编程风格

参考 Airbnb 公司的 JavaScript 风格规范。

1. 块级作用域

1.1. let 取代 var

两个新的声明变量的命令：`let` 和 `const` 中，`let` 完全可以取代 `var`，因为两者语义相同，而且 `let` 没有副作用。

```
'use strict';
if (true) {
  let x = 'hello';
}
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

上例如果用 `var` 替代 `let`，实际上就声明了两个全局变量，这显然不是本意。变量应该只在其声明的代码块内有效，`var` 命令做不到这一点，`var` 不存在块级作用域。

`var` 命令存在变量提升效用，`let` 命令没有这个问题。

```
'use strict';
if (true) {
  console.log(x); // ReferenceError
  let x = 'hello';
}
```

上例如果使用 `var` 替代 `let`，`console.log` 那一行就不会报错，而是会输出 `undefined`，因为变量声明提升到代码块的头部。这违反了 **变量先声明后使用** 的原则。建议不再使用 `var` 命令，而是使用 `let` 命令取代。

1.2. 全局常量和线程安全

在 `let` 和 `const` 之间，建议优先使用 `const`，尤其是在全局环境，不应该设置变量，只应设置常量。

`const` 优于 `let` 有几个原因：

> - `const` 可以提醒阅读程序的人，这个变量不应该改变；
> - 防止了无意间修改变量值所导致的错误。
> - `const` 比较符合函数式编程思想，运算不改变值，只是新建值，而且这样也有利于将来的分布式运算；
> - JavaScript 编译器会对 `const` 进行优化，所以多使用 `const`，有利于提高程序的运行效率，也就是说 `let` **和** `const` **的本质区别，其实是编译器内部的处理不同。**

```
// bad
var a = 1, b = 2, c = 3;

// good
const a = 1;
const b = 2;
const c = 3;

// best
const [a, b, c] = [1, 2, 3];
```

所有的函数都应该设置为常量。

2. 字符串

静态字符串一律使用单引号或反引号，不使用双引号。**动态字符串使用反引号。**

```
// bad
const a = "foobar";
const b = 'foo' + a + 'bar';

// acceptable
const c = `foobar`;

// good
const a = 'foobar';
const b = `foo${a}bar`;
```

3. 解构赋值

使用数组成员对变量赋值时，优先使用解构赋值。

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

函数的参数如果是对象的成员，优先使用解构赋值。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
```

```
  const lastName = user.lastName;
}

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
}

// best
function getFullName({ firstName, lastName }) {}
```

如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

```
// bad
function processInput(input) {
  return [left, right, top, bottom];
}

// good
function processInput(input) {
  return { left, right, top, bottom };
}
const { left, right } = processInput(input);
```

4. 对象

单行定义的对象，最后一个成员不以逗号结尾。多行定义的对象，最后一个成员以逗号结尾。

```
// bad
const a = { k1: v1, k2: v2, };
const b = {
  k1: v1,
  k2: v2
};

// good
const a = { k1: v1, k2: v2 }; // 单行，v2 不用逗号结尾
const b = {
  k1: v1,
  k2: v2, // 多行，v2 用逗号结尾
};
```

对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用 `Object.assign` 方法。

```
// bad
const a = {};
a.x = 3;

// if reshape unavoidable
const a = {};
Object.assign(a, { x: 3 });

// good
const a = { x: null };
a.x = 3;
```

如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义。

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

上例中，对象 `obj` 的最后一个属性的属性名，需要计算得到。这时最好采用属性表达式，在新建 `obj` 的时候，将该属性与其他属性定义在一起，**所有属性在一个地方定义。**

对象的属性和方法，尽量采用简洁表达法，这样易于描述和书写。

```
const ref = 'some value';

// bad
const atom = {
  ref: ref, // 属性名就是变量名 'ref'，属性值就是变量 'ref' 的值，可以简写。
  value: 1,
  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  ref,
  value: 1,
```

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

5. 数组

使用扩展运算符（...）拷贝数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

使用 Array.from 方法，将类似数组的对象转为数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

6. 函数

立即执行函数可以写成箭头函数的形式。

```
(() => {
  console.log('Welcome to the Internet.');
})();
```

那些使用匿名函数当作参数的场合，尽量用箭头函数代替。因为这样更简洁，而且绑定了 this。

```
// bad
[1, 2, 3].map(function (x) {
  return x * x;
});

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

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

箭头函数取代 `Function.prototype.bind`，不应再用 `self/_this/that` 绑定 `this`。

```
// bad
const self = this;
const boundMethod = function(...params) {
  return method.apply(self, params);
}

// acceptable
const boundMethod = method.bind(this);

// best
const boundMethod = (...params) => method.apply(this, params);
```

简单的、单行的、不会复用的函数，建议采用箭头函数。如果函数体较为复杂，行数较多，还是应该采用传统的函数写法。

所有配置项都应该集中在一个对象，放在最后一个参数，布尔值最好不要直接作为参数，因为代码语义会很差，也不利于将来增加其他配置项。

```
// bad
function divide(a, b, option = false ) { /* doSomething */ }

// good
function divide(a, b, { option = false } = {}) { /* doSomething */ }
```

不要在函数体内使用 `arguments` 变量，使用 `rest` 运算符（`...`）代替。因为 `rest` 运算符显式表明你想要获取参数，而且 `arguments` 是一个类似数组的对象，而 `rest` 运算符可以提供一个真正的数组。

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

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

使用默认值语法设置函数参数的默认值。

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

// good
function handleThings(opts = {}) { /* doSomething */ }
```

7. Map 结构

只有模拟现实世界的实体对象时，才使用 `Object`。 如果只是需要 `key: value` 的数据结构，使用 `Map` 结构，因为 `Map` 有内建的遍历机制。

```
let map = new Map(arr);

for (let key of map.keys()) {
  console.log(key);
}

for (let value of map.values()) {
  console.log(value);
}

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}

for (let [key, value] of map) {
  console.log(key, value);
}
```

8. Class

总是用 Class，取代需要 prototype 的操作。因为 Class 的写法更简洁，更易于理解。

```
// bad
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// good
class Queue {
  constructor(contents = []) {
```

```
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

使用 `extends` 实现继承，因为这样更简单，不会有破坏 `instanceof` 运算的危险。

```
// 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];
  }
}
```

9. 模块

ES6 模块语法是 JavaScript 模块的标准写法，坚持使用这种写法，取代 Node.js 的 CommonJS 语法。

首先，使用 `import` 取代 `require()`。

```
// CommonJS 的写法
const moduleA = require('moduleA');
const func1 = moduleA.func1;
const func2 = moduleA.func2;

// ES6 的写法
import { func1, func2 } from 'moduleA';
```

其次，使用 `export` 取代 `module.exports`。

```
// commonJS 的写法
var React = require('react');
var Breadcrumbs = React.createClass({
```

```
  render() {
    return <nav />;
  }
});
module.exports = Breadcrumbs;

// ES6 的写法
import React from 'react';
class Breadcrumbs extends React.Component {
  render() {
    return <nav />;
  }
};
export default Breadcrumbs;
```

如果模块只有一个输出值，就使用 `export default`，如果模块有多个输出值，除非其中某个输出值特别重要，否则建议不要使用 `export default`，即多个输出值如果是平等关系，`export default` 与普通的 `export` 就不要同时使用。

如果模块默认输出一个函数，函数名的首字母应该小写，表示这是一个工具方法。

```
function makeStyleGuide() {}
export default makeStyleGuide;
```

如果模块默认输出一个对象，对象名的首字母应该大写，表示这是一个配置值对象。

```
const StyleGuide = {
  es6: {
  }
};
export default StyleGuide;
```

10. ESLint 的使用

ESLint 是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。

首先，在项目的根目录安装 ESLint。

```
npm install --save-dev eslint
```

然后，安装 Airbnb 语法规则，以及 `import`、`a11y`、`react` 插件。

```
npm install --save-dev eslint-config-airbnb
npm install --save-dev eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-
react
```

最后，在项目的根目录下新建一个 `.eslintrc` 文件，配置 ESLint。

```
{
  "extends": "eslint-config-airbnb"
}
```

现在就可以检查，当前项目的代码是否符合预设的规则。

`index.js` 文件的代码如下。

```
var unused = 'I have no purpose!';
function greet() {
    var message = 'Hello, World!';
    console.log(message);
}
greet();
```

使用 ESLint 检查这个文件，就会报出错误。

```
npx eslint index.js

index.js
  1:1  error  Unexpected var, use let or const instead          no-var
  1:5  error  unused is defined but never used                 no-unused-vars
  4:5  error  Expected indentation of 2 characters but found 4  indent
  4:5  error  Unexpected var, use let or const instead          no-var
  5:5  error  Expected indentation of 2 characters but found 4  indent

✖ 5 problems (5 errors, 0 warnings)
```