JavaScript 编码规范

编码风格

Rule1：使用驼峰式命名对象、函数和实例

Rule2：给注释增加 FIXME 或 TODO 的前缀

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。

Rule3：空格使用

- 使用两个空格作为缩进
- 在花括号 `{` 前放一个空格
- 在控制语句（if、while 等）的小括号 `(` 前放一个空格
- 在文件末尾插入一个空行
- 使用空格把运算符隔开
- 在使用长方法链时进行缩进。使用放置在前面的点 `.` 强调这是方法调用而不是新语句
- 在块末和新语句前插入空行

示例：

推荐：

```
function (value) {
  if (true) {
```

```
    $.ajax()
      .success(function(){})
      .error(function(){})
      .complete(function(){});
  }
}

function (foo) {
  const obj = {
    foo() {
    },

    bar() {
    }
  }

  if (foo) {
    return a;
  }

  return baz;
}
```

Rule4：增加结尾的逗号

示例：

不推荐：

```
const hero = {
  firstName: 'Dana',
  lastName: 'Scully'
};

const heroes = [
  'Batman',
  'Superman'
];
```

推荐：

```
const hero = {
  firstName: 'Dana',
  lastName: 'Scully',
};

const heroes = [
  'Batman',
```

```
  'Superman',
];
```

Rule5：字符串使用单引号''

Rule6：过长的字符串避免使用字串连接符号\，而使用+号连接

示例：

不推荐：

```
const errorMessage = 'This is a super long error that was thrown because
of Batman. When you stop to think about how Batman had anything to do wit
h this, you would get nowhere fast.';

const errorMessage = 'This is a super long error that was thrown because
 \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';
```

推荐：

```
const errorMessage = 'This is a super long error that was thrown because
 ' +
  'of Batman. When you stop to think about how Batman had anything to do
 ' +
  'with this, you would get nowhere fast.';
```

Rule7：程序化生成字符串时，使用模板字符串代替字符串连接

模板字符串更为简洁，更具可读性。

示例：

不推荐：

```
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}
```

推荐：

```
function sayHi(name) {
  return 'How are you, ${name}?';
}
```

Rule8：对数字使用 parseInt 转换，要带上类型转换的基数

在一些场景下也可以使用位操作解决性能问题。
但要注意：
小心使用位操作运算符。数字会被当成 64 位值，但是位操作运算符总是返回 32 位的整数。位操作处理大于 32 位的整数值时还会导致意料之外的行为。最大的 32 位整数是 2,147,483,647

示例：

不推荐：

```
const inputValue = '4';

const val = new Number(inputValue);

const val = +inputValue;

const val = parseInt(inputValue);
```

推荐：

```
const val = Number(inputValue);

const val = parseInt(inputValue, 10);

const val = inputValue >> 0;
```

Rule9：不保存`this`的引用，使用箭头函数或 Function#bind

示例：

不推荐：

```
function foo() {
  const _this = this;
  return function() {
    console.log(_this);
  };
}
```

推荐：

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

Rule10：如果一定要使用`this`的引用，使用`_this`来保存

编码规范

1、变量

Rule11：对所有的引用使用 const ，不要使用 var

这能确保你无法对引用重新赋值，也不会导致出现 bug 或难以理解

示例：

不推荐：

```
var a = 1;
var b = 2;
```

推荐：

```
const a = 1;
const b = 2;
```

Rule12：如果一定需要可变动的引用，使用 let 代替 var

因为 let 是块级作用域，而 var 是函数作用域。

示例：

不推荐：

```
var count = 1;

if (true) {
  count += 1;
}
```

推荐：

```
let count = 1;

if (true) {
  count += 1;
}
```

Rule13：避免使用保留字作为属性值。

Rule14：一直使用 const 来声明变量

如果不这样做就会产生全局变量。我们需要避免全局命名空间的污染。

示例：

不推荐：

```
superPower = new SuperPower();
```

推荐：

```
const superPower = new SuperPower();
```

Rule15：使用 const 声明每一个变量

增加新变量将变的更加容易，而且你永远不用再担心调换错；跟 ,。

示例：

不推荐

```
let i, len, dr, gonball,
  items = getItems(),
  goSportsTeam = true;
```

推荐：

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

2、对象

Rule16：使用字面值创建对象

示例：

不推荐：

```
const item = new Object();
```

推荐：

```
const item = {};
```

Rule17：使用对象方法的简写

示例：

不推荐：

```
const atom = {
  value: 1,
  addValue: function (value) {
    return atom.value + value
  },
};
```

推荐：

```
const atom = {
  value: 1,
  addValue(value) {
    return atom.value + value;
  },
};
```

Rule18：使用对象属性值的简写

示例：

不推荐：

```
const worker = 'worker';
```

```
const obj = {
  worker: worker,
};
```

推荐：

```
const obj = {
  worker,
};
```

3、数组

Rule19：使用字面值创建数组

示例：

不推荐：

```
const items = new Array();
```

推荐：

```
const items = [];
```

Rule20：向数组添加元素时使用 Arrary.push 替代直接赋值。

示例：

不推荐：

```
const someStack = [];
someStack[someStack.length] = 'abracadabra';
```

推荐：

```
someStack.push('abracadabra');
```

Rule21：使用 Array.from 把一个类数组对象转换成数组。

示例：

推荐：

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

Rule22：使用拓展运算符 `...` 复制数组

示例：

不推荐：

```
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}
```

推荐：

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

4、解构

Rule23：使用解构存取和使用多属性对象

因为解构能减少临时引用属性。

示例：

不推荐：

```
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;
  return `${firstName} ${lastName}`;
}
```

推荐：

```
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
```

```
}
```

Rule24：对数组使用解构赋值

示例：

不推荐：

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

const first = arr[0];
const second = arr[1];
```

推荐：

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

Rule25：需要回传多个值时，使用对象解构，而不是数组解构

增加属性或者改变排序不会改变调用时的位置。

示例：

不推荐：

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

// 调用时需要考虑回调数据的顺序。
const [left, __, top] = processInput(input);
```

推荐：

```
// 调用时只选择需要的数据
const { left, right } = processInput(input);
```

5、函数

Rule26：使用函数声明代替函数表达式

因为函数声明是可命名的，所以他们在调用栈中更容易被识别。此外，函数声明会把整个函数提升（hoisted），而函数表达式只会把函数的引用变量名提升。这条规则使得箭头函数可以取代函数表

达式。
示例：

不推荐：

```
const foo = function () {};
```

推荐：

```
function foo() {}
```

Rule27：函数表达式的使用

示例：

推荐：

```
/* 立即调用的函数表达式 (IIFE) */
(() => {
   console.log('Welcome to the Internet. Please follow me.');
})();
```

Rule28：永远不要在一个非函数代码块（if、while 等）中声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但它 的解析表现不一致

示例：
不推荐：

```
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}
```

推荐

```
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

Rule29：当你必须使用函数表达式（或传递一个匿名函数）时，使用箭头函数符号

示例：

不推荐：

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

推荐：

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

Rule30：如果一个函数适合用一行写出并且只有一个参数，那就把花括号、圆括号和 return 都省略掉。如果不是，那就不要省略

为什么？语法糖。在链式调用中可读性很高。
为什么不？当你打算回传一个对象的时候。

示例：

推荐：

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

[1, 2, 3].reduce((total, n) => {
  return total + n;
}, 0);
```

Rule31：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

示例：

不推荐：

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

```
function handleThings(opts) {
  if (opts === void 0) {
      opts = {};
  }
}
```

推荐：

```
function handleThings(opts = {}) {}
```

Rule32：总是使用 class。避免直接操作 prototype

示例：

不推荐：

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

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

推荐：

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

Rule33：方法可以返回 this 来帮助链式调用

Rule34：不要使用 iterators，使用高阶函数例如 map() 和 reduce() 替代 for-of

示例：

不推荐：

```
const numbers = [1, 2, 3, 4, 5];

let sum = 0;
for (let num of numbers) {
  sum += num;
}

sum === 15;
```

推荐：

```
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;
```

6、模块

Rule35：总是使用模组 (import/export)而不是其他非标准模块系统

示例：

不推荐：

```
const AirbnbStyleGuide = require('./AirbnbStyleGuide');
module.exports = AirbnbStyleGuide.es6;
```

推荐：

```
import { es6 } from './AirbnbStyleGuide';
export default es6;
```

Rule36：不要从 import 中直接 export

虽然一行代码简洁明了，但让 import 和 export 各司其职让事情能保持一致。

示例：

不推荐：

```
/* filename es6.js */
export { es6 as default } from './script';
```

推荐：

```
import { es6 } from './script';
export default es6;
```

7、表达式

Rule37：优先使用 `===` 和 `!==` 而不是 `==` 和 `!=`

Rule38：条件判断尽量使用简写

示例：

不推荐：

```
if (name !== '') {
  // ...stuff...
}

if (collection.length > 0) {
  // ...stuff...
}
```

推荐：

```
if (name) {
  // ...stuff...
}

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

附录：

表达式计算结果判定规则

- 对象 被计算为 true
- Undefined 被计算为 false
- Null 被计算为 false
- 布尔值 被计算为 布尔的值
- 数字 如果是 +0、-0、或 NaN 被计算为 false, 否则为 true

- 字符串 如果是空字符串 ” 被计算为 false，否则为 true