JavaScript编码规范

# 变量

## 所有的赋值都用const或者let，避免使用var

// 不好

var a = 1;

var b = 2;

// 好

const a = 1;

let b = 2;

b = 3;

原因：var是全局作用域，会污染全局，且有变量提升；const/let是文件或块级作用域，没有变量提升；如果变量不需要重新赋值，则优先使用const

eslint：prefer-const，no-const-assign，no-var

# 对象

## 使用字面量来创建对象，避免new Object()

// 不好

const item = new Object();

// 好

const item = {};

原因：没有本质上的区别，对象字面量的形式更简洁、更方便阅读，可以在一个地方定义多个属性

eslint：no-new-object

## 当创建一个带有动态属性名的对象时，用计算后属性名

function getKey(k) {

return `a key named ${k}`;

}

// 不好

const obj = {

id: 5,

name: 'San Francisco',

};

obj[getKey('enabled')] = true;

// 好

const obj = {

id: 5,

name: 'San Francisco',

[getKey('enabled')]: true,

};

## 用对象方法简写、属性缩写，且简写部分排在前面

const value = 1;

// 不好

const atom = {

value: value,

addValue: function (value) {

return atom.value + value;

},

};

// 好

const atom = {

value,

addValue(value) {

return atom.value + value;

},

name: 'San Francisco',

};

原因：更简洁，哪些是简写形式，一目了然

eslint：object-shorthand

## 只对那些无效的标示使用引号

// 不好

const bad = {

'foo': 3,

'bar': 4,

'data-blah': 5,

};

// 好

const good = {

foo: 3,

bar: 4,

'data-blah': 5,

};

原因：更容易阅读，方便语法高亮，并且方便JS引擎做优化

eslint：quote-props

## 不直接调用Object.prototype上的方法

// 不好

console.log(object.hasOwnProperty(key));

// 好

console.log(Object.prototype.hasOwnProperty.call(object, key));

原因：对象中有自定义的同名属性，此时直接调用就会报错，比如： { hasOwnProperty: false }，或者是一个空对象Object.create(null)

eslint：no-prototype-builtins

## 对象浅拷贝时，推荐使用扩展运算符，而不是Object.assign。获取对象指定的几个属性时，用对象的rest解构运算符

// 非常不好

const original = { a: 1, b: 2 };

const copy = Object.assign(original, { c: 3 }); // 修改了original

delete copy.a; // 修改了original

// 不好

const original = { a: 1, b: 2 };

const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }

// 好

const original = { a: 1, b: 2 };

const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3

const { a, ...noA } = copy; // noA => { b: 2, c: 3 }

# 数组

## 用字面量赋值，避免new Array()

// 不好

const items = new Array();

// 好

const items = [];

原因：1.单参数陷阱，2.Array可能被重定义

例外：给构造函数传入指定的一个数值来创建稀疏数组

eslint：no-array-constructor

## 用Array#push 代替直接向数组中添加一个值

const someStack = [];

// 不好

someStack[someStack.length] = 'abracadabra';

// 好

someStack.push('abracadabra');

## 用扩展运算符做数组浅拷贝，类似上面的对象浅拷贝

// 不好

const len = items.length;

const itemsCopy = [];

let i;

for (i = 0; i < len; i += 1) {

itemsCopy[i] = items[i];

}

// 好

const itemsCopy = [...items];

## 用扩展运算符而不是Array.from来将一个可迭代的对象转换成数组

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

// 好

const nodes = Array.from(foo);

// 更好

const nodes = [...foo];

## 用 Array.from 去将一个类数组对象转成一个数组

const arrLike = { 0: 'foo', 1: 'bar', 2: 'baz', length: 3 };

// 不好

const arr = Array.prototype.slice.call(arrLike);

// 好

const arr = Array.from(arrLike);

## 用 Array.from 而不是扩展运算符去做map遍历。 因为这样可以避免创建一个临时数组。

// 不好

const baz = [...foo].map(bar);

// 好

const baz = Array.from(foo, bar);

## 在数组方法的回调函数中使用 return 语句。 如果函数体由一条返回一个表达式的语句组成， 并且这个表达式没有副作用， 这个时候可以忽略return

// 不好

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

const y = x + 1;

});

// 好

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

const y = x + 1;

return x \* y;

});

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

原因：部分数组方法回调要求有返回值

eslint：array-callback-return

## 如果一个数组有很多行，在数组的 [ 后和 ] 前断行

// 不好

const arr = [

[0, 1], [2, 3], [4, 5],

];

const objectInArray = [{

id: 1,

}, {

id: 2,

}];

const numberInArray = [

1, 2,

];

// 好

const arr = [[0, 1], [2, 3], [4, 5]];

const objectInArray = [

{

id: 1,

},

{

id: 2,

},

];

const numberInArray = [

1,

2,

];

# 解构

## 用解构赋值来获取对象或数组的某个或多个属性值

// 不好

function getFullName(user) {

const firstName = user.firstName;

const lastName = user.lastName;

return `${firstName} ${lastName}`;

}

// 好

function getFullName(user) {

const { firstName, lastName } = user;

return `${firstName} ${lastName}`;

}

// 更好

function getFullName({ firstName, lastName }) {

return `${firstName} ${lastName}`;

}

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

// 不好

const first = arr[0];

const second = arr[1];

// 好

const [first, second] = arr;

原因：新的语法，更简洁

eslint：prefer-destructuring

## 多个返回值用对象的解构，而不是数据解构

// 不好

function processInput(input) {

return [left, right, top, bottom];

}

// 需要考虑返回值顺序

const [left, \_\_, top] = processInput(input);

// 好

function processInput(input) {

return { left, right, top, bottom };

}

// 只需要考虑需要的参数

const { left, top } = processInput(input);

原因：不用考虑返回值的顺序

# 字符串

## 对string用单引号 ''

// 不好

const name = "Capt. Janeway";

// 不好 – 模板字符串需要包含插值或者换行

const name = `Capt. Janeway`;

// 好

const name = 'Capt. Janeway';

原因：风格统一

eslint：quotes

## 超过100个字符的字符串不应该用string串联成多行

// 不好

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.';

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.';

// 好

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.';

原因：不是很友好，且不易被搜索

## 用字符串模板而不是字符串拼接来组织可编程字符串

// 不好

function sayHi(name) {

return 'How are you, ' + name + '?';

}

function sayHi(name) {

return ['How are you, ', name, '?'].join();

}

function sayHi(name) {

return `How are you, ${ name }?`;

}

// 好

function sayHi(name) {

return `How are you, ${name}?`;

}

原因：模板字符串可以提供针对插值和换行更易于阅读，更简明的语法

eslint: prefer-template template-curly-spacing

## 不要对字符串使用eval()

原因：有很多的问题，比如，1.修改词法作用域，2.不方便编译器优化，3.不利于阅读等

eslint: [no-eval](https://eslint.org/docs/rules/no-eval)

## 不要使用无意义的转义字符

// 不好

const foo = '\'this\' \i\s \"quoted\"';

// 好

const foo = '\'this\' is "quoted"';

const foo = `my name is '${name}'`;

原因：影响可读性

eslint: [no-useless-escape](https://eslint.org/docs/rules/no-useless-escape)

# 函数

## 使用命名函数表达式，替代函数声明

// 不好

function foo() {

}

const foo = function () {

};

// 好

// 函数表达式名和声明的函数名是不一样的

const short = function longUniqueMoreDescriptiveLexicalFoo() {

};

原因：函数声明时作用域被提前了，这意味着在一个文件里函数很容易（太容易了）在其定义之前被引用。这样伤害了代码可读性和可维护性。如果你发现一个函数有大又复杂，这个函数妨碍这个文件其他部分的理解性，这可能就是时候把这个函数单独抽成一个模块了。别忘了给表达式显示的命名，不用管这个名字是不是由一个确定的变量推断出来的，这消除了由匿名函数在错误调用栈产生的所有假设，这在现代浏览器和类似babel编译器中很常见

eslint: [func-style](https://eslint.org/docs/rules/func-style)

## 把立即执行函数包裹在圆括号里

// immediately-invoked function expression (IIFE)

(function () {

console.log('Welcome to the Internet. Please follow me.');

}());

原因：一个立即调用的函数表达式是一个单元 - 把它和他的调用者（圆括号）包裹起来，在括号中可以清晰的地表达这些。

eslint: wrap-iife

## 不要在非函数块（if、while等等）内声明函数

// 不好

if (currentUser) {

function test() {

console.log('Nope.');

}

}

// 好

let test;

if (currentUser) {

test = () => {

console.log('Yup.');

};

}

原因：变量提升，作用域

eslint: no-loop-func

## 不要用arguments命名参数

// 不好

function foo(name, options, arguments) {

}

// 好

function foo(name, options, args) {

}

原因：他的优先级高于每个函数作用域自带的 arguments 对象， 这会导致函数自带的 arguments 值被覆盖

## 不要使用arguments，用rest语法...代替

// 不好

function concatenateAll() {

const args = Array.prototype.slice.call(arguments);

return args.join('');

}

// 好

function concatenateAll(...args) {

return args.join('');

}

原因：明确你想用那个参数。而且rest参数是真数组，而不是类似数组的arguments

eslint: prefer-rest-params

## 用默认参数语法而不是在函数里对参数重新赋值

// 不好

function handleThings(opts) {

// 第一， 我们不该改arguments

// 第二： 如果 opts 的值为 false, 它会被赋值为 {}

// 虽然你想这么写， 但是这个会带来一些细微的bug

opts = opts || {};

}

function handleThings(opts) {

if (opts === void 0) {

opts = {};

}

}

// 好

function handleThings(opts = {}) {

}

## 默认参数避免副作用

var b = 1;

// 不好

function count(a = b++) {

console.log(a);

}

count(); // 1

count(); // 2

count(3); // 3

count(); // 3

## 把默认参数赋值放在最后

// 不好

function handleThings(opts = {}, name) {

}

// 好

function handleThings(name, opts = {}) {

}

## 不要用函数构造器创建函数

// 不好

var add = new Function('a', 'b', 'return a + b');

var subtract = Function('a', 'b', 'return a - b');

原因：以这种方式创建函数将类似于字符串 eval()，这会打开漏洞

eslint: no-new-func

## 函数签名部分要有空格

// 不好

const f = function(){};

const g = function (){};

const h = function() {};

// 好

const x = function () {};

const y = function a() {};

原因：统一性好，而且在你添加/删除一个名字的时候不需要添加/删除空格

eslint: space-before-function-paren space-before-blocks

## 不要改参数，不要对参数重新赋值

// 不好

function f1(obj) {

obj.key = 1;

};

function f2(a) {

a = 1;

}

function f3(a) {

if (!a) { a = 1; }

}

// 好

function f4(obj) {

const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key : 1;

};

function f5(a) {

const b = a || 1;

}

function f6(a = 1) {

}

原因：修改参数会产生副作用，参数重新赋值会导致意外行为，尤其是对 arguments。这也会导致优化问题，特别是在V8里

eslint: no-param-reassign

## 用spread操作符去调用可变参数函数更好

// 不好

const x = [1, 2, 3, 4, 5];

console.log.apply(console, x);

new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));

// 好

const x = [1, 2, 3, 4, 5];

console.log(...x);

new Date(...[2016, 8, 5]);

原因：这样更清晰，你不必提供上下文，而且你不能简单的将new和apply进行组合

eslint: prefer-spread

## 调用或者书写一个包含多个参数的函数应该每行值包含一个参数，每行逗号结尾

// 不好

function foo(bar,

baz,

quux) {

// ...

}

console.log(foo,

bar,

baz);

// good

function foo(

bar,

baz,

quux,

) {

// ...

}

console.log(

foo,

bar,

baz,

);

原因：和其他部分风格保持一致

eslint: function-paren-newline

# 箭头函数

## 当你一定要用函数表达式（在回调函数里）的时候就用箭头表达式吧

// 不好

[1, 2, 3].map(function (x) {

const y = x + 1;

return x \* y;

});

// 好

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

const y = x + 1;

return x \* y;

});

原因：箭头函数创建了一个this的当前执行上下文的函数的版本，这通常就是你想要的；而且箭头函数是更简洁的语法

eslint: prefer-arrow-callback, arrow-spacing

## 如果函数体由一个没有副作用的表达式语句组成，删除大括号和return

// 不好

[1, 2, 3].map(number => {

return `A string containing the ${number}.`;

});

// 好

[1, 2, 3].map(number => `A string containing the ${number}.`);

[1, 2, 3].map((number, index) => ({

[index]: number

}));

原因：语法糖，当多个函数链在一起的时候好读

eslint: arrow-parens, arrow-body-style

## 万一表达式涉及多行，把他包裹在圆括号里更可读

// 不好

['get', 'post', 'put'].map(httpMethod => Object.prototype.hasOwnProperty.call(

httpMagicObjectWithAVeryLongName,

httpMethod

)

);

// 好

['get', 'post', 'put'].map(httpMethod => (

Object.prototype.hasOwnProperty.call(

httpMagicObjectWithAVeryLongName,

httpMethod

)

));

原因：这样清晰的显示函数的开始和结束

## 如果你的函数只有一个参数并且函数体没有大括号，就删除圆括号

// 不好

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

[1, 2, 3].map(x => {

const y = x + 1;

return x \* y;

});

// 好

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

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

const y = x + 1;

return x \* y;

});

原因：风格统一

eslint: arrow-parens

## 避免箭头函数(=>)和比较操作符（<=, >=）混淆

// 不好

const itemHeight = item => item.height > 256 ? item.largeSize : item.smallSize;

const itemHeight = (item) => item.height > 256 ? item.largeSize : item.smallSize;

// 好

const itemHeight = item => (item.height > 256 ? item.largeSize : item.smallSize);

const itemHeight = (item) => {

const { height, largeSize, smallSize } = item;

return height > 256 ? largeSize : smallSize;

};

原因：避免引起歧义

eslint: no-confusing-arrow

## 在隐式return中强制约束函数体的位置，写在箭头后面

// 不好

(foo) =>

bar;

(foo) =>

(bar);

// 好

(foo) => bar;

(foo) => (bar);

(foo) => (

bar

)

eslint: implicit-arrow-linebreak

# 类&构造函数

## 常用class，避免直接操作prototype

// 不好

function Queue(contents = []) {

this.queue = [...contents];

}

Queue.prototype.pop = function () {

const value = this.queue[0];

this.queue.splice(0, 1);

return value;

};

// 好

class Queue {

constructor(contents = []) {

this.queue = [...contents];

}

pop() {

const value = this.queue[0];

this.queue.splice(0, 1);

return value;

}

}

原因：class语法更简洁更易理解

## 用extends实现继承

// 不好

const inherits = require('inherits');

function PeekableQueue(contents) {

Queue.apply(this, contents);

}

inherits(PeekableQueue, Queue);

PeekableQueue.prototype.peek = function () {

return this.\_queue[0];

}

// 好

class PeekableQueue extends Queue {

peek() {

return this.\_queue[0];

}

}

原因：它是一种内置的方法来继承原型功能而不打破instanceof

## 方法可以返回this来实现方法链

// 不好

Jedi.prototype.jump = function () {

this.jumping = true;

return true;

};

Jedi.prototype.setHeight = function (height) {

this.height = height;

};

const luke = new Jedi();

luke.jump(); // => true

luke.setHeight(20); // => undefined

// 好

class Jedi {

jump() {

this.jumping = true;

return this;

}

setHeight(height) {

this.height = height;

return this;

}

}

const luke = new Jedi();

luke.jump()

.setHeight(20);

## 写一个定制的toString()方法是可以的，只要保证它是可以正常工作且没有副作用的

class Jedi {

constructor(options = {}) {

this.name = options.name || 'no name';

}

getName() {

return this.name;

}

toString() {

return `Jedi - ${this.getName()}`;

}

}

## 如果没有具体说明，类有默认的构造方法。

// 不好

class Jedi {

constructor() {}

getName() {

return this.name;

}

}

class Rey extends Jedi {

constructor(...args) {

super(...args);

}

}

// 好

class Rey extends Jedi {

constructor(...args) {

super(...args);

this.name = 'Rey';

}

}

原因：一个空的构造函数或只是代表父类的构造函数是不需要写的

eslint: no-useless-constructor

## 避免重复类成员

// 不好

class Foo {

bar() { return 1; }

bar() { return 2; }

}

// 好

class Foo {

bar() { return 1; }

}

class Foo {

bar() { return 2; }

}

原因：重复类成员会默默的执行最后一个，并且重复本身也是一个bug

eslint: no-dupe-class-members

# 模块