目录

一、HTML

（一）、语法

1、【推荐】用两个空格来代替制表符（tab） -- 这是唯一能保证在所有环境下获得一致展现的方法

2、【推荐】嵌套元素应当缩进一次（即两个空格）。

3、【强制】对于属性的定义，确保全部使用双引号，绝不要使用单引号。

4、【强制】不要在自闭合（self-closing）元素的尾部添加斜线。

5、【强制】不要省略可选的结束标签（closing tag）

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page title</title>
  </head>
  <body>
    <img src="images/company-logo.png" alt="Company">
    <h1 class="hello-world">Hello, world!</h1>
  </body>
</html>
```

（二）、HTML5 doctype

【强制】为每个 HTML 页面的第一行添加标准模式（standard mode）的声明，这样能够确保在每个浏览器中拥有一致的展现。

```
<!DOCTYPE html>
<html>
  <head>
  </head>
</html>
```

（三）、语言属性

【强制】根据 HTML5 规范：强烈建议为 html 根元素指定 lang 属性，从而为文档设置正确的语言。这将有助于语音合成工具确定其所应该采用的发音，有助于翻译工具确定其翻译时所应遵守的规则等等。

```
<html lang="en-us">
  <!-- ... -->
</html>
```

（四）、IE 兼容模式

【强制】IE 支持通过特定的 <meta> 标签来确定绘制当前页面所应该采用的 IE 版本。除非有强烈的特殊需求，否则最好是设置为 edge mode，从而通知 IE 采用其所支持的最新的模式。

```
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
```

（五）、字符编码

【强制】通过明确声明字符编码，能够确保浏览器快速并容易的判断页面内容的渲染方式。这样做的好处是，可以避免在 HTML 中使用字符实体标记（character entity），从而全部与文档编码一致（一般采用 UTF-8 编码）。

```
<head>
  <meta charset="UTF-8">
</head>
```

（六）、引入 CSS 和 JavaScript 文件

【强制】根据 HTML5 规范，在引入 CSS 和 JavaScript 文件时一般不需要指定 type 属性，因为 text/css 和 text/javascript 分别是它们的默认值。

```
<!-- External CSS -->
<link rel="stylesheet" href="code-guide.css">

<!-- In-document CSS -->
<style>
  /* ... */
</style>

<!-- JavaScript -->
<script src="code-guide.js"></script>
```

（七）、实用为王

【推荐】尽量遵循 HTML 标准和语义，但是不要以牺牲实用性为代价。任何时候都要尽量使用最少的标签并保持最小的复杂度

（八）、属性顺序

【推荐】HTML 属性应当按照以下给出的顺序依次排列，确保代码的易读性。

- `class`
- `id`, `name`
- `data-*`
- `src`, `for`, `type`, `href`, `value`
- `title`, `alt`
- `role`, `aria-*`

class 用于标识高度可复用组件，因此应该排在首位。id 用于标识具体组件，应当谨慎使用（例如，页面内的书签），因此排在第二位。

```
<a class="..." id="..." data-toggle="modal" href="#">
  Example link
</a>

<input class="form-control" type="text">

<img src="..." alt="...">
```

（九）、布尔（boolean）型属性

【强制】布尔型属性可以在声明时不赋值。XHTML 规范要求为其赋值，但是 HTML5 规范不需要。

```
<input type="text" disabled>

<input type="checkbox" value="1" checked>

<select>
  <option value="1" selected>1</option>
</select>
```

（十）、减少标签的数量

【强制】编写 HTML 代码时，尽量避免多余的父元素。

（十一）、JavaScript 生成的标签

【强制】通过 JavaScript 生成的标签让内容变得不易查找、编辑，并且降低性能。能避免时尽量避免。

二、CSS

（一）、语法

1、【强制】用两个空格来代替制表符（tab） -- 这是唯一能保证在所有环境下获得一致展现的方法。

2、【强制】为选择器分组时，将单独的选择器单独放在一行。

3、【强制】为了代码的易读性，在每个声明块的左花括号前添加一个空格。

4、【强制】声明块的右花括号应当单独成行。

5、【强制】每条声明语句的 ： 后应该插入一个空格。

6、【强制】为了获得更准确的错误报告，每条声明都应该独占一行。

7、【强制】所有声明语句都应当以分号结尾。最后一条声明语句后面的分号是可选的，但是，如果省略这个分号，你的代码可能更易出错。

8、【强制】对于以逗号分隔的属性值，每个逗号后面都应该插入一个空格（例如，box-shadow）。

9、【强制】不要在 rgb()、rgba()、hsl()、hsla() 或 rect() 值的内部的逗号后面插入空格。这样利于从多个属性值（既加逗号也加空格）中区分多个颜色值（只加逗号，不加空格）。

10、【强制】对于属性值或颜色参数，省略小于 1 的小数前面的 0 （例如，.5 代替 0.5；-.5px 代替 -0.5px）。

11、【强制】十六进制值应该全部小写，例如，#fff。在扫描文档时，小写字符易于分辨，因为他们的形式更易于区分。

12、【推荐】尽量使用简写形式的十六进制值，例如，用 #fff 代替 #ffffff。

13、【推荐】为选择器中的属性添加双引号，例如，input[type="text"]。只有在某些情况下是可选的，但是，为了代码的一致性，建议都加上双引号。

14、【强制】避免为 0 值指定单位，例如，用 margin: 0; 代替 margin: 0px;。

```
/* Bad CSS */
.selector, .selector-secondary, .selector[type=text] {
  padding:15px;
  margin:0px 0px 15px;
  background-color:rgba(0, 0, 0, 0.5);
  box-shadow:0px 1px 2px #CCC,inset 0 1px 0 #FFFFFF
```

```
}

/* Good CSS */
.selector,
.selector-secondary,
.selector[type="text"] {
  padding: 15px;
  margin-bottom: 15px;
  background-color: rgba(0,0,0,.5);
  box-shadow: 0 1px 2px #ccc, inset 0 1px 0 #fff;
}
```

（二）、声明顺序

【强制】相关的属性声明应当归为一组，并按照下面的顺序排列：

1. Positioning
2. Box model
3. Typographic
4. Visual

由于定位（positioning）可以从正常的文档流中移除元素，并且还能覆盖盒模型（box model）相关的样式，因此排在首位。盒模型排在第二位，因为它决定了组件的尺寸和位置。

其他属性只是影响组件的内部（inside）或者是不影响前两组属性，因此排在后面。

```
.declaration-order {
  /* Positioning */
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  z-index: 100;

  /* Box-model */
  display: block;
  float: right;
  width: 100px;
  height: 100px;

  /* Typography */
  font: normal 13px "Helvetica Neue", sans-serif;
  line-height: 1.5;
  color: #333;
  text-align: center;
```

```
  /* Visual */
  background-color: #f5f5f5;
  border: 1px solid #e5e5e5;
  border-radius: 3px;

  /* Misc */
  opacity: 1;
}
```

（三）、不要使用 @import

【强制】与 <link> 标签相比，@import 指令要慢很多，不光增加了额外的请求次数，还会导致不可预料的问题。替代办法有以下几种：

1、使用多个 <link> 元素

2、通过 Sass 或 Less 类似的 CSS 预处理器将多个 CSS 文件编译为一个文件

3、通过 Webpack 或其他系统中提供过 CSS 文件合并功能

（四）、媒体查询（Media query）的位置

【强制】将媒体查询放在尽可能相关规则的附近。不要将他们打包放在一个单一样式文件中或者放在文档底部。如果你把他们分开了，将来只会被大家遗忘。下面给出一个典型的实例：

```
.element { ... }
.element-avatar { ... }
.element-selected { ... }

@media (min-width: 480px) {
  .element { ...}
  .element-avatar { ... }
  .element-selected { ... }
}
```

（五）、带前缀的属性

【推荐】当使用特定厂商的带有前缀的属性时，通过缩进的方式，让每个属性的值在垂直方向对齐，这样便于多行编辑，一般使用 PostCSS 处理前缀兼容问题。

```
/* Prefixed properties */
.selector {
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.15);
```

```
        box-shadow: 0 1px 2px rgba(0,0,0,.15);
}
```

（六）、单行规则声明

【强制】对于只包含一条声明的样式，为了易读性和便于快速编辑，建议将语句放在同一行。对于带有多条声明的样式，还是应当将声明分为多行。

这样做的关键因素是为了错误检测 -- 例如，CSS 校验器指出在 183 行有语法错误。如果是单行单条声明，你就不会忽略这个错误；如果是单行多条声明的话，你就要仔细分析避免漏掉错误了。

（七）、简写形式的属性声明

【强制】在需要显示地设置所有值的情况下，应当尽量限制使用简写形式的属性声明。常见的滥用简写属性声明的情况如下：

- `padding`
- `margin`
- `font`
- `background`
- `border`
- `border-radius`

大部分情况下，我们不需要为简写形式的属性声明指定所有值。例如，`HTML` 的 `heading` 元素只需要设置上、下边距（`margin`）的值，因此，在必要的时候，只需覆盖这两个值就可以。过度使用简写形式的属性声明会导致代码混乱，并且会对属性值带来不必要的覆盖从而引起意外的副作用。

```
/* Bad example */
.element {
  margin: 0 0 10px;
  background: red;
  background: url("image.jpg");
  border-radius: 3px 3px 0 0;
}

/* Good example */
.element {
  margin-bottom: 10px;
  background-color: red;
  background-image: url("image.jpg");
  border-top-left-radius: 3px;
```

```
  border-top-right-radius: 3px;
}
```

（八）、注释

【推荐】代码是由人编写并维护的。请确保你的代码能够自描述、注释良好并且易于他人理解。好的代码注释能够传达上下文关系和代码目的。不要简单地重申组件或 class 名称。

对于较长的注释，务必书写完整的句子；对于一般性注解，可以书写简洁的短语。

（九）、class 命名

【强制】1、class 名称中只能出现小写字符和破折号（dashe）（不是下划线，也不是驼峰命名法）。破折号应当用于相关 class 的命名（类似于命名空间）（例如，.btn 和 .btn-danger）。

【强制】2、避免过度任意的简写。.btn 代表 button，但是 .s 不能表达任何意思。

【强制】3、class 名称应当尽可能短，并且意义明确。

【强制】4、使用有意义的名称。使用有组织的或目的明确的名称，不要使用表现形式（presentational）的名称。

【强制】5、基于最近的父 class 或基本（base） class 作为新 class 的前缀。

（十）、选择器

【强制】1、对于通用元素使用 class ，这样利于渲染性能的优化。

【强制】2、对于经常出现的组件，避免使用属性选择器（例如，[class^="..."]）。浏览器的性能会受到这些因素的影响。

【强制】3、选择器要尽可能短，并且尽量限制组成选择器的元素个数，建议不要超过 3 。

【强制】4、只有在必要的时候才将 class 限制在最近的父元素内（也就是后代选择器）（例如，不使用带前缀的 class 时 -- 前缀类似于命名空间）

```
/* Bad example */
span { ... }
.page-container #stream .stream-item .tweet .tweet-header .username
{ ... }
.avatar { ... }

/* Good example */
.avatar { ... }
.tweet-header .username { ... }
```

```
.tweet .avatar { ... }
```

（十一）、编辑器配置

【推荐】将你的编辑器按照下面的配置进行设置，以避免常见的代码不一致和差异：

- 用两个空格代替制表符（soft-tab 即用空格代表 tab 符）。
- 保存文件时，删除尾部的空白符。
- 设置文件编码为 UTF-8。
- 在文件结尾添加一个空白行。

四、JavaScript

（一）、【强制】ESLint（Standard 标准）

1. 使用两个空格进行缩进。

```
ESLint: indent
function hello (name) {
  console.log('hi', name)
}
```

2.除需要转义的情况外，字符串统一使用单引号。

```
ESLint: quotes
console.log('hello there')
$("<div class='box'>")
```

3.不要定义未使用的变量。

```
ESLint: no-unused-vars
function myFunction () {
  var result = something()   // ✗ avoid
}
```

4.关键字后面加空格。

```
ESLint: keyword-spacing
if (condition) { ... }   // ✓ ok
if(condition) { ... }   // ✗ avoid
```

5.函数声明时括号与函数名间加空格。

ESLint: space-before-function-paren

function name (arg) { ... } // ✓ ok

function name(arg) { ... } // ✗ avoid

run(function () { ... }) // ✓ ok

run(function() { ... }) // ✗ avoid

6.始终使用 === 替代 ==。

例外： obj == null 可以用来检查 null || undefined。

ESLint: eqeqeq

if (name === 'John') // ✓ ok

if (name == 'John') // ✗ avoid

if (name !== 'John') // ✓ ok

if (name != 'John') // ✗ avoid

7.字符串拼接操作符 (Infix operators) 之间要留空格。

ESLint: space-infix-ops

// ✓ ok

var x = 2

var message = 'hello, ' + name + '!'

// ✗ avoid

var x=2

var message = 'hello, '+name+'!'

8.逗号后面加空格。

ESLint: comma-spacing

// ✓ ok

var list = [1, 2, 3, 4]

function greet (name, options) { ... }

// ✗ avoid

var list = [1,2,3,4]

function greet (name,options) { ... }

9.else 关键字要与花括号保持在同一行。

ESLint: brace-style

// ✓ ok

```
if (condition) {
  // ...
} else {
  // ...
}
// ✗ avoid
if (condition)
{
  // ...
}
else
{
  // ...
}
```

10.多行 if 语句的的括号不能省。

ESLint: curly

```
// ✓ ok
if (options.quiet !== true) console.log('done')
// ✓ ok
if (options.quiet !== true) {
  console.log('done')
}
// ✗ avoid
if (options.quiet !== true)
  console.log('done')
```

11.不要丢掉异常处理中 err 参数。

ESLint: handle-callback-err

```
// ✓ ok
run(function (err) {
  if (err) throw err
  window.alert('done')
})
// ✗ avoid
```

```
run(function (err) {
  window.alert('done')
})
```

12.使用浏览器全局变量时加上 window. 前缀。

例外: document, console and navigator

```
ESLint: no-undef
window.alert('hi')    // ✓ ok
```

13.不允许有连续多行空行。

```
ESLint: no-multiple-empty-lines
// ✓ ok
var value = 'hello world'
console.log(value)
// ✗ avoid
var value = 'hello world'



console.log(value)
```

14.对于三元运算符 ? 和 : 与他们所负责的代码处于同一行。

```
ESLint: operator-linebreak
// ✓ ok
var location = env.development ? 'localhost' : 'www.api.com'

// ✓ ok
var location = env.development
  ? 'localhost'
  : 'www.api.com'
// ✗ avoid
var location = env.development ?
  'localhost' :
  'www.api.com'
```

15.每个 var 关键字单独声明一个变量。

```
ESLint: one-var
// ✓ ok
```

```
var silent = true
var verbose = true
// ✗ avoid
var silent = true, verbose = true


// ✗ avoid
var silent = true,
    verbose = true
```

16.条件语句中赋值语句使用括号包起来。这样使得代码更加清晰可读，而不会认为是将条件判断语句的全等号（===）错写成了等号（=）。

```
ESLint: no-cond-assign
// ✓ ok
while ((m = text.match(expr))) {
  // ...
}


// ✗ avoid
while (m = text.match(expr)) {
  // ...
}
```

17.单行代码块两边加空格。

```
ESLint: block-spacing
function foo () {return true}      // ✗ avoid
function foo () { return true }   // ✓ ok
```

18.对于变量和函数名统一使用驼峰命名法。

```
ESLint: camelcase
function my_function () { }      // ✗ avoid
function myFunction () { }       // ✓ ok


var my_var = 'hello'               // ✗ avoid
var myVar = 'hello'                // ✓ ok
```

19.不允许有多余的行末逗号。

```
ESLint: comma-dangle
```

```
var obj = {
    message: 'hello',    // ✗ avoid
}
```

20.始终将逗号置于行末。

ESLint: comma-style

```
var obj = {
    foo: 'foo'
    ,bar: 'bar'    // ✗ avoid
}

var obj = {
    foo: 'foo',
    bar: 'bar'    // ✓ ok
}
```

21.文件末尾留一空行。

elint: eol-last

22.函数调用时标识符与括号间不留间隔。

ESLint: func-call-spacing

```
console.log ('hello') // ✗ avoid
console.log('hello')  // ✓ ok
```

23.键值对当中冒号与值之间要留空白。

ESLint: key-spacing

```
var obj = { 'key' : 'value' }      // ✗ avoid
var obj = { 'key' :'value' }       // ✗ avoid
var obj = { 'key':'value' }        // ✗ avoid
var obj = { 'key': 'value' }       // ✓ ok
```

24.构造函数要以大写字母开头。

ESLint: new-cap

```
function animal () {}
var dog = new animal()      // ✗ avoid
function Animal () {}
var dog = new Animal()      // ✓ ok
```

25.无参的构造函数调用时要带上括号。

ESLint: new-parens

```
function Animal () {}
var dog = new Animal      // ✗ avoid
var dog = new Animal()   // ✓ ok
```

26.对象中定义了存值器，一定要对应的定义取值器。

ESLint: accessor-pairs

```
var person = {
  set name (value) {      // ✗ avoid
    this.name = value
  }
}
var person = {
  set name (value) {
    this.name = value
  },
  get name () {              // ✓ ok
    return this.name
  }
}
```

27.子类的构造器中一定要调用 super

ESLint: constructor-super

```
class Dog {
  constructor () {
    super()    // ✗ avoid
  }
}

class Dog extends Mammal {
  constructor () {
    super()    // ✓ ok
  }
}
```

28.使用数组字面量而不是构造器。

ESLint: no-array-constructor

```
var nums = new Array(1, 2, 3)     // ✗ avoid
var nums = [1, 2, 3]                    // ✓ ok
```

29.避免使用 arguments.callee 和 arguments.caller。

ESLint: no-caller

```
function foo (n) {
  if (n <= 0) return

  arguments.callee(n - 1)     // ✗ avoid
}

function foo (n) {
  if (n <= 0) return

  foo(n - 1)
}
```

30.避免对类名重新赋值。

ESLint: no-class-assign

```
class Dog {}
Dog = 'Fido'      // ✗ avoid
```

31.避免修改使用 const 声明的变量。

ESLint: no-const-assign

```
const score = 100
score = 125          // ✗ avoid
```

32.避免使用常量作为条件表达式的条件（循环语句除外）。

ESLint: no-constant-condition

```
if (false) {      // ✗ avoid
  // ...
}

if (x === 0) {   // ✓ ok
  // ...
}
```

```
while (true) {   // ✓ ok
  // ...
}
```

33.正则中不要使用控制符。

ESLint: no-control-regex

```
var pattern = /\x1f/      // ✗ avoid
var pattern = /\x20/      // ✓ ok
```

34.不要使用 debugger。

ESLint: no-debugger

```
function sum (a, b) {
  debugger          // ✗ avoid
  return a + b
}
```

35.不要对变量使用 delete 操作。

ESLint: no-delete-var

```
var name
delete name        // ✗ avoid
```

36.不要定义冗余的函数参数。

ESLint: no-dupe-args

```
function sum (a, b, a) {   // ✗ avoid
  // ...
}


function sum (a, b, c) {   // ✓ ok
  // ...
}
```

37.类中不要定义冗余的属性。

ESLint: no-dupe-class-members

```
class Dog {
  bark () {}
  bark () {}      // ✗ avoid
}
```

38.对象字面量中不要定义重复的属性。

ESLint: no-dupe-keys

```
var user = {
  name: 'Jane Doe',
  name: 'John Doe'      // ✗ avoid
}
```

39.switch 语句中不要定义重复的 case 分支。

ESLint: no-duplicate-case

```
switch (id) {
  case 1:
    // ...
  case 1:       // ✗ avoid
}
```

40.同一模块有多个导入时一次性写完。

ESLint: no-duplicate-imports

```
import { myFunc1 } from 'module'
import { myFunc2 } from 'module'              // ✗ avoid

import { myFunc1, myFunc2 } from 'module' // ✓ ok
```

41.正则中不要使用空字符。

ESLint: no-empty-character-class

```
const myRegex = /^abc[]/         // ✗ avoid
const myRegex = /^abc[a-z]/     // ✓ ok
```

42.不要解构空值。

ESLint: no-empty-pattern

```
const { a: {} } = foo             // ✗ avoid
const { a: { b } } = foo         // ✓ ok
```

43.不要使用 eval()。

ESLint: no-eval

```
eval( "var result = user." + propName ) // ✗ avoid
var result = user[propName]                    // ✓ ok
```

44.catch 中不要对错误重新赋值。

ESLint: no-ex-assign

```
try {
  // ...
} catch (e) {
  e = 'new value'                    // ✗ avoid
}

try {
  // ...
} catch (e) {
  const newVal = 'new value'   // ✓ ok
}
```

45.不要扩展原生对象。

```
ESLint: no-extend-native
Object.prototype.age = 21        // ✗ avoid
```

46.避免多余的函数上下文绑定。

```
ESLint: no-extra-bind
const name = function () {
  getName()
}.bind(user)      // ✗ avoid

const name = function () {
  this.getName()
}.bind(user)      // ✓ ok
```

47.避免不必要的布尔转换。

```
ESLint: no-extra-boolean-cast
const result = true
if (!!result) {     // ✗ avoid
  // ...
}

const result = true
if (result) {       // ✓ ok
  // ...
```

```
}
```

48.不要使用多余的括号包裹函数。

ESLint: no-extra-parens

```
const myFunc = (function () { })     // ✗ avoid
const myFunc = function () { }       // ✓ ok
```

49.switch 一定要使用 break 来将条件分支正常中断。

ESLint: no-fallthrough

```
switch (filter) {
  case 1:
    doSomething()      // ✗ avoid
  case 2:
    doSomethingElse()
}

switch (filter) {
  case 1:
    doSomething()
    break              // ✓ ok
  case 2:
    doSomethingElse()
}

switch (filter) {
  case 1:
    doSomething()
    // fallthrough // ✓ ok
  case 2:
    doSomethingElse()
}
```

50.不要省去小数点前面的 0。

ESLint: no-floating-decimal

```
const discount = .5         // ✗ avoid
const discount = 0.5        // ✓ ok
```

51.避免对声明过的函数重新赋值。

ESLint: no-func-assign

```
function myFunc () { }
myFunc = myOtherFunc      // ✗ avoid
```

52.不要对全局只读对象重新赋值。

ESLint: no-global-assign

```
window = {}       // ✗ avoid
```

53.注意隐式的 eval()。

ESLint: no-implied-eval

```
setTimeout("alert('Hello world')")                    // ✗ avoid
setTimeout(function () { alert('Hello world') })      // ✓ ok
```

54.嵌套的代码块中禁止再定义函数。

ESLint: no-inner-declarations

```
if (authenticated) {
  function setAuthUser () {}      // ✗ avoid
}
```

55.不要向 RegExp 构造器传入非法的正则表达式。

ESLint: no-invalid-regexp

```
RegExp('[a-z')      // ✗ avoid
RegExp('[a-z]')     // ✓ ok
```

56.不要使用非法的空白符。

ESLint: no-irregular-whitespace

```
function myFunc () /*<NBSP>*/{}     // ✗ avoid
```

57.禁止使用 iterator。

ESLint: no-iterator

```
Foo.prototype.__iterator__ = function () {}     // ✗ avoid
```

58.外部变量不要与对象属性重名。

ESLint: no-label-var

```
var score = 100
function game () {
  score: 50             // ✗ avoid
}
```

59.不要使用标签语句

ESLint: no-labels

```
label:
  while (true) {
    break label        // ✗ avoid
  }
```

60.不要书写不必要的嵌套代码块。

ESLint: no-lone-blocks

```
function myFunc () {
  {                          // ✗ avoid
    myOtherFunc()
  }
}


function myFunc () {
  myOtherFunc()          // ✓ ok
}
```

61.不要混合使用空格与制表符作为缩进。

ESLint: no-mixed-spaces-and-tabs

62.除了缩进，不要使用多个空格。

ESLint: no-multi-spaces

```
const id =      1234      // ✗ avoid
const id = 1234           // ✓ ok
```

63.不要使用多行字符串。

ESLint: no-multi-str

```
const message = 'Hello \ world'        // ✗ avoid
```

64.new 创建对象实例后需要赋值给变量。

ESLint: no-new

```
new Character()                          // ✗ avoid
const character = new Character()    // ✓ ok
```

65.禁止使用 Function 构造器。

ESLint: no-new-func

```
var sum = new Function('a', 'b', 'return a + b')      // ✗ avoid
```

66.禁止使用 Object 构造器。

ESLint: no-new-object

let config = new Object() // ✗ avoid

67.禁止使用 new require。

ESLint: no-new-require

const myModule = new require('my-module') // ✗ avoid

68.禁止使用 Symbol 构造器。

ESLint: no-new-symbol

const foo = new Symbol('foo') // ✗ avoid

69.禁止使用原始包装器。

ESLint: no-new-wrappers

const message = new String('hello') // ✗ avoid

70.不要将全局对象的属性作为函数调用。

ESLint: no-obj-calls

const math = Math() // ✗ avoid

71.不要使用八进制字面量。

ESLint: no-octal

const num = 042 // ✗ avoid

const num = '042' // ✓ ok

72.字符串字面量中也不要使用八进制转义字符。

ESLint: no-octal-escape

const copyright = 'Copyright \251' // ✗ avoid

73.使用 __dirname 和 __filename 时尽量避免使用字符串拼接。

ESLint: no-path-concat

const pathToFile = __dirname + '/app.js' // ✗ avoid

const pathToFile = path.join(__dirname, 'app.js') // ✓ ok

74.使用 getPrototypeOf 来替代 proto。

ESLint: no-proto

const foo = obj.__proto__ // ✗ avoid

const foo = Object.getPrototypeOf(obj) // ✓ ok

75.不要重复声明变量。

ESLint: no-redeclare

let name = 'John'

let name = 'Jane' // ✗ avoid

```
let name = 'John'
name = 'Jane'                // ✓ ok
```

76.正则中避免使用多个空格。

ESLint: no-regex-spaces

```
const regexp = /test value/     // ✗ avoid

const regexp = /test {3}value/   // ✓ ok
const regexp = /test value/      // ✓ ok
```

77.return 语句中的赋值必需有括号包裹。

ESLint: no-return-assign

```
function sum (a, b) {
  return result = a + b        // ✗ avoid
}

function sum (a, b) {
  return (result = a + b)     // ✓ ok
}
```

78.避免将变量赋值给自己。

ESLint: no-self-assign

```
name = name     // ✗ avoid
```

79.避免将变量与自己进行比较操作。

esint: no-self-compare

```
if (score === score) {}     // ✗ avoid
```

80.避免使用逗号操作符。

ESLint: no-sequences

```
if (doSomething(), !!test) {}     // ✗ avoid
```

81.不要随意更改关键字的值。

ESLint: no-shadow-restricted-names

```
let undefined = 'value'        // ✗ avoid
```

82.禁止使用稀疏数组（Sparse arrays）。

ESLint: no-sparse-arrays

```
let fruits = ['apple',, 'orange']          // ✗ avoid
```

83.不要使用制表符。

ESLint: no-tabs

84.正确使用 ES6 中的字符串模板。

ESLint: no-template-curly-in-string

```
const message = 'Hello ${name}'     // ✗ avoid
const message = `Hello ${name}`     // ✓ ok
```

85.使用 this 前请确保 super() 已调用。

ESLint: no-this-before-super

```
class Dog extends Animal {
  constructor () {
    this.legs = 4        // ✗ avoid
    super()
  }
}
```

86.用 throw 抛错时，抛出 Error 对象而不是字符串。

ESLint: no-throw-literal

```
throw 'error'                    // ✗ avoid
throw new Error('error')      // ✓ ok
```

87.行末不留空格。

ESLint: no-trailing-spaces

88.不要使用 undefined 来初始化变量。

ESLint: no-undef-init

```
let name = undefined      // ✗ avoid


let name
name = 'value'               // ✓ ok
```

89.循环语句中注意更新循环变量。

ESLint: no-unmodified-loop-condition

```
for (let i = 0; i < items.length; j++) {...}      // ✗ avoid
for (let i = 0; i < items.length; i++) {...}      // ✓ ok
```

90.如果有更好的实现，尽量不要使用三元表达式。

ESLint: no-unneeded-ternary

```
let score = val ? val : 0        // ✗ avoid
```

```
let score = val || 0                // ✓ ok
```

91.return，throw，continue 和 break 后不要再跟代码。

```
ESLint: no-unreachable
function doSomething () {
  return true
  console.log('never called')       // ✗ avoid
}
```

92.finally 代码块中不要再改变程序执行流程。

```
ESLint: no-unsafe-finally
try {
  // ...
} catch (e) {
  // ...
} finally {
  return 42         // ✗ avoid
}
```

93.关系运算符的左值不要做取反操作。

```
ESLint: no-unsafe-negation
if (!key in obj) {}           // ✗ avoid
```

94.避免不必要的 .call() 和 .apply()。

```
ESLint: no-useless-call
sum.call(null, 1, 2, 3)     // ✗ avoid
```

95.避免使用不必要的计算值作对象属性。

```
ESLint: no-useless-computed-key
const user = { ['name']: 'John Doe' }     // ✗ avoid
const user = { name: 'John Doe' }          // ✓ ok
```

96.禁止多余的构造器。

```
ESLint: no-useless-constructor
class Car {
  constructor () {          // ✗ avoid
  }
}
```

97.禁止不必要的转义。

ESLint: no-useless-escape

let message = 'Hell\o' // ✗ avoid

98.import, export 和解构操作中，禁止赋值到同名变量。

ESLint: no-useless-rename

import { config as config } from './config' // ✗ avoid

import { config } from './config' // ✓ ok

99.属性前面不要加空格。

ESLint: no-whitespace-before-property

user .name // ✗ avoid

user.name // ✓ ok

100.禁止使用 with。

ESLint: no-with

with (val) {...} // ✗ avoid

101.对象属性换行时注意统一代码风格。

ESLint: object-property-newline

```
const user = {
  name: 'Jane Doe', age: 30,
  username: 'jdoe86'  // ✗ avoid
}

const user = { name: 'Jane Doe', age: 30, username: 'jdoe86' }  // ✓ ok

const user = {
  name: 'Jane Doe',
  age: 30,
  username: 'jdoe86'
}
```

102.代码块中避免多余留白。

ESLint: padded-blocks

```
if (user) {
                                    // ✗ avoid
  const name = getName()
```

```
}


if (user) {
   const name = getName()      // ✓ ok
}
```

103.展开运算符与它的表达式间不要留空白。

```
ESLint: rest-spread-spacing
fn(... args)      // ✗ avoid
fn(...args)       // ✓ ok
```

104.遇到分号时空格要后留前不留。

```
ESLint: semi-spacing
for (let i = 0 ;i < items.length ;i++) {...}      // ✗ avoid
for (let i = 0; i < items.length; i++) {...}      // ✓ ok
```

105.代码块首尾留空格。

```
ESLint: space-before-blocks
if (admin){...}       // ✗ avoid
if (admin) {...}      // ✓ ok
```

106.圆括号间不留空格

```
ESLint: space-in-parens
getName( name )       // ✗ avoid
getName(name)         // ✓ ok
```

107.一元运算符后面跟一个空格。

```
ESLint: space-unary-ops
typeof!admin            // ✗ avoid
typeof !admin           // ✓ ok
```

108.注释首尾留空格。

```
ESLint: spaced-comment
//comment //  ✗ avoid
// comment //  ✓ ok


/*comment*/              // ✗ avoid
/* comment */           // ✓ ok
```

109.模板字符串中变量前后不加空格。

ESLint: template-curly-spacing

const message = `Hello, ${ name }` // ✗ avoid

const message = `Hello, ${name}` // ✓ ok

110.检查 NaN 的正确姿势是使用 isNaN()。

ESLint: use-isnan

if (price === NaN) { } // ✗ avoid

if (isNaN(price)) { } // ✓ ok

111.用合法的字符串跟 typeof 进行比较操作。

ESLint: valid-typeof

typeof name === 'undefimed' // ✗ avoid

typeof name === 'undefined' // ✓ ok

112.自调用匿名函数 (IIFEs) 使用括号包裹。

ESLint: wrap-iife

const getName = function () { }() // ✗ avoid

const getName = (function () { }()) // ✓ ok

const getName = (function () { })() // ✓ ok

113.yield * 中的 * 前后都要有空格。

ESLint: yield-star-spacing

yield* increment() // ✗ avoid

yield * increment() // ✓ ok

114.请书写优雅的条件语句（avoid Yoda conditions）。

ESLint: yoda

if (42 === age) { } // ✗ avoid

if (age === 42) { } // ✓ ok

115.使用分号。

ESLint: semi

window.alert('hi') // ✗ avoid

window.alert('hi'); // ✓ ok

116.不要使用 (, [, or ` 等作为一行的开始。在没有分号的情况下代码压缩后会导致报错，而坚持这一规范则可避免出错。

** ESLint: no-unexpected-multiline **

```
// ✓ ok
;(function () {
  window.alert('ok')
}())

// ✗ avoid
(function () {
  window.alert('ok')
}())
// ✓ ok
;[1, 2, 3].forEach(bar)

// ✗ avoid
[1, 2, 3].forEach(bar)
// ✓ ok
;`hello`.indexOf('o')

// ✗ avoid
`hello`.indexOf('o')
```

相比更加可读易懂的代码，那些看似投巧的写法是不可取的。

譬如：

```
;[1, 2, 3].forEach(bar)
```

建议的写法是：

```
var nums = [1, 2, 3]
nums.forEach(bar)
```

（二）、注释

1、使用 /** ... */ 作为多行注释。包含描述、指定所有参数和返回值的类型和值。

```
// bad
// make() returns a new element
// based on the passed in tag name
```

```
//
// @param {String} tag
// @return {Element} element
function make(tag) {
  // ...stuff...
  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 * @param {String} tag
 * @return {Element} element
 */
function make(tag) {
  // ...stuff...
  return element;
}
```

2、使用 // 作为单行注释。在评论对象上面另起一行使用单行注释。在注释前插入空行。

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this._type || 'no type';
```

```
  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this._type || 'no type';

  return type;
}
```

3、给注释增加 FIXME 或 TODO 的前缀可以帮助其他开发者快速了解这是一个需要复查的问题，或是给需要实现的功能提供一个解决方式。这将有别于常见的注释，因为它们是可操作的。使用 FIXME -- need to figure this out 或者 TODO -- need to implement。

（1） 使用 // FIXME: 标注问题。

```
class Calculator {
  constructor() {
    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

（2）使用 // TODO: 标注问题的解决方式。

```
class Calculator {
  constructor() {
    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

五、版本规约

（一）、git 版本规约

1、【强制】master 分支一般由测试 release 和 hotfix 分支且测试通过后合并代码，任何时候都不能直接修改 master 分支代码。

2、【强制】dev 为开发分支，始终为最新完成以及 bug 修复后的代码。不允许没写完或者 bug 未修复的代码进行上传，导致另一个开发人员拉取代码导致项目报错。一般地开发新功能时 feature 分支都是基于 dev 分支下创建的。

feature 命名；feature/开头的为特性分支，命名规则：feature/user_module、feature/cart_module

3、【强制】当有一组 feature 分支开发完成，首先会合并到 dev 分支，进入提测时，会合并到 release 分支。如果测试过程中若存在 bug 需要修复，则直接由开发者在 release 分支修复并提交。

当测试完成之后，合并 release 分支到 master 和 dev 分支，此时 master 为最新代码，用作上线。

4、【强制】如果线上出现紧急 bug 情况，以 master 为基准，创建一个 hotfix 分支，它命名跟 feature 相似，为 hotfix/开头的修复分支，修复成功后需要合并到 master 和 dev 分支。