}

**反例：**

**function** foo()

{

**return** **true**;

}

**if** (foo)

{

bar();

}

**try**

{

somethingRisky();

} **catch**(e)

{

handleError();

}

**if** (foo) {

bar();

}

**else** {

baz();

}

### 禁止出现多行空行

**自动修复：是**

**检测规则：[no-multiple-empty-lines](https://eslint.org/docs/rules/no-multiple-empty-lines)**

**概述：不允许有连续多行空行**。

说明：空白对于分离代码代码段逻辑是有帮助的，但过量的空白会占用更多的屏幕

**正例：**

**var** foo **=** 5;

**var** bar **=** 3;

**反例：**

**var** foo **=** 5;

**var** bar **=** 3;

### 操作符使用一致的换行符

**自动修复：是**

**检测规则：[operator-linebreak](https://eslint.org/docs/rules/operator-linebreak)**

**概述：强制操作符使用一致的换行符风格**。

说明：

**正例：**

**if** (someCondition **||**

otherCondition) {

}

answer **=** everything ?

42 :

foo;

**反例：**

**if** (someCondition

**||** otherCondition) {

}

answer **=** everything

? 42

: foo;

### 变量声明

**自动修复：否**

**检测规则：[one-var](https://eslint.org/docs/rules/one-var)**

**概述：每个关键字单独声明一个变量**。

说明：

**正例：**

**function** foo() {

**var** bar,

baz;

**let** qux,

norf;

}

**function** foo(){

**const** bar **=** **true**,

baz **=** **false**;

**let** qux,

norf;

}

**function** foo() {

**var** bar,

qux;

**if** (baz) {

qux **=** **true**;

}

}

**function** foo(){

**let** bar;

**if** (baz) {

**let** qux;

}

}

**反例：**

**function** foo() {

**var** bar;

**var** baz;

**let** qux;

**let** norf;

}

**function** foo(){

**const** bar **=** **false**;

**const** baz **=** **true**;

**let** qux;

**let** norf;

}

**function** foo() {

**var** bar;

**if** (baz) {

**var** qux **=** **true**;

}

}

### 单行代码块两边加空格

**自动修复：是**

**检测规则：[block-spacing](https://eslint.org/docs/rules/block-spacing)**

**概述：代码块中开括号前和闭括号后有空格**。

说明：强制在左花括号和同一行上的下一个 token 之间有一个空格。该规则同样强制右花括号和在同一行的前一个 token 之间有一个空格

**正例：**

**function** foo() { **return** **true**; }

**if** (foo) { bar **=** 0; }

**反例：**

**function** foo() {**return** **true**;}

**if** (foo) { bar **=** 0;}

**function** baz() {**let** i **=** 0;

**return** i;

}

### 变量和函数名用驼峰命名

**自动修复：否**

**检测规则：[camelcase](https://eslint.org/docs/rules/camelcase)**

**概述：对于变量和函数名统一使用驼峰命名法**。

说明：

**正例：**

**var** myFavoriteColor **=** "#112C85";

**function** myFunction() {

*// ...*

};

**反例：**

**var** my\_favorite\_color **=** "#112C85";

**function** my\_function() {

*// ...*

}

### 行末逗号

**自动修复：是**

**检测规则：[comma-dangle](https://eslint.org/docs/rules/comma-dangle)**

**概述：不允许有多余的行末逗号**。

说明：

**正例：**

**var** foo **=** {

bar: "baz",

qux: "quux"

};

**var** arr **=** [1,2];

**反例：**

**var** foo **=** {

bar: "baz",

qux: "quux",

};

**var** arr **=** [1,2,];

### 逗号风格

**自动修复：是**

**检测规则：[comma-style](https://eslint.org/docs/rules/comma-style)**

**概述：始终将逗号置于行末**。

说明：

**正例：**

**var** foo **=** 1, bar **=** 2;

**var** foo **=** 1,

bar **=** 2;

**var** foo **=** ["apples",

"oranges"];

**反例：**

**var** foo **=** 1

,

bar **=** 2;

**var** foo **=** 1

, bar **=** 2;

**var** foo **=** ["apples"

, "oranges"];

### 文件末尾留一空行

**自动修复：是**

**检测规则：[eol-last](https://eslint.org/docs/rules/eol-last)**

**概述：文件末尾留一空行**。

说明：在非空文件中存在拖尾换行是一个常见的 UNIX 风格。它的好处同输出文件到终端一样，方便在串联和追加文件时不会打断 shell 的提示

**正例：**

**function** doSmth() {

**var** foo **=** 2;

}\n

**反例：**

**function** doSmth() {

**var** foo **=** 2;

}

### 函数标识和调用之间空格

**自动修复：是**

**检测规则：[func-call-spacing](https://eslint.org/docs/rules/func-call-spacing)**

**概述：禁止在函数标识符和其调用之间有空格**。

说明：

**正例：**

fn();

**反例：**

fn ();

fn

();

### 冒号与值之间用一致的空格

**自动修复：是**

**检测规则：[key-spacing](https://eslint.org/docs/rules/key-spacing)**

**概述：在对象字面量的键和值之间使用一致的空格**。

说明：键值对当中冒号与值之间要留空格，之前不要有空格

**正例：**

**var** obj **=** { "foo": 42 };

**反例：**

**var** obj **=** { "foo" : 42 };

### 禁用 Array 构造函数

**自动修复：否**

**检测规则：[no-array-constructor](https://eslint.org/docs/rules/no-array-constructor)**

**概述：禁止使用 Array 构造函数**。

说明：使用数组字面量而不是构造器

**正例：**

**var** arr **=** [0, 1, 2]

**反例：**

**var** arr **=** Array(0, 1, 2)

### 禁用 Object 的构造函数

**自动修复：否**

**检测规则：[no-new-object](https://eslint.org/docs/rules/no-new-object)**

**概述：禁止使用 Object 构造函数**。

说明：

**正例：**

**var** myObject **=** **new** CustomObject();

**var** myObject **=** {};

**反例：**

**var** myObject **=** **new** Object();

**var** myObject **=** **new** Object;

### 尽量不要使用三元表达式(如果有更好的实现)

**自动修复：是**

**检测规则：[no-unneeded-ternary](https://eslint.org/docs/rules/no-unneeded-ternary)**

**概述：**。

说明：在 JavaScript 中一个常见的错误是使用一个条件表达式在两个 Boolean 值之间进行选择而不是使用！将测试条件转为一个 Boolean 类型，另一个常见的错误是使用单个变量同时作为测试条件和结果。在这种情况下，逻辑或操作符可以实现相同的功能

**正例：**

**var** a **=** x **===** 2 ? "Yes" : "No";

**var** a **=** x **!==** **false**;

**var** a **=** x ? "Yes" : "No";

**var** a **=** x ? y : x;

**var** a **=** x ? x : 1;

**反例：**

**var** a **=** x **===** 2 ? **true** : **false**;

**var** a **=** x ? **true** : **false**;

### 块内填充

**自动修复：是**

**检测规则：[padded-blocks](https://eslint.org/docs/rules/padded-blocks)**

**概述：禁止块内填充**。

说明：代码块中避免多余留白

**正例：**

**if** (a) {

b();

}

**反例**

**if** (a) {

b();

}

### 关于分号

**自动修复：是否**

**检测规则：[semi](https://eslint.org/docs/rules/semi)**

**概述：不要使用分号**。

说明：

**正例：**

**var** name **=** "ESLint"

object.method **=** **function**() {

*// ...*

}

**var** name **=** "ESLint"

;(**function**() {

*// ...*

})()

**反例：**

**var** name **=** "ESLint";

object.method **=** **function**() {

*// ...*

};

### 分号前后的空格

**自动修复：是**

**检测规则：[semi-spacing](https://eslint.org/docs/rules/semi-spacing)**

**概述：遇到分号时空格要后留前不留**。

说明：

**正例：**

**for** (i **=** 0; i **<** 10; i**++**) {}

**for** (;;) {}

**反例**

**for** (i **=** 0 ; i **<** 10 ; i**++**) {}

**for** (i **=** 0;i **<** 10;i**++**) {}

## ECMAScript 6

### 禁止重复模块导入

**自动修复：否**

**检测规则：[no-duplicate-imports](https://eslint.org/docs/rules/no-duplicate-imports)**

**概述：禁止重复导入**。

说明：同一模块有多个导入时一次性写完。

**正例：**

**import** { merge, find } **from** 'module';

**import** something **from** 'another-module';

**反例：**

**import** { merge } **from** 'module';

**import** something **from** 'another-module';

**import** { find } **from** 'module';

### 展开运算符

**自动修复：是**

**检测规则：[rest-spread-spacing](https://eslint.org/docs/rules/rest-spread-spacing)**

**概述：展开运算符与它的表达式间不要留空白**。

说明：

**正例：**

fn(...args)

[...arr, 4, 5, 6]

**let** [a, b, ...arr] **=** [1, 2, 3, 4, 5];

**function** fn(...args) { console.log(args); }

**let** { x, y, ...z } **=** { x: 1, y: 2, a: 3, b: 4 };

**let** n **=** { x, y, ...z };

**反例：**

fn(... args)

[... arr, 4, 5, 6]

**let** [a, b, ... arr] **=** [1, 2, 3, 4, 5];

**function** fn(... args) { console.log(args); }

**let** { x, y, ... z } **=** { x: 1, y: 2, a: 3, b: 4 };

**let** n **=** { x, y, ... z };

附录 A 操作手册

## 开发环境搭建

项目在nodejs下开发，使用npm管理项目依赖包

**Window 上安装Node.js**

你可以采用以下两种方式来安装。

Windows 安装包(.msi)

32 位安装包下载地址 : <https://nodejs.org/dist/v8.12.0/node-v8.12.0-x86.msi>

64 位安装包下载地址 : <https://nodejs.org/dist/v8.12.0/node-v8.12.0-x64.msi>

本文实例以 v0.10.26 版本为例，其他版本类似， 安装步骤：

步骤 1 : 双击下载后的安装包 **v0.10.26**，如下所示：



步骤 2 : 点击以上的Run(运行)，将出现如下界面：



步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :



步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录，并点击 next（下一步）：



步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）



步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步）：



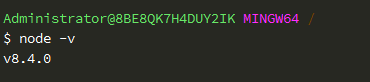
安装过程：



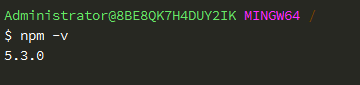
点击 Finish（完成）按钮退出安装向导。



安装完成后，打开命令行工具输入命令node -v，如下图，如果出现对应版本号，就说明安装成功了。



我们所需要的npm包管理器，是集成在node中的，所以，直接输入npm -v就会如下图所示，显示出npm的版本信息。



你可以采用以下两种方式来安装。

Windows 安装包(.msi)

32 位安装包下载地址 : <https://nodejs.org/dist/v8.12.0/node-v8.12.0-x86.msi>

64 位安装包下载地址 : <https://nodejs.org/dist/v8.12.0/node-v8.12.0-x64.msi>

本文实例以 v0.10.26 版本为例，其他版本类似， 安装步骤：

步骤 1 : 双击下载后的安装包 **v0.10.26**，如下所示：



步骤 2 : 点击以上的Run(运行)，将出现如下界面：



步骤 3 : 勾选接受协议选项，点击 next（下一步） 按钮 :



步骤 4 : Node.js默认安装目录为 "C:\Program Files\nodejs\" , 你可以修改目录，并点击 next（下一步）：



步骤 5 : 点击树形图标来选择你需要的安装模式 , 然后点击下一步 next（下一步）



步骤 6 :点击 Install（安装） 开始安装Node.js。你也可以点击 Back（返回）来修改先前的配置。 然后并点击 next（下一步）：



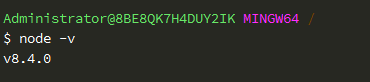
安装过程：



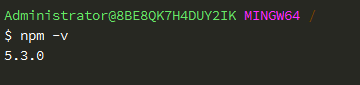
点击 Finish（完成）按钮退出安装向导。



安装完成后，打开命令行工具输入命令node -v，如下图，如果出现对应版本号，就说明安装成功了。



我们所需要的npm包管理器，是集成在node中的，所以，直接输入npm -v就会如下图所示，显示出npm的版本信息。



## 编辑器

### Sublime

#### 插件

1. Expand Tabs on Save // 保存时把tab 转成空格【仍需要在设置中开启】

#### 设置

位置：

英文版Preferences 🡪 Settings 🡪 Preferences.sublime-settings

中文版 首选项 🡪 设置 🡪 Preferences.sublime-settings

1. **"tab\_size"**: 2 // 缩进量
2. **"convert\_tabspaces\_on\_save"**: true // 启用插件[ Expand Tabs on Save ]
3. **"trim\_trailing\_white\_space\_on\_save"**: true // 保存时删除行尾空格
4. **"detect\_indentation"**: false // Set to false to disable detection of tabs vs. spaces on load( 渣翻：设置为false以关闭缩进设置在文件加载时重新检测)
5. **"ensure\_newline\_at\_eof\_on\_save"**: true // 保存时自动在文件末尾插入一行
6. **"translate\_tabs\_to\_spaces"**: true // 设置为true 以启用在按下tab键时使用空格缩进

### 其它编辑器

## Vue安装

在用 vue.js 构建大型应用时推荐使用 npm 安装，npm 能很好地和诸如 webpack 或 browserify 模块打包器配合使用。vue.js 也提供配套工具来开发单文件组件

**安装vue-cli脚手架构建工具**

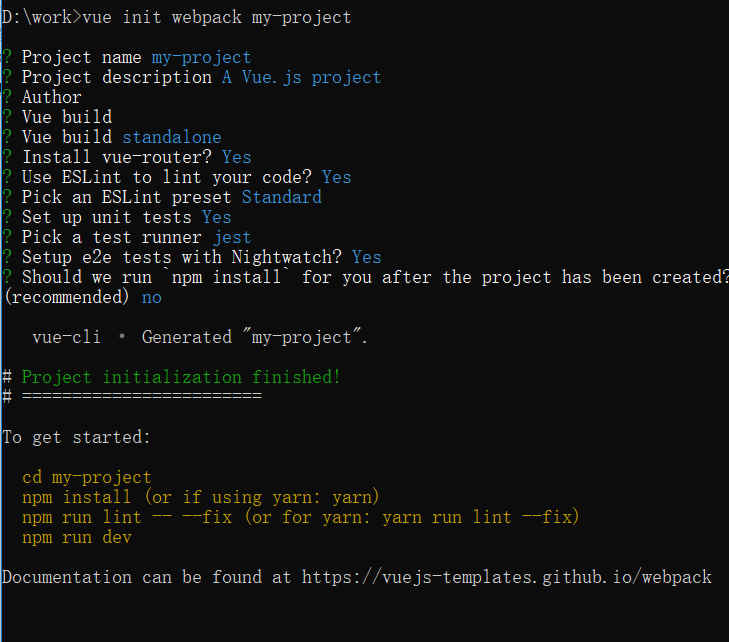
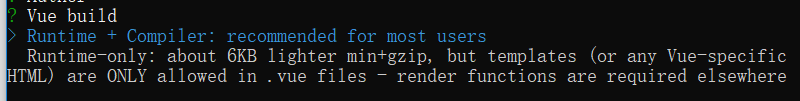
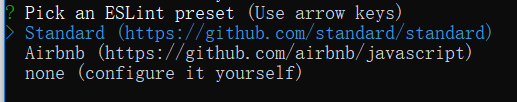
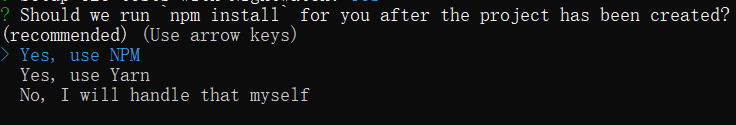
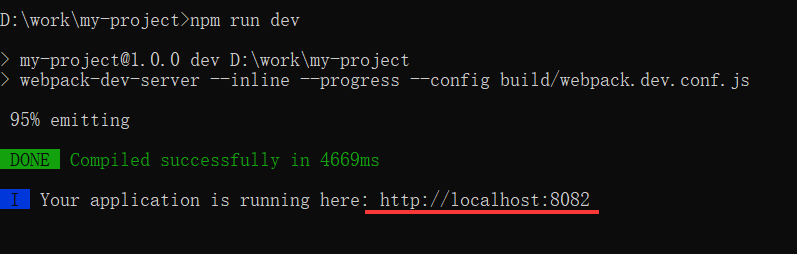
vue-cli 提供一个官方命令行工具，可用于快速搭建大型单页应用。该工具提供开箱即用的构建工具配置，带来现代化的前端开发流程。只需几分钟即可创建并启动一个带热重载、保存时静态检查以及可用于生产环境的构建配置的项目

推荐 全局安装 vue-cli

npm install --global vue-cli

命令行使用vue –-version 查看是否安装成功

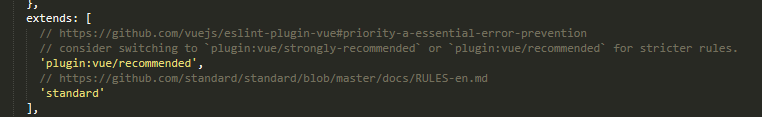
## 创建vue项目

1. 手动构建项目不做更多介绍，参考[Vue CLI guide](https://cli.vuejs.org/zh/guide/)
2. 使用模板构建项目
   * 1. 可用模板 webpack、pwa、webpack-simple、simple、browserify、browserify-simple
     2. 详细介绍见 <https://github.com/vuejs-templates>
3. 创建一个基于 webpack 模板的新项目
   * + 1. 命令行输入
       2. vue init webpack my-project #my-project为自定义项目名
     1. 运行初始化命令的时候回让用户输入几个基本的选项，如项目名称，描述，作者等信息，如果不想填直接回车默认就好（**需要注意的是项目的名称不能大写，不然会报错**）
     2. 
     3. 详细介绍
        + 1. 项目构建方式，上下键选择，enter键选中
          2. 
          3. 是否安装vue-router, 输入y + enter或者直接 enter确认安装
          4. 
          5. 使用ESLint 规范代码
          6. 
          7. 选择ESLint 的预设规则，默认使用Standard
          8. 
          9. 是否在项目创建后自动安装项目依赖包
          10. 
4. 运行项目
   * + 1. npm run dev
       2. 
5. 停止运行 Ctrl+C

## webpack项目中ESLint的使用

1. 环境
   1. 需要NodeJs 环境下使用
2. 配置
   1. 配置文件**:** 项目根目录下的 .eslintrc.js 文件
   2. ESLint 的JavaScript 配置
      1. ESLint 预设规则使用Standard [JavaScript standard 代码规范](https://standardjs.com/rules-zhcn.html)
      2. [查看更多](https://standardjs.com/readme-zhcn.html)
   3. ESLint 的Vue配置
   4. 配置文件**:** 项目根目录下的 .eslintrc.js 文件
   5. 更改规则

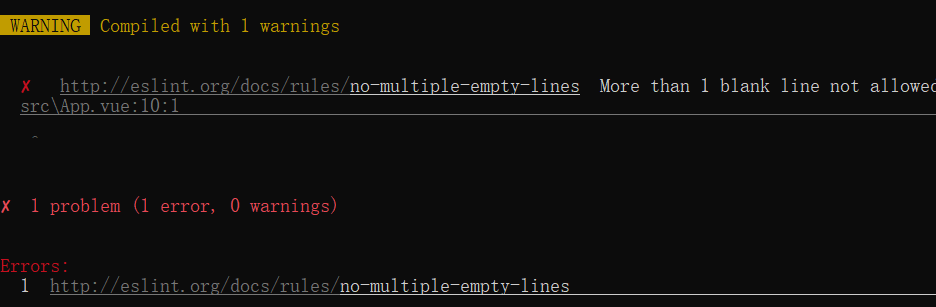
.eslintrc.js 中 extends 增加或更改plugin:vue/recommended



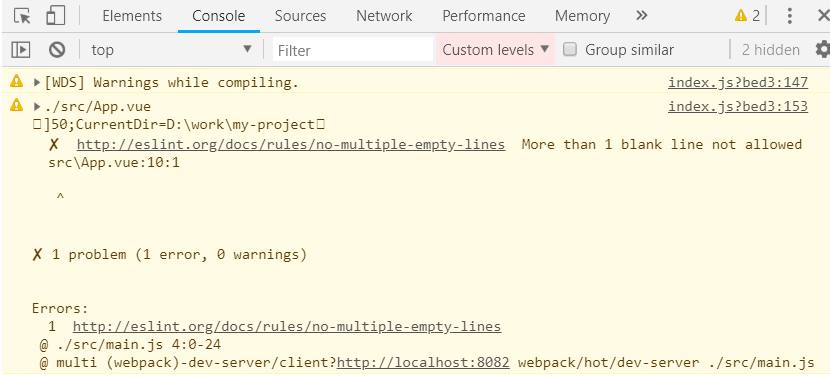
.eslintrc.js 中 parserOptions增加或更改parser: 'babel-eslint'



1. 使用
   1. 查看错误
      1. 命令行中



* + 1. 浏览器控制台



* 1. 更改格式错误
     1. 可使用自动修复的错误

命令行运行 npm run lint -- --fix 自动修复

* + 1. 需要手动修复的错误

## 已有项目添加eslint

* 1. 添加 eslint-plugin-vue 插件
  2. 设置参考 第 2 小节: webpack项目中ESLint的使用
  3. 例外情况

npm run lint -- --fix 运行后,有规范错误提示，但是不能修复

原因：eslint 版本过低

1. 删除 package-lock.json 中 的 eslint 字段



1. 删除 package.json 中 的 eslint 字段



1. 重新安新 eslint 依赖 npm i eslint -D