JavaScript 编码规范

1 代码风格

1.1 结构

1.1.1 缩进

使用 2 个空格作为缩进

```
// bad
function() {
    const name
}

// bad
function() {
 const name
}

// good
function() {
  const name
}
```

> Google、Twitter、Facebook、Github、Mozilla 以及 npm 上 top 10 下载中有 7 个是 2 个空格的

1.1.2 换行

左花括号 `{` 不要换行。

```
// bad
if (condition)
{
  // ...
}

// good
if (condition) {
  // ...
}
```

运算符换行时，运算符在新行的行首。

```
// 链式调用
target.setPosition(300, 50)
  .moveTo(700, 500)

// 超长的三元运算
const result = thisIsAVeryVeryLongCondition
  ? resultA : resultB
```

不同行为或逻辑的语句集，使用空行隔开，更易阅读。

```javascript
function setStyle(element, property, value) {
  if (element == null) return

  element.style[property] = value
}
```

对象的属性和方法间保留空行

```
// bad
const obj = {
  foo() {
  },
  bar() {
  }
}

// good
const obj = {
  foo() {
  },

  bar() {
  }
}
```

1.1.3 空格

二元运算符两侧必须有一个空格，一元运算符与操作对象之间不允许有空格

```
// bad
const x=y+5

// good
const x = y + 5

// bad
const isValid = !! valid
```

```
// good
const isValid = !!valid
```

`if / for / while / switch / do / try / catch / finally` 关键字后以及 **`else / {`** 之前必须有一个空格。

```
// bad
if(condition){
  // ...
}

// good
if (condition) {
  // ...
}

// bad
while(condition){
  // ...
}

// good
while (condition) {
  // ...
}

// bad
(function () {
})()

// good
(function() {
})()
```

在对象创建时，属性中的 `:` 之后必须有空格

```
// bad
const obj = {
  a:1,
  b:2,
  c:3
}

// good
const obj = {
  a: 1,
  b: 2,
  c: 3
}
```

`import/export` 后面的花括号左右各保留一个空格

```
// bad
import {Person, Relation} from 'zone'

export {name, age}

// good
import { Person, Relation } from 'zone'

export { name, age }
```

`()` 和 **`[]`** 内紧贴括号部分不允许有空格

```
// bad
```

```
callFunc( param1, param2, param3 )
save( this.list[ this.indexes[ i ] ] )
const arr = [ 1, 2, 3 ]
const obj = { name: 'obj' }
needIncreament && ( variable += increament )
if ( num > list.length ) {
}
while ( len-- ) {
}

// good
callFunc(param1, param2, param3)
save(this.list[this.indexes[i]])
const arr = [1, 2, 3]
const obj = {name: 'obj'}
needIncream && (variable += increament)
if (num > list.length) {
}
while (len--) {
}
```

单行注释符 `//` 后跟一个空格

1.1.4 语句

多个声明分开，避免删除时多出的逗号

```
// bad
const hangModules = [],
  missModules = [],
  visited = {}

// good
```

```
const hangModules = []
const missModules = []
const visited = {}
```

将所有的 `const` 和 `let` 分组

```
// bad
let i
const items = getItems()
let dragonball
const goSportsTeam = true
let len

// good
const goSportsTeam = true
const items = getItems()
let dragonball
let i
let length
```

代码块

```
// bad
if (condition)
  callFunc()

// good
if (condition) callFunc()

// good
if (condition) {
  callFunc()
```

```
}

// bad
function() { return false }

// good
function() {
  return false
}

// bad
if (condition) {
  thing1()
  thing2()
}
else {
  thing3()
}

// good
if (condition) {
  thing1()
  thing2()
} else {
  thing3()
}
```

IIFE (即时执行方法) 必须在函数表达式外添加 `(`

```
// bad
const task = function () {
  // ...
}()
```

```
// bad
const task = (function () {
  // ...
}())

// good
const task = (function () {
 // ...
})()
```

1.1.5 分号

行末不使用分号，自执行函数前加分号

```
// bad
(() => {
  // ...
})()

// good
;(() => {
  // ...
})()
```

1.1.6 引号

使用单引号

```
// bad
const name = "Capt. Janeway"
```

```
// good
const name = 'Capt. Janeway'
```

1.2 命名

标识符（变量、常量、函数、属性）：**variableNamesLikeThis**（驼峰式）

> 常量全部大写的方式阅读起来比较困难

构造函数、单例模式：**ClassNamesLikeThis**（帕斯卡式）

专有名词风格保持不变

```
// bad
function insertHtml(element, html) {
  // ...
}

// good
function insertHTML(element, html) {
  // ...
}
```

使用前导下划线 `_` 命名私有属性

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
```

```
this._firstName = 'Panda';
```

> JavaScript 没有私有属性的概念，虽然这样做并不能防止有人滥用，但至少可以阐明意图，这样做是错误的。

严禁滥用下划线，比如局部变量的声明。作用域本身限制了局部变量不可被外部访问。

```
// bad
function doSomething() {
  let _name = 'Panda'
  let _age = 11
}
```

避免使用单字母等无意义的名称，命名应具有描述性

```
// bad
function q() {
  // ...
}

// good
function query() {
  // ...
}
```

不用别名引用 `this`，使用箭头函数，直接使用，更加简洁

```
// bad
function foo() {
  const self = this
  return function () {
```

```
    console.log(self)
  }
}

// bad
function foo() {
  const that = this
  return function () {
    console.log(that)
  }
}

// good
function foo() {
  return () => {
    console.log(this)
  }
}
```

存取方法使用 **`getVal() / setVal()`**，意图更明确

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

布尔值，使用 `isVal` 或 `hasVal`

```
const isReady = false
const hasMoreCommands = function() {
  // ...
  // return Boolean
}
```

1.3 注释

使用 `/** ... */` 作为多行注释，参考JSDoc

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

在代码上一行注释，注释前留一个空行

```
// 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
}
```

使用 `// TODO:` 标注要解决的问题

> 方便自己和他人明确代码存在的问题

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

2 语言特性

2.1 引用

使用 `let` 代替 `var`

> `let` 声明的变量属于块级作用域， `let` 不会带来变量提升（Hoisting）

```
// bad
for (var i = 0; i < 3; i++) {
  // ...
}
console.log(i) // 3

// good
for (let i = 0; i < 3; i++) {
  // ...
}
console.log(i) // i is not defined
```

使用 `const` 声明常量

如果变量不需要重新分配值，全部用 `const` 声明

> `const` 声明后，无法修改引用的值，可避免被重写
>
> `const` 也属于块级作用域
>
> `const` 不会带来变量提升（Hoisting）

多次使用的命名空间，使用对象解构替换，简洁易读

```
// bad
const x = obj.x
const y = obj.y
const a = arr[0]
const b = arr[2]

// good
const {x, y} = obj
const [a, ,b] = arr
```

2.2 类型转换

转换成字符串时，使用 `String` 构造器

```
const reviewScore = 9

// bad
const totalScore = reviewScore + ''
// 实际调用: reviewScore.valueOf()
const obj = {
  valueOf: () => return 1
}
obj + '' // '1' 而不是 '[object Object]'

// bad
const totalScore = reviewScore.toString() // toString 方法可能被改写

// good
const totalScore = String(reviewScore)
```

转换成数字时，使用 `Number` 构造器

```
const inputValue = '4'

// bad
const val = +inputValue

// bad
const val = new Number(inputValue)

// bad
/**
 * JavaScript 位运算完全套用 Java，有三个问题：
 * 1、位操作针对的是整数，小数部分会被舍弃
 * 2、整数不能超过32位
 * 3、JavaScript 的数字都是以双精度浮点数存储的，实际执行还需要先转换为整数
 */
const val = inputValue >> 0

// good
const val = Number(inputValue)
```

解析成整数时，使用 `ParseInt`

> 使用 `parseInt` 时，如果不指定进制，转换的变量以 `0` 开头（比如一些月份和天），ECMAScript 3 会当作 8 进制。

```
const inputValue = '200px'

// bad
const val = parseInt(inputValue)

// good
```

```
const val = parseInt(inputValue, 10)
```

转换成 `boolean` 时，使用 `Boolean` 构造器或者 `!!variable`

```
const age = 0

// bad
const hasAge = new Boolean(age)

// good
const hasAge = Boolean(age)

// good
const hasAge = !!age
```

2.3 条件

使用 `===` 和 `!==` 而不是 `==` 和 `!=`

> 严格判断会检查对象的类型，避免隐式的类型转换

```
0 == false // true
0 == '0' // true
0 === false // false
0 === '0' // false
```

使用简写

```
// bad
if (collection.length > 0) {
```

```
  // ...
}

// good
if (collection.length) {
  // ...
}
```

避免使用 `switch`

`switch` 的方式需要逐条 `case` 判断且匹配的 `case`，如果漏掉 `break`，会执行下一条 `case` （不论是否满足）或 default，直到遇到 `break` 为止。

> 使用字典对象代替，速度更快，同时避免未预料的结果。

```
const cases = {
  alpha: function() {
    // ...
  },
  beta: function() {
    // ...
  },
  _default: function() {
    // ...
  }
}
```

2.4 循环

循环体不要包含函数表达式，事先将函数提取到循环体外，避免多次声明函数对象。

```
// bad
for (let i = 0, len = elements.length; i < len; i++) {
  const element = elements[i]
  addListener(element, 'click', function () {})
}

// good
function clicker() {
  // ...
}

for (let i = 0, len = elements.length; i < len; i++) {
  const element = elements[i]
  addListener(element, 'click', clicker)
}
```

对有序集合进行遍历时，缓存 `length`

> 虽然现代浏览器都对数组长度进行了缓存，但对于一些宿主对象和老旧浏览器的数组对象，在每次 `length` 访问时会动态计算元素个数，此时缓存 `length` 能有效提高程序性能。

```
// bad
for (let i = 0; i < arr.length; i++) {
  console.log(i)
}

// good
for (let i = 0, len = arr.length; i < len; i++) {
  console.log(i)
}
```

使用倒序遍历（不考虑先后顺序的情况下）

```
for (let i = elements.length; i--; ) {
  const element = elements[i]
}
```

> 无需额外变量缓存集合长度
>
> 前测条件只需判断数字是否为 true，无需大小比较
>
> 无需运行后执行体
>
> 避免了数组越界

遍历对象若无需获取原型链的属性，用 **`Object.keys`** 和 **`for`** 代替 **`for...in`**

```
for (let keys = Object.keys(obj), i = keys.length; i--; ) {
  const key = keys[i]
  // ...
}
```

> `for in` 的速度很慢
>
> 无需遍历原型链的可枚举属性
>
> 无需 `hasOwnProperty` 判断来避免遍历原型链（ `Object.create(null)` 除外 ）

10万个属性的对象两种遍历方式在 Chrome48 测试结果：

`for in`：143ms `Object.keys` + `for`：45ms

2.5 字符串

字符串模版

```
const name = 'lucy'

// bad
const greetings = 'Hello ' + name

// good
const greetings = `Hello ${name}`
```

换行的字符串

const html = `<article> <h1>Title here</h1> <p>This is a paragraph</p> <footer>Complete</footer> </article>`

```
### 2.6 数组

##### 使用数组字面量 `[]` 创建新数组，除非想要创建的是指定长度的数组

```javascript
// bad
const items = new Array()

// good
const items = []
```

使用扩展运算符（**spread operator**）`...` 复制数组

```
// bad
```
```

```
const list = [1,2,3]
const result = list.concat()

// good
const list = [1,2,3]
const result = [...list]
```

转换类数组（**array-like object**）成数组时，使用 `Array.from`

```
const foo = document.querySelectorAll('.foo')

// bad
const nodes = Array.prototype.slice.call(foo, 0)

// good
const nodes = Array.from(foo)
```

2.7 对象

使用对象字面量 `{}` 创建新 `Object`

```
// bad
const item = new Object()

// good
const item = {}
```

对象的属性不使用关键字/保留字，只对无效属性的属性名添加引号

```
// bad
```

```
const superman = {
  default: { clark: 'kent' },
  'bar': true,
  'data-blah': 5
}

// good
const superman = {
  defaults: { clark: 'kent' },
  bar: true,
  'data-blah': 5
}
```

不允许修改和扩展任何原生对象和宿主对象的原型，避免干扰他人使用

```
// bad
JSON.stringify = function() {
  // ...
}
```

> 如果必须重写内置方法，功能上保持一致性

遍历这块前面的循环部分提过

动态属性名使用属性名表达式

> 所有属性可在对象创建的时候一次性定义

```
// bad
const prop = condition ? 'testA' : 'testB'
const item = {
  test: 0
```

```
}
item[prop] = true

// good
const prop = condition ? 'testA' : 'testB'
const item = {
  test: 0,
  [prop]: true
}
```

对象方法省略 `function`

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  }
}

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  }
}
```

属性简写

```
const lukeSkywalker = 'Luke Skywalker'

// bad
const obj = {
  lukeSkywalker: lukeSkywalker
}

// good
const obj = {
  lukeSkywalker
}
```

简写的属性放在前面

> 很容易识别哪些是简写属性

```
const anakinSkywalker = 'Anakin Skywalker'
const lukeSkywalker = 'Luke Skywalker'

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker
}

// good
const obj = {
  lukeSkywalker,
```

```
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4
}
```

2.8 函数

不在循环体内声明函数，前面的循环提过

不用使用 `arguments`，可以选择扩展运算符 `...` 替代（rest 参数）

> 使用 `...` 能表明你要传入的参数
>
> rest 参数是一个真正的数组，而 `arguments` 是一个类数组

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments)
  return args.join('')
}

// good
function concatenateAll(...args) {
  return args.join('')
}
```

直接给函数的参数指定默认值，不要使用一个变化的函数参数

> 默认值只会在参数未传入或值为 `undefined` 的情况下被使用

```
// bad
function handleThings(opts) {
  opts = opts || {}
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

使用函数表达式（或传递一个匿名函数），使用箭头函数

> 箭头函数体内的 `this` 对象为定义时所在的对象而不是使用时所在的对象且写法更简洁

```
// bad
[1, 2, 3].map(function (x) {
  this.count += x
  return x * x
}, this)

// good
[1, 2, 3].map(x => {
  this.count += x
  return x * x
})
```

箭头函数如果只有一个参数，省略圆括号；如果函数体只有一行返回语句，省略花括号和 **`return`**

```
// bad
[1, 2, 3].map((x) => {
```

```
  return x * x
})

// good
[1, 2, 3].map(x => x * x)
```

2.9 面向对象

总是使用 class, 避免直接操作 `prototype`

> `class` 语法更符合标准面向对象结构，更简洁易读

```
// bad
function Queue(contents = []) {
  this._queue = [...contents]
}

Queue.prototype.pop = function() {
  return this._queue.pop()
}

// good
class Queue {

  constructor(contents = []) {
    this._queue = [...contents]
  }

  pop() {
    return this._queue.pop()
  }
}
```

使用 `extends` 继承

```
// bad
const inherits = require('inherits')

function PeekableQueue(contents) {
  Queue.apply(this, contents)
}

inherits(PeekableQueue, Queue)

PeekableQueue.prototype.peek = function() {
  return this._queue[0]
}

// good
class PeekableQueue extends Queue {
  peek() {
    return this._queue[0]
  }
}
```

2.10 模块

使用 `import` / `export` 而不是其他非标准模块系统

```
// bad
const styleGuide = require('./styleGuide')

// good
import styleGuide from './styleGuide'
```

2.11 动态特性

避免使用直接 `eval` 函数

> 动态代码如果通过其他来源传入 `eval('document.' + potato + '.style.color = "red"')`，可能导致注入攻击
>
> 动态代码调试起来不方便，如具体的行数不明
>
> 动态代码执行更慢（不能编译、缓存）
>
> `eval` 直接调用时，作用域为当前作用域，间接调用时，作用域为全局作用域，可能会造成干扰

```
// 直接调用
function test() {
  let count = 0
  eval('count++')
}

// 相当于
function test() {
  let count = 0
  count++
}

// 间接调用
function test() {
  let count = 0
  const myEval = eval
  myEval('count++')
}
```

```
// 相当于
function test() {
  let count = 0
}
count++
```

如果一定要执行动态代码，使用 `new Function` 执行

> `new Function` 相当于在全局作用域下声明一个函数，不会干扰其他作用域

```
function test() {
  const foo = new Function('name', 'return name')
}

// 相当于
function test() {

}
const foo = function(name) {
  return name
}
```

使用函数代替动态代码

> 动态代码的执行作用域是全局的，会影响全局作用域

```
// bad
setTimeout('a++', 0)
// 与HTML上直接定义事件相同
element.setAttribute('onclick', 'doSomething()')

// good
```

```
setTimeout(function() {
  a++
}, 0)
element.addEventListener('click', function() { ... }, false)
```

尽量不要使用 `with`

> `with` 作用域下如果没找到，会向父级寻找，可能造成未预料的结果

```
// 如果 o 没有属性 x，则会读取参数 x，可能不是想要的结果
function f(x, o) {
  with (o)
    print(x)
}
```

3 浏览器环境

3.1 DOM

尽量减少 DOM 操作

> 使用变量缓存 DOM 对象

```
// bad
document.getElementById('container').setAttribute('class', 'active')
document.getElementById('container').setAttribute('index', 0)

// good
const el = document.getElementById('container')
```

```
el.setAttribute('class', 'active')
el.setAttribute('index', 0)
```

操作 DOM 时，尽量减少页面 reflow

页面 reflow 是非常耗时的行为，非常容易导致性能瓶颈。下面一些场景会触发浏览器的reflow：

- DOM 元素的添加、修改（内容）、删除
- 应用新的样式或者修改任何影响元素布局的属性
- Resize 浏览器窗口、滚动页面
- 读取元素的某些属性，
  如 `offsetLeft`、`offsetTop`、`offsetHeight`、`offsetWidth`、`scrollTop/Left/Width/Height`、`clientTop/Left/Width/Height`、`getComputedStyle()`、`currentStyle` (IE)

```
// bad
el.style.width = '100px'
el.style.height = '100px'
while (i--) {
  el.style.left = el.offsetWidth + 10 + 'px'
}

// good
el.style.cssText = 'width: 100px; height: 100px;'
const offsetWidth = el.offsetWidth
while (i--) {
  el.style.left = offsetWidth + 10 + 'px'
}
```

> 操作 document fragment 是在内存中操作而非 DOM 树下，不会导致 reflow

```
// bad
for (let i = 0; i < 5; i++) {
  const li = document.createElement('li')
  ul.appendChild(li)
}

// good
const docFrag = document.createDocumentFragment()
for (let i = 0; i < 5; i++) {
  const li = document.createElement('li')
  docFrag.appendChild(li)
}
ul.appendChild(docFrag)
```

获取子元素使用 `children`，避免使用 `childNodes`，除非子元素包含文本、注释和属性节点

> `childNodes` 的范围包括 `children`、文本、注释和属性节点

优先使用 `addEventListener / attachEvent` 绑定事件，避免直接在 HTML 属性中或 DOM 的属性绑定事件

> `addEventListener / attachEvent` 可绑定多个事件
>
> 直接在 HTML 属性中或 DOM 的属性绑定事件属于动态代码，在全局作用域下执行

4 附录

4.1 参考

- Airbnb JavaScript Style

4.2 工具

代码规范检查

- ESlint
- JSHint