# **<https://github.com/ryanmcdermott/clean-code-javascript>**

# **JavaScript 风格指南**

## **目录**

1. [介绍](https://github.com/alivebao/clean-code-js" \l "%E4%BB%8B%E7%BB%8D)
2. [变量](https://github.com/alivebao/clean-code-js" \l "%E5%8F%98%E9%87%8F)
3. [函数](https://github.com/alivebao/clean-code-js" \l "%E5%87%BD%E6%95%B0)
4. [对象和数据结构](https://github.com/alivebao/clean-code-js" \l "objects-and-data-structures)
5. [类](https://github.com/alivebao/clean-code-js" \l "%E7%B1%BB)
6. [测试](https://github.com/alivebao/clean-code-js" \l "%E6%B5%8B%E8%AF%95)
7. [并发](https://github.com/alivebao/clean-code-js" \l "%E5%B9%B6%E5%8F%91)
8. [错误处理](https://github.com/alivebao/clean-code-js" \l "%E9%94%99%E8%AF%AF%E5%A4%84%E7%90%86)
9. [格式化](https://github.com/alivebao/clean-code-js" \l "%E6%A0%BC%E5%BC%8F%E5%8C%96)
10. [注释](https://github.com/alivebao/clean-code-js" \l "%E6%B3%A8%E9%87%8A)

## **介绍**

[作者](https://github.com/ryanmcdermott)根据 Robert C. Martin [《代码整洁之道》](https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882)总结了适用于 JavaScript 的软件工程原则[《Clean Code JavaScript》](https://github.com/ryanmcdermott/clean-code-javascript)。

本文是对其的翻译。

不必严格遵守本文的所有原则，有时少遵守一些效果可能会更好，具体应根据实际情况决定。这是根据《代码整洁之道》作者多年经验整理的代码优化建议，但也仅仅只是一份建议。

软件工程已经发展了 50 多年，至今仍在不断前进。现在，把这些原则当作试金石，尝试将他们作为团队代码质量考核的标准之一吧。

最后你需要知道的是，这些东西不会让你立刻变成一个优秀的工程师，长期奉行他们也并不意味着你能够高枕无忧不再犯错。千里之行，始于足下。我们需要时常和同行们进行代码评审，不断优化自己的代码。不要惧怕改善代码质量所需付出的努力，加油。

## ****变量****

### **使用有意义，可读性好的变量名**

****反例****:

var yyyymmdstr = moment().format('YYYY/MM/DD');

****正例****:

var yearMonthDay = moment().format('YYYY/MM/DD');

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用 ES6 的 const 定义常量**

反例中使用"var"定义的"常量"是可变的。

在声明一个常量时，该常量在整个程序中都应该是不可变的。

****反例****:

var FIRST\_US\_PRESIDENT = "George Washington";

****正例****:

const FIRST\_US\_PRESIDENT = "George Washington";

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **对功能类似的变量名采用统一的命名风格**

****反例****:

getUserInfo();getClientData();getCustomerRecord();

****正例****:

getUser();

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用易于检索名称**

我们需要阅读的代码远比自己写的要多，使代码拥有良好的可读性且易于检索非常重要。阅读变量名晦涩难懂的代码对读者来说是一种相当糟糕的体验。 让你的变量名易于检索。

****反例****:

// 525600 是什么?for (var i = 0; i < 525600; i++) {

runCronJob();}

****正例****:

// Declare them as capitalized `var` globals.var MINUTES\_IN\_A\_YEAR = 525600;for (var i = 0; i < MINUTES\_IN\_A\_YEAR; i++) {

runCronJob();}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用说明变量(即有意义的变量名)**

****反例****:

const cityStateRegex = /^(.+)[,\\s]+(.+?)\s\*(\d{5})?$/;saveCityState(cityStateRegex.match(cityStateRegex)[1], cityStateRegex.match(cityStateRegex)[2]);

****正例****:

const ADDRESS = 'One Infinite Loop, Cupertino 95014';var cityStateRegex = /^(.+)[,\\s]+(.+?)\s\*(\d{5})?$/;var match = ADDRESS.match(cityStateRegex)var city = match[1];var state = match[2];saveCityState(city, state);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **不要绕太多的弯子**

显式优于隐式。

****反例****:

var locations = ['Austin', 'New York', 'San Francisco'];locations.forEach((l) => {

doStuff();

doSomeOtherStuff();

...

...

...

// l是什么？

dispatch(l);});

****正例****:

var locations = ['Austin', 'New York', 'San Francisco'];locations.forEach((location) => {

doStuff();

doSomeOtherStuff();

...

...

...

dispatch(location);});

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免重复的描述**

当类/对象名已经有意义时，对其变量进行命名不需要再次重复。

****反例****:

var Car = {

carMake: 'Honda',

carModel: 'Accord',

carColor: 'Blue'};

function paintCar(car) {

car.carColor = 'Red';}

****正例****:

var Car = {

make: 'Honda',

model: 'Accord',

color: 'Blue'};

function paintCar(car) {

car.color = 'Red';}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免无意义的条件判断**

****反例****:

function createMicrobrewery(name) {

var breweryName;

if (name) {

breweryName = name;

} else {

breweryName = 'Hipster Brew Co.';

}}

****正例****:

function createMicrobrewery(name) {

var breweryName = name || 'Hipster Brew Co.'}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****函数****

### **函数参数 (理想情况下应不超过 2 个)**

限制函数参数数量很有必要，这么做使得在测试函数时更加轻松。过多的参数将导致难以采用有效的测试用例对函数的各个参数进行测试。

应避免三个以上参数的函数。通常情况下，参数超过两个意味着函数功能过于复杂，这时需要重新优化你的函数。当确实需要多个参数时，大多情况下可以考虑这些参数封装成一个对象。

JS 定义对象非常方便，当需要多个参数时，可以使用一个对象进行替代。

****反例****:

function createMenu(title, body, buttonText, cancellable) {

...}

****正例****:

var menuConfig = {

title: 'Foo',

body: 'Bar',

buttonText: 'Baz',

cancellable: true}

function createMenu(menuConfig) {

...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **函数功能的单一性**

这是软件功能中最重要的原则之一。

功能不单一的函数将导致难以重构、测试和理解。功能单一的函数易于重构，并使代码更加干净。

****反例****:

function emailClients(clients) {

clients.forEach(client => {

let clientRecord = database.lookup(client);

if (clientRecord.isActive()) {

email(client);

}

});}

****正例****:

function emailClients(clients) {

clients.forEach(client => {

emailClientIfNeeded(client);

});}

function emailClientIfNeeded(client) {

if (isClientActive(client)) {

email(client);

}}

function isClientActive(client) {

let clientRecord = database.lookup(client);

return clientRecord.isActive();}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **函数名应明确表明其功能**

****反例****:

function dateAdd(date, month) {

// ...}

let date = new Date();

// 很难理解dateAdd(date, 1)是什么意思dateAdd(date, 1);

****正例****:

function dateAddMonth(date, month) {

// ...}

let date = new Date();dateAddMonth(date, 1);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **函数应该只做一层抽象**

当函数的需要的抽象多于一层时通常意味着函数功能过于复杂，需将其进行分解以提高其可重用性和可测试性。

****反例****:

function parseBetterJSAlternative(code) {

let REGEXES = [

// ...

];

let statements = code.split(' ');

let tokens;

REGEXES.forEach((REGEX) => {

statements.forEach((statement) => {

// ...

})

});

let ast;

tokens.forEach((token) => {

// lex...

});

ast.forEach((node) => {

// parse...

})}

****正例****:

function tokenize(code) {

let REGEXES = [

// ...

];

let statements = code.split(' ');

let tokens;

REGEXES.forEach((REGEX) => {

statements.forEach((statement) => {

// ...

})

});

return tokens;}

function lexer(tokens) {

let ast;

tokens.forEach((token) => {

// lex...

});

return ast;}

function parseBetterJSAlternative(code) {

let tokens = tokenize(code);

let ast = lexer(tokens);

ast.forEach((node) => {

// parse...

})}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **移除重复的代码**

永远、永远、永远不要在任何循环下有重复的代码。

这种做法毫无意义且潜在危险极大。重复的代码意味着逻辑变化时需要对不止一处进行修改。JS 弱类型的特点使得函数拥有更强的普适性。好好利用这一优点吧。

****反例****:

function showDeveloperList(developers) {

developers.forEach(developer => {

var expectedSalary = developer.calculateExpectedSalary();

var experience = developer.getExperience();

var githubLink = developer.getGithubLink();

var data = {

expectedSalary: expectedSalary,

experience: experience,

githubLink: githubLink

};

render(data);

});}

function showManagerList(managers) {

managers.forEach(manager => {

var expectedSalary = manager.calculateExpectedSalary();

var experience = manager.getExperience();

var portfolio = manager.getMBAProjects();

var data = {

expectedSalary: expectedSalary,

experience: experience,

portfolio: portfolio

};

render(data);

});}

****正例****:

function showList(employees) {

employees.forEach(employee => {

var expectedSalary = employee.calculateExpectedSalary();

var experience = employee.getExperience();

var portfolio;

if (employee.type === 'manager') {

portfolio = employee.getMBAProjects();

} else {

portfolio = employee.getGithubLink();

}

var data = {

expectedSalary: expectedSalary,

experience: experience,

portfolio: portfolio

};

render(data);

});}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **采用默认参数精简代码**

****反例****:

function writeForumComment(subject, body) {

subject = subject || 'No Subject';

body = body || 'No text';}

****正例****:

function writeForumComment(subject = 'No subject', body = 'No text') {

...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用 Object.assign 设置默认对象**

****反例****:

var menuConfig = {

title: null,

body: 'Bar',

buttonText: null,

cancellable: true}

function createMenu(config) {

config.title = config.title || 'Foo'

config.body = config.body || 'Bar'

config.buttonText = config.buttonText || 'Baz'

config.cancellable = config.cancellable === undefined ? config.cancellable : true;

}

createMenu(menuConfig);

****正例****:

var menuConfig = {

title: 'Order',

// User did not include 'body' key

buttonText: 'Send',

cancellable: true}

function createMenu(config) {

config = Object.assign({

title: 'Foo',

body: 'Bar',

buttonText: 'Baz',

cancellable: true

}, config);

// config now equals: {title: "Order", body: "Bar", buttonText: "Send", cancellable: true}

// ...}

createMenu(menuConfig);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **不要使用标记(Flag)作为函数参数**

这通常意味着函数的功能的单一性已经被破坏。此时应考虑对函数进行再次划分。

****反例****:

function createFile(name, temp) {

if (temp) {

fs.create('./temp/' + name);

} else {

fs.create(name);

}}

****正例****:

function createTempFile(name) {

fs.create('./temp/' + name);}

----------

function createFile(name) {

fs.create(name);}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免副作用**

当函数产生了除了“接受一个值并返回一个结果”之外的行为时，称该函数产生了副作用。比如写文件、修改全局变量或将你的钱全转给了一个陌生人等。

程序在某些情况下确实需要副作用这一行为，如先前例子中的写文件。这时应该将这些功能集中在一起，不要用多个函数/类修改某个文件。用且只用一个 service 完成这一需求。

****反例****:

// Global variable referenced by following function.// If we had another function that used this name, now it'd be an array and it could break it.var name = 'Ryan McDermott';

function splitIntoFirstAndLastName() {

name = name.split(' ');}

splitIntoFirstAndLastName();

console.log(name); // ['Ryan', 'McDermott'];

****正例****:

function splitIntoFirstAndLastName(name) {

return name.split(' ');}

var name = 'Ryan McDermott'var newName = splitIntoFirstAndLastName(name);

console.log(name); // 'Ryan McDermott';console.log(newName); // ['Ryan', 'McDermott'];

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **不要写全局函数**

在 JS 中污染全局是一个非常不好的实践，这么做可能和其他库起冲突，且调用你的 API 的用户在实际环境中得到一个 exception 前对这一情况是一无所知的。

想象以下例子：如果你想扩展 JS 中的 Array，为其添加一个 diff 函数显示两个数组间的差异，此时应如何去做？你可以将 diff 写入 Array.prototype，但这么做会和其他有类似需求的库造成冲突。如果另一个库对 diff 的需求为比较一个数组中首尾元素间的差异呢？

使用 ES6 中的 class 对全局的 Array 做简单的扩展显然是一个更棒的选择。

****反例****:

Array.prototype.diff = function(comparisonArray) {

var values = [];

var hash = {};

for (var i of comparisonArray) {

hash[i] = true;

}

for (var i of this) {

if (!hash[i]) {

values.push(i);

}

}

return values;}

****正例****:

class SuperArray extends Array {

constructor(...args) {

super(...args);

}

diff(comparisonArray) {

var values = [];

var hash = {};

for (var i of comparisonArray) {

hash[i] = true;

}

for (var i of this) {

if (!hash[i]) {

values.push(i);

}

}

return values;

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **采用函数式编程**

函数式的编程具有更干净且便于测试的特点。尽可能的使用这种风格吧。

****反例****:

const programmerOutput = [

{

name: 'Uncle Bobby',

linesOfCode: 500

}, {

name: 'Suzie Q',

linesOfCode: 1500

}, {

name: 'Jimmy Gosling',

linesOfCode: 150

}, {

name: 'Gracie Hopper',

linesOfCode: 1000

}];

var totalOutput = 0;

for (var i = 0; i < programmerOutput.length; i++) {

totalOutput += programmerOutput[i].linesOfCode;}

****正例****:

const programmerOutput = [

{

name: 'Uncle Bobby',

linesOfCode: 500

}, {

name: 'Suzie Q',

linesOfCode: 1500

}, {

name: 'Jimmy Gosling',

linesOfCode: 150

}, {

name: 'Gracie Hopper',

linesOfCode: 1000

}];

var totalOutput = programmerOutput

.map((programmer) => programmer.linesOfCode)

.reduce((acc, linesOfCode) => acc + linesOfCode, 0);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **封装判断条件**

****反例****:

if (fsm.state === 'fetching' && isEmpty(listNode)) {

/// ...}

****正例****:

function shouldShowSpinner(fsm, listNode) {

return fsm.state === 'fetching' && isEmpty(listNode);}

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {

// ...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免“否定情况”的判断**

****反例****:

function isDOMNodeNotPresent(node) {

// ...}

if (!isDOMNodeNotPresent(node)) {

// ...}

****正例****:

function isDOMNodePresent(node) {

// ...}

if (isDOMNodePresent(node)) {

// ...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免条件判断**

这看起来似乎不太可能。

大多人听到这的第一反应是：“怎么可能不用 if 完成其他功能呢？”许多情况下通过使用多态(polymorphism)可以达到同样的目的。

第二个问题在于采用这种方式的原因是什么。答案是我们之前提到过的：保持函数功能的单一性。

****反例****:

class Airplane {

//...

getCruisingAltitude() {

switch (this.type) {

case '777':

return getMaxAltitude() - getPassengerCount();

case 'Air Force One':

return getMaxAltitude();

case 'Cessna':

return getMaxAltitude() - getFuelExpenditure();

}

}}

****正例****:

class Airplane {

//...}

class Boeing777 extends Airplane {

//...

getCruisingAltitude() {

return getMaxAltitude() - getPassengerCount();

}}

class AirForceOne extends Airplane {

//...

getCruisingAltitude() {

return getMaxAltitude();

}}

class Cessna extends Airplane {

//...

getCruisingAltitude() {

return getMaxAltitude() - getFuelExpenditure();

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免类型判断(part 1)**

JS 是弱类型语言，这意味着函数可接受任意类型的参数。

有时这会对你带来麻烦，你会对参数做一些类型判断。有许多方法可以避免这些情况。

****反例****:

function travelToTexas(vehicle) {

if (vehicle instanceof Bicycle) {

vehicle.peddle(this.currentLocation, new Location('texas'));

} else if (vehicle instanceof Car) {

vehicle.drive(this.currentLocation, new Location('texas'));

}}

****正例****:

function travelToTexas(vehicle) {

vehicle.move(this.currentLocation, new Location('texas'));}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免类型判断(part 2)**

如果需处理的数据为字符串，整型，数组等类型，无法使用多态并仍有必要对其进行类型检测时，可以考虑使用 TypeScript。

****反例****:

function combine(val1, val2) {

if (typeof val1 == "number" && typeof val2 == "number" ||

typeof val1 == "string" && typeof val2 == "string") {

return val1 + val2;

} else {

throw new Error('Must be of type String or Number');

}}

****正例****:

function combine(val1, val2) {

return val1 + val2;}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免过度优化**

现代的浏览器在运行时会对代码自动进行优化。有时人为对代码进行优化可能是在浪费时间。

[这里可以找到许多真正需要优化的地方](https://github.com/petkaantonov/bluebird/wiki/Optimization-killers)

****反例****:

// 这里使用变量len是因为在老式浏览器中，// 直接使用正例中的方式会导致每次循环均重复计算list.length的值，// 而在现代浏览器中会自动完成优化，这一行为是没有必要的for (var i = 0, len = list.length; i < len; i++) {

// ...}

****正例****:

for (var i = 0; i < list.length; i++) {

// ...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **删除无效的代码**

不再被调用的代码应及时删除。

****反例****:

function oldRequestModule(url) {

// ...}

function newRequestModule(url) {

// ...}

var req = newRequestModule;inventoryTracker('apples', req, 'www.inventory-awesome.io');

****正例****:

function newRequestModule(url) {

// ...}

var req = newRequestModule;inventoryTracker('apples', req, 'www.inventory-awesome.io');

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****对象和数据结构****

### **使用 getters 和 setters**

JS 没有接口或类型，因此实现这一模式是很困难的，因为我们并没有类似 public 和 private 的关键词。

然而，使用 getters 和 setters 获取对象的数据远比直接使用点操作符具有优势。为什么呢？

1. 当需要对获取的对象属性执行额外操作时。
2. 执行 set 时可以增加规则对要变量的合法性进行判断。
3. 封装了内部逻辑。
4. 在存取时可以方便的增加日志和错误处理。
5. 继承该类时可以重载默认行为。
6. 从服务器获取数据时可以进行懒加载。

****反例****:

class BankAccount {

constructor() {

this.balance = 1000;

}}

let bankAccount = new BankAccount();

// Buy shoes...bankAccount.balance = bankAccount.balance - 100;

****正例****:

class BankAccount {

constructor() {

this.balance = 1000;

}

// It doesn't have to be prefixed with `get` or `set` to be a getter/setter

withdraw(amount) {

if (verifyAmountCanBeDeducted(amount)) {

this.balance -= amount;

}

}}

let bankAccount = new BankAccount();

// Buy shoes...bankAccount.withdraw(100);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **让对象拥有私有成员**

可以通过闭包完成

****反例****:

var Employee = function(name) {

this.name = name;}

Employee.prototype.getName = function() {

return this.name;}

var employee = new Employee('John Doe');console.log('Employee name: ' + employee.getName()); // Employee name: John Doedelete employee.name;console.log('Employee name: ' + employee.getName()); // Employee name: undefined

****正例****:

var Employee = (function() {

function Employee(name) {

this.getName = function() {

return name;

};

}

return Employee;}());

var employee = new Employee('John Doe');console.log('Employee name: ' + employee.getName()); // Employee name: John Doedelete employee.name;console.log('Employee name: ' + employee.getName()); // Employee name: John Doe

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****类****

### **单一职责原则 (SRP)**

如《代码整洁之道》一书中所述，“修改一个类的理由不应该超过一个”。

将多个功能塞进一个类的想法很诱人，但这将导致你的类无法达到概念上的内聚，并经常不得不进行修改。

最小化对一个类需要修改的次数是非常有必要的。如果一个类具有太多太杂的功能，当你对其中一小部分进行修改时，将很难想象到这一修够对代码库中依赖该类的其他模块会带来什么样的影响。

****反例****:

class UserSettings {

constructor(user) {

this.user = user;

}

changeSettings(settings) {

if (this.verifyCredentials(user)) {

// ...

}

}

verifyCredentials(user) {

// ...

}}

****正例****:

class UserAuth {

constructor(user) {

this.user = user;

}

verifyCredentials() {

// ...

}}

class UserSettings {

constructor(user) {

this.user = user;

this.auth = new UserAuth(user)

}

changeSettings(settings) {

if (this.auth.verifyCredentials()) {

// ...

}

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **开/闭原则 (OCP)**

“代码实体(类，模块，函数等)应该易于扩展，难于修改。”

这一原则指的是我们应允许用户方便的扩展我们代码模块的功能，而不需要打开 js 文件源码手动对其进行修改。

****反例****:

class AjaxRequester {

constructor() {

// What if we wanted another HTTP Method, like DELETE? We would have to

// open this file up and modify this and put it in manually.

this.HTTP\_METHODS = ['POST', 'PUT', 'GET'];

}

get(url) {

// ...

}

}

****正例****:

class AjaxRequester {

constructor() {

this.HTTP\_METHODS = ['POST', 'PUT', 'GET'];

}

get(url) {

// ...

}

addHTTPMethod(method) {

this.HTTP\_METHODS.push(method);

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **利斯科夫替代原则 (LSP)**

“子类对象应该能够替换其超类对象被使用”。

也就是说，如果有一个父类和一个子类，当采用子类替换父类时不应该产生错误的结果。

****反例****:

class Rectangle {

constructor() {

this.width = 0;

this.height = 0;

}

setColor(color) {

// ...

}

render(area) {

// ...

}

setWidth(width) {

this.width = width;

}

setHeight(height) {

this.height = height;

}

getArea() {

return this.width \* this.height;

}}

class Square extends Rectangle {

constructor() {

super();

}

setWidth(width) {

this.width = width;

this.height = width;

}

setHeight(height) {

this.width = height;

this.height = height;

}}

function renderLargeRectangles(rectangles) {

rectangles.forEach((rectangle) => {

rectangle.setWidth(4);

rectangle.setHeight(5);

let area = rectangle.getArea(); // BAD: Will return 25 for Square. Should be 20.

rectangle.render(area);

})}

let rectangles = [new Rectangle(), new Rectangle(), new Square()];renderLargeRectangles(rectangles);

****正例****:

class Shape {

constructor() {}

setColor(color) {

// ...

}

render(area) {

// ...

}}

class Rectangle extends Shape {

constructor() {

super();

this.width = 0;

this.height = 0;

}

setWidth(width) {

this.width = width;

}

setHeight(height) {

this.height = height;

}

getArea() {

return this.width \* this.height;

}}

class Square extends Shape {

constructor() {

super();

this.length = 0;

}

setLength(length) {

this.length = length;

}

getArea() {

return this.length \* this.length;

}}

function renderLargeShapes(shapes) {

shapes.forEach((shape) => {

switch (shape.constructor.name) {

case 'Square':

shape.setLength(5);

case 'Rectangle':

shape.setWidth(4);

shape.setHeight(5);

}

let area = shape.getArea();

shape.render(area);

})}

let shapes = [new Rectangle(), new Rectangle(), new Square()];renderLargeShapes(shapes);

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **接口隔离原则 (ISP)**

“客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。”

在 JS 中，当一个类需要许多参数设置才能生成一个对象时，或许大多时候不需要设置这么多的参数。此时减少对配置参数数量的需求是有益的。

****反例****:

class DOMTraverser {

constructor(settings) {

this.settings = settings;

this.setup();

}

setup() {

this.rootNode = this.settings.rootNode;

this.animationModule.setup();

}

traverse() {

// ...

}}

let $ = new DOMTraverser({

rootNode: document.getElementsByTagName('body'),

animationModule: function() {} // Most of the time, we won't need to animate when traversing.

// ...});

****正例****:

class DOMTraverser {

constructor(settings) {

this.settings = settings;

this.options = settings.options;

this.setup();

}

setup() {

this.rootNode = this.settings.rootNode;

this.setupOptions();

}

setupOptions() {

if (this.options.animationModule) {

// ...

}

}

traverse() {

// ...

}}

let $ = new DOMTraverser({

rootNode: document.getElementsByTagName('body'),

options: {

animationModule: function() {}

}});

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **依赖反转原则 (DIP)**

该原则有两个核心点：

1. 高层模块不应该依赖于低层模块。他们都应该依赖于抽象接口。
2. 抽象接口应该脱离具体实现，具体实现应该依赖于抽象接口。

****反例****:

class InventoryTracker {

constructor(items) {

this.items = items;

// BAD: We have created a dependency on a specific request implementation.

// We should just have requestItems depend on a request method: `request`

this.requester = new InventoryRequester();

}

requestItems() {

this.items.forEach((item) => {

this.requester.requestItem(item);

});

}}

class InventoryRequester {

constructor() {

this.REQ\_METHODS = ['HTTP'];

}

requestItem(item) {

// ...

}}

let inventoryTracker = new InventoryTracker(['apples', 'bananas']);inventoryTracker.requestItems();

****正例****:

class InventoryTracker {

constructor(items, requester) {

this.items = items;

this.requester = requester;

}

requestItems() {

this.items.forEach((item) => {

this.requester.requestItem(item);

});

}}

class InventoryRequesterV1 {

constructor() {

this.REQ\_METHODS = ['HTTP'];

}

requestItem(item) {

// ...

}}

class InventoryRequesterV2 {

constructor() {

this.REQ\_METHODS = ['WS'];

}

requestItem(item) {

// ...

}}

// By constructing our dependencies externally and injecting them, we can easily// substitute our request module for a fancy new one that uses WebSockets.let inventoryTracker = new InventoryTracker(['apples', 'bananas'], new InventoryRequesterV2());inventoryTracker.requestItems();

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用 ES6 的 classes 而不是 ES5 的 Function**

典型的 ES5 的类(function)在继承、构造和方法定义方面可读性较差。

当需要继承时，优先选用 classes。

但是，当在需要更大更复杂的对象时，最好优先选择更小的 function 而非 classes。

****反例****:

var Animal = function(age) {

if (!(this instanceof Animal)) {

throw new Error("Instantiate Animal with `new`");

}

this.age = age;};

Animal.prototype.move = function() {};

var Mammal = function(age, furColor) {

if (!(this instanceof Mammal)) {

throw new Error("Instantiate Mammal with `new`");

}

Animal.call(this, age);

this.furColor = furColor;};

Mammal.prototype = Object.create(Animal.prototype);Mammal.prototype.constructor = Mammal;Mammal.prototype.liveBirth = function() {};

var Human = function(age, furColor, languageSpoken) {

if (!(this instanceof Human)) {

throw new Error("Instantiate Human with `new`");

}

Mammal.call(this, age, furColor);

this.languageSpoken = languageSpoken;};

Human.prototype = Object.create(Mammal.prototype);Human.prototype.constructor = Human;Human.prototype.speak = function() {};

****正例****:

class Animal {

constructor(age) {

this.age = age;

}

move() {}}

class Mammal extends Animal {

constructor(age, furColor) {

super(age);

this.furColor = furColor;

}

liveBirth() {}}

class Human extends Mammal {

constructor(age, furColor, languageSpoken) {

super(age, furColor);

this.languageSpoken = languageSpoken;

}

speak() {}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **使用方法链**

这里我们的理解与《代码整洁之道》的建议有些不同。

有争论说方法链不够干净且违反了[德米特法则](https://en.wikipedia.org/wiki/Law_of_Demeter)，也许这是对的，但这种方法在 JS 及许多库(如 JQuery)中显得非常实用。

因此，我认为在 JS 中使用方法链是非常合适的。在 class 的函数中返回 this，能够方便的将类需要执行的多个方法链接起来。

****反例****:

class Car {

constructor() {

this.make = 'Honda';

this.model = 'Accord';

this.color = 'white';

}

setMake(make) {

this.name = name;

}

setModel(model) {

this.model = model;

}

setColor(color) {

this.color = color;

}

save() {

console.log(this.make, this.model, this.color);

}}

let car = new Car();car.setColor('pink');car.setMake('Ford');car.setModel('F-150')car.save();

****正例****:

class Car {

constructor() {

this.make = 'Honda';

this.model = 'Accord';

this.color = 'white';

}

setMake(make) {

this.name = name;

// NOTE: Returning this for chaining

return this;

}

setModel(model) {

this.model = model;

// NOTE: Returning this for chaining

return this;

}

setColor(color) {

this.color = color;

// NOTE: Returning this for chaining

return this;

}

save() {

console.log(this.make, this.model, this.color);

}}

let car = new Car()

.setColor('pink')

.setMake('Ford')

.setModel('F-150')

.save();

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **优先使用组合模式而非继承**

在著名的[设计模式](https://en.wikipedia.org/wiki/Design_Patterns)一书中提到，应多使用组合模式而非继承。

这么做有许多优点，在想要使用继承前，多想想能否通过组合模式满足需求吧。

那么，在什么时候继承具有更大的优势呢？这取决于你的具体需求，但大多情况下，可以遵守以下三点：

1. 继承关系表现为"是一个"而非"有一个"(如动物->人 和 用户->用户细节)
2. 可以复用基类的代码("Human"可以看成是"All animal"的一种)
3. 希望当基类改变时所有派生类都受到影响(如修改"all animals"移动时的卡路里消耗量)

****反例****:

class Employee {

constructor(name, email) {

this.name = name;

this.email = email;

}

// ...}

// Bad because Employees "have" tax data. EmployeeTaxData is not a type of Employeeclass EmployeeTaxData extends Employee {

constructor(ssn, salary) {

super();

this.ssn = ssn;

this.salary = salary;

}

// ...}

****正例****:

class Employee {

constructor(name, email) {

this.name = name;

this.email = email;

}

setTaxData(ssn, salary) {

this.taxData = new EmployeeTaxData(ssn, salary);

}

// ...}

class EmployeeTaxData {

constructor(ssn, salary) {

this.ssn = ssn;

this.salary = salary;

}

// ...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****测试****

[一些好的覆盖工具](http://gotwarlost.github.io/istanbul/)。

[一些好的 JS 测试框架](http://jstherightway.org/" \l "testing-tools)。

### **单一的测试每个概念**

****反例****:

const assert = require('assert');

describe('MakeMomentJSGreatAgain', function() {

it('handles date boundaries', function() {

let date;

date = new MakeMomentJSGreatAgain('1/1/2015');

date.addDays(30);

date.shouldEqual('1/31/2015');

date = new MakeMomentJSGreatAgain('2/1/2016');

date.addDays(28);

assert.equal('02/29/2016', date);

date = new MakeMomentJSGreatAgain('2/1/2015');

date.addDays(28);

assert.equal('03/01/2015', date);

});});

****正例****:

const assert = require('assert');

describe('MakeMomentJSGreatAgain', function() {

it('handles 30-day months', function() {

let date = new MakeMomentJSGreatAgain('1/1/2015');

date.addDays(30);

date.shouldEqual('1/31/2015');

});

it('handles leap year', function() {

let date = new MakeMomentJSGreatAgain('2/1/2016');

date.addDays(28);

assert.equal('02/29/2016', date);

});

it('handles non-leap year', function() {

let date = new MakeMomentJSGreatAgain('2/1/2015');

date.addDays(28);

assert.equal('03/01/2015', date);

});});

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****并发****

### **用 Promises 替代回调**

回调不够整洁并会造成大量的嵌套。ES6 内嵌了 Promises，使用它吧。

****反例****:

require('request').get('https://en.wikipedia.org/wiki/Robert\_Cecil\_Martin', function(err, response) {

if (err) {

console.error(err);

}

else {

require('fs').writeFile('article.html', response.body, function(err) {

if (err) {

console.error(err);

} else {

console.log('File written');

}

})

}})

****正例****:

require('request-promise').get('https://en.wikipedia.org/wiki/Robert\_Cecil\_Martin')

.then(function(response) {

return require('fs-promise').writeFile('article.html', response);

})

.then(function() {

console.log('File written');

})

.catch(function(err) {

console.error(err);

})

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **Async/Await 是较 Promises 更好的选择**

Promises 是较回调而言更好的一种选择，但 ES7 中的 async 和 await 更胜过 Promises。

在能使用 ES7 特性的情况下可以尽量使用他们替代 Promises。

****反例****:

require('request-promise').get('https://en.wikipedia.org/wiki/Robert\_Cecil\_Martin')

.then(function(response) {

return require('fs-promise').writeFile('article.html', response);

})

.then(function() {

console.log('File written');

})

.catch(function(err) {

console.error(err);

})

****正例****:

async function getCleanCodeArticle() {

try {

var request = await require('request-promise')

var response = await request.get('https://en.wikipedia.org/wiki/Robert\_Cecil\_Martin');

var fileHandle = await require('fs-promise');

await fileHandle.writeFile('article.html', response);

console.log('File written');

} catch(err) {

console.log(err);

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****错误处理****

错误抛出是个好东西！这使得你能够成功定位运行状态中的程序产生错误的位置。

### **别忘了捕获错误**

对捕获的错误不做任何处理是没有意义的。

代码中 try/catch 的意味着你认为这里可能出现一些错误，你应该对这些可能的错误存在相应的处理方案。

****反例****:

try {

functionThatMightThrow();} catch (error) {

console.log(error);}

****正例****:

try {

functionThatMightThrow();} catch (error) {

// One option (more noisy than console.log):

console.error(error);

// Another option:

notifyUserOfError(error);

// Another option:

reportErrorToService(error);

// OR do all three!}

### **不要忽略被拒绝的 promises**

理由同 try/catch。

****反例****:

getdata().then(data => {

functionThatMightThrow(data);}).catch(error => {

console.log(error);});

****正例****:

getdata().then(data => {

functionThatMightThrow(data);}).catch(error => {

// One option (more noisy than console.log):

console.error(error);

// Another option:

notifyUserOfError(error);

// Another option:

reportErrorToService(error);

// OR do all three!});

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****格式化****

格式化是一件主观的事。如同这里的许多规则一样，这里并没有一定/立刻需要遵守的规则。可以在[这里](http://standardjs.com/rules.html)完成格式的自动化。

### **大小写一致**

JS 是弱类型语言，合理的采用大小写可以告诉你关于变量/函数等的许多消息。

这些规则是主观定义的，团队可以根据喜欢进行选择。重点在于无论选择何种风格，都需要注意保持一致性。

****反例****:

var DAYS\_IN\_WEEK = 7;var daysInMonth = 30;

var songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];var Artists = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {}function restore\_database() {}

class animal {}class Alpaca {}

****正例****:

var DAYS\_IN\_WEEK = 7;var DAYS\_IN\_MONTH = 30;

var songs = ['Back In Black', 'Stairway to Heaven', 'Hey Jude'];var artists = ['ACDC', 'Led Zeppelin', 'The Beatles'];

function eraseDatabase() {}function restoreDatabase() {}

class Animal {}class Alpaca {}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **调用函数的函数和被调函数应放在较近的位置**

当函数间存在相互调用的情况时，应将两者置于较近的位置。

理想情况下，应将调用其他函数的函数写在被调用函数的上方。

****反例****:

class PerformanceReview {

constructor(employee) {

this.employee = employee;

}

lookupPeers() {

return db.lookup(this.employee, 'peers');

}

lookupMananger() {

return db.lookup(this.employee, 'manager');

}

getPeerReviews() {

let peers = this.lookupPeers();

// ...

}

perfReview() {

getPeerReviews();

getManagerReview();

getSelfReview();

}

getManagerReview() {

let manager = this.lookupManager();

}

getSelfReview() {

// ...

}}

let review = new PerformanceReview(user);review.perfReview();

****正例****:

class PerformanceReview {

constructor(employee) {

this.employee = employee;

}

perfReview() {

getPeerReviews();

getManagerReview();

getSelfReview();

}

getPeerReviews() {

let peers = this.lookupPeers();

// ...

}

lookupPeers() {

return db.lookup(this.employee, 'peers');

}

getManagerReview() {

let manager = this.lookupManager();

}

lookupMananger() {

return db.lookup(this.employee, 'manager');

}

getSelfReview() {

// ...

}}

let review = new PerformanceReview(employee);review.perfReview();

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

## ****注释****

### **只对存在一定业务逻辑复杂性的代码进行注释**

注释并不是必须的，好的代码是能够让人一目了然，不用过多无谓的注释。

****反例****:

function hashIt(data) {

// The hash

var hash = 0;

// Length of string

var length = data.length;

// Loop through every character in data

for (var i = 0; i < length; i++) {

// Get character code.

var char = data.charCodeAt(i);

// Make the hash

hash = ((hash << 5) - hash) + char;

// Convert to 32-bit integer

hash = hash & hash;

}}

****正例****:

function hashIt(data) {

var hash = 0;

var length = data.length;

for (var i = 0; i < length; i++) {

var char = data.charCodeAt(i);

hash = ((hash << 5) - hash) + char;

// Convert to 32-bit integer

hash = hash & hash;

}}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **不要在代码库中遗留被注释掉的代码**

版本控制的存在是有原因的。让旧代码存在于你的 history 里吧。

****反例****:

doStuff();// doOtherStuff();// doSomeMoreStuff();// doSoMuchStuff();

****正例****:

doStuff();

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **不需要版本更新类型注释**

记住，我们可以使用版本控制。废代码、被注释的代码及用注释记录代码中的版本更新说明都是没有必要的。

需要时可以使用 git log 获取历史版本。

****反例****:

/\*\* \* 2016-12-20: Removed monads, didn't understand them (RM) \* 2016-10-01: Improved using special monads (JP) \* 2016-02-03: Removed type-checking (LI) \* 2015-03-14: Added combine with type-checking (JR) \*/function combine(a, b) {

return a + b;}

****正例****:

function combine(a, b) {

return a + b;}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免位置标记**

这些东西通常只能代码麻烦，采用适当的缩进就可以了。

****反例****:

////////////////////////////////////////////////////////////////////////////////// Scope Model Instantiation////////////////////////////////////////////////////////////////////////////////let $scope.model = {

menu: 'foo',

nav: 'bar'};

////////////////////////////////////////////////////////////////////////////////// Action setup////////////////////////////////////////////////////////////////////////////////let actions = function() {

// ...}

****正例****:

let $scope.model = {

menu: 'foo',

nav: 'bar'};

let actions = function() {

// ...}

**[回到目录](https://github.com/alivebao/clean-code-js" \l "%E7%9B%AE%E5%BD%95)**

### **避免在源文件中写入法律评论**

将你的 LICENSE 文件置于源码目录树的根目录。

****反例****:

/\*The MIT License (MIT)Copyright (c) 2016 Ryan McDermottPermission is hereby granted, free of charge, to any person obtaining a copyof this software and associated documentation files (the "Software"), to dealin the Software without restriction, including without limitation the rightsto use, copy, modify, merge, publish, distribute, sublicense, and/or sellcopies of the Software, and to permit persons to whom the Software isfurnished to do so, subject to the following conditions:The above copyright notice and this permission notice shall be included in allcopies or substantial portions of the Software.THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS ORIMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THEAUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHERLIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THESOFTWARE\*/

function calculateBill() {

// ...}

****正例****:

function calculateBill() {

// ...}