

# 代码整洁的 JavaScript

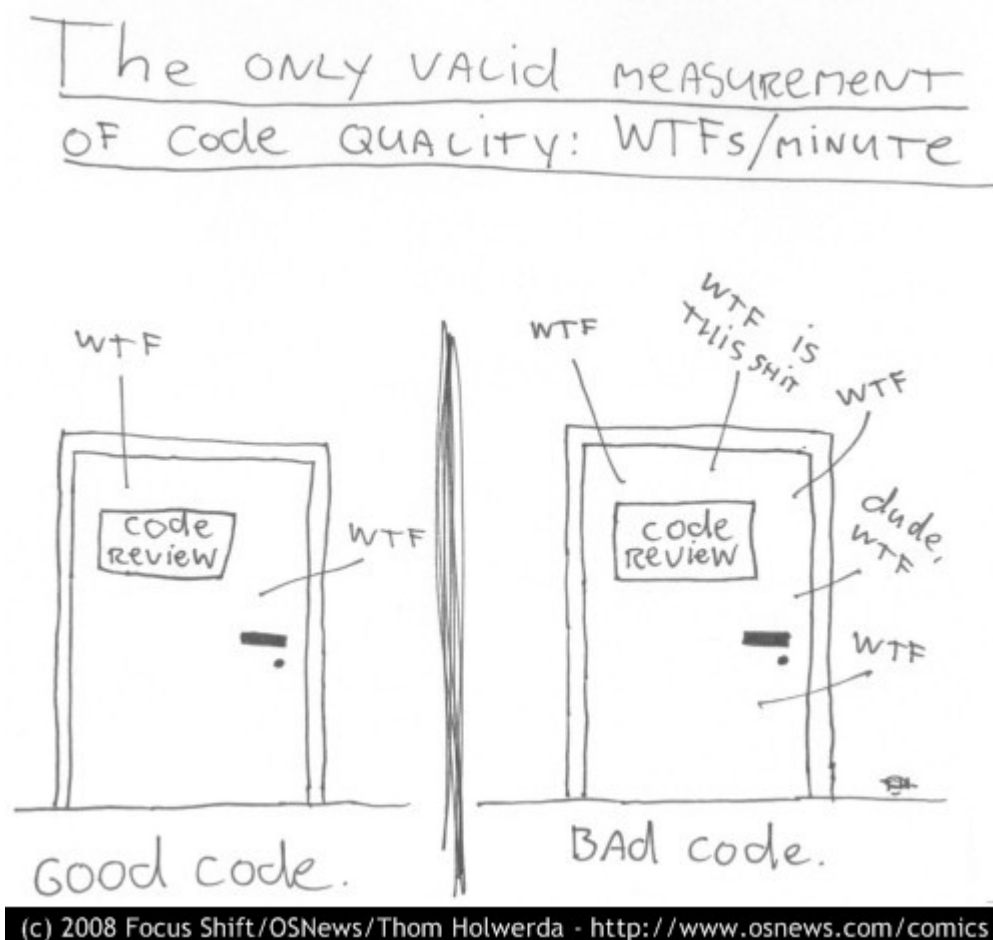
---

## 目录

---

1. [简介](#)
2. [变量](#)
3. [函数](#)
4. [对象和数据结构](#)
5. [类](#)
6. [SOLID](#)
7. [测试](#)
8. [并发](#)
9. [错误处理](#)
10. [格式化](#)
11. [注释](#)

## 简介



将源自 Robert C. Martin 的 [Clean Code](#)

的软件工程原则适配到 JavaScript。这不是一个代码风格指南，它是一个使用 JavaScript 来生产可读的，可重用的，以及可重构的软件的指南。

这里的每一项原则都不是必须遵守的，甚至只有更少的能够被广泛认可。这些仅仅是指南而已，但是却是 *Clean Code* 作者多年经验的结晶。

我们的软件工程行业只有短短的 50 年，依然有很多要我们去学习。当软件架构与建筑架构一样古老时，也许我们将会有硬性的规则去遵守。而现在，让这些指南做为你和你的团队生产的 JavaScript 代码的质量的标准。

还有一件事：知道这些指南并不能马上让你成为一个更加出色的软件开发者，并且使用它们工作多年也并不意味着你不再会犯错误。每一段代码最开始都是草稿，像湿粘土一样被打造成最终的形态。最后当我们和搭档们一起审查代码时清除那些不完善之处，不要因为最初需要改善的草稿代码而自责，而是对那些代码下手。

## 变量

---

### 使用有意义并且可读的变量名称

不好的：

```
const yyyymmddstr = moment().format('YYYY/MM/DD');
```

好的：

```
const currentDate = moment().format('YYYY/MM/DD');
```

[↑ 返回顶部](#)

### 为相同类型的变量使用相同的词汇

不好的：

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

好的：

```
getUser();
```

[↑ 返回顶部](#)

## 使用可搜索的名称

我们要阅读的代码比要写的代码多得多，所以我们写出的代码的可读性和可搜索性是很重要的。使用没有意义的变量名将会导致我们的程序难于理解，将会伤害我们的读者，所以请使用可搜索的变量名。类似 [buddy.js](#) 和 [ESLint](#) 的工具可以帮助我们找到未命名的常量。

不好的：

```
// ++, 86400000 是什么鬼？
setTimeout(blastOff, 86400000);
```

好的：

```
// 将它们声明为全局常量 `const`。
const MILLISECONDS_IN_A_DAY = 86400000;

setTimeout(blastOff, MILLISECONDS_IN_A_DAY);
```

[↑ 返回顶部](#)

## 使用解释性的变量

不好的：

```
const address = 'One Infinite Loop, Cupertino 95014';
const cityZipCodeRegex = /^[^\,\\]+[\,\\s]+(?:\s*(\d{5})?)?$/;
saveCityZipCode(address.match(cityZipCodeRegex)[1], address.matc
```

好的：

```
const address = 'One Infinite Loop, Cupertino 95014';
const cityZipCodeRegex = /^[^\,\\]+[\,\\s]+(?:\s*(\d{5})?)?$/;
const [, city, zipCode] = address.match(cityZipCodeRegex) || [];
saveCityZipCode(city, zipCode);
```

[↑ 返回顶部](#)

## 避免心理映射

显示比隐式更好

不好的：

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((l) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  // 等等, `l` 是啥?
  dispatch(l);
});
```

好的：

```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((location) => {
  doStuff();
  doSomeOtherStuff();
  // ...
  // ...
  // ...
  dispatch(location);
});
```

[↑ 返回顶部](#)

## 不添加不必要的上下文

如果你的类名/对象名有意义，不要在变量名上再重复。

不好的：

```
const Car = {
  carMake: 'Honda',
  carModel: 'Accord',
  carColor: 'Blue'
```

```
};

function paintCar(car) {
  car.carColor = 'Red';
}
```

**好的：**

```
const Car = {
  make: 'Honda',
  model: 'Accord',
  color: 'Blue'
};

function paintCar(car) {
  car.color = 'Red';
}
```

[↑ 返回顶部](#)

## 使用默认变量替代短路运算或条件

**不好的：**

```
function createMicrobrewery(name) {
  const breweryName = name || 'Hipster Brew Co.';
  // ...
}
```

**好的：**

```
function createMicrobrewery(breweryName = 'Hipster Brew Co.') {
  // ...
}
```

[↑ 返回顶部](#)

# 函数

---

## 函数参数 (两个以下最理想)

限制函数参数的个数是非常重要的，因为这样将使你的函数容易进行测试。一旦超过三个参数将会导致组合爆炸，因为你不得不编写大量针对每个参数的测试用例。

没有参数是最理想的，一个或者两个参数也是可以的，三个参数应该避免，超过三个应该被重构。通常，如果你有一个超过两个函数的参数，那就意味着你的函数尝试做太多的事情。如果不是，多数情况下一个更高级对象可能会满足需求。

由于 JavaScript 允许我们不定义类型/模板就可以创建对象，当你发现你自己需要大量的参数时，你可以使用一个对象。

**不好的：**

```
function createMenu(title, body, buttonText, cancellable) {  
  // ...  
}
```

**好的：**

```
const menuConfig = {  
  title: 'Foo',  
  body: 'Bar',  
  buttonText: 'Baz',  
  cancellable: true  
};  
  
function createMenu(config) {  
  // ...  
}
```

[↑ 返回顶部](#)

## 函数应当只做一件事情

这是软件工程中最重要的一条规则，当函数需要做更多的事情时，它们将会更难进行编写、测试和推理。

当你能将一个函数隔离到只有一个动作，他们将能够被容易的进行重构并且你的代码将会更容易阅读。如

果你严格遵守本指南中的这一条，你将会领先于许多开发者。

**不好的：**

```
function emailClients(clients) {
  clients.forEach((client) => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}
```

**好的：**

```
function emailClients(clients) {
  clients
    .filter(isClientActive)
    .forEach(email);
}

function isClientActive(client) {
  const clientRecord = database.lookup(client);
  return clientRecord.isActive();
}
```

[↑ 返回顶部](#)

## 函数名称应该说明它要做什么

**不好的：**

```
function addToDate(date, month) {
  // ...
}
```



```
const date = new Date();

// 很难从函数名看出加了什么
addToDate(date, 1);
```

**好的：**

```
function addMonthToDate(month, date) {
  // ...
}

const date = new Date();
addMonthToDate(1, date);
```

[↑ 返回顶部](#)

## 函数应该只有一个抽象级别

当在你的函数中有多于一个抽象级别时，你的函数通常做了太多事情。拆分函数将会提升重用性和测试性。

**不好的：**

```
function parseBetterJSAlternative(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      // ...
    });
  });

  const ast = [];
  tokens.forEach((token) => {
    // lex...
  });

  ast.forEach((node) => {
    // parse...
  });
}
```

好的：

```
function tokenize(code) {
  const REGEXES = [
    // ...
  ];

  const statements = code.split(' ');
  const tokens = [];
  REGEXES.forEach((REGEX) => {
    statements.forEach((statement) => {
      tokens.push( /* ... */ );
    });
  });

  return tokens;
}

function lexer(tokens) {
  const ast = [];
  tokens.forEach((token) => {
    ast.push( /* ... */ );
  });

  return ast;
}

function parseBetterJSAlternative(code) {
  const tokens = tokenize(code);
  const ast = lexer(tokens);
  ast.forEach((node) => {
    // parse...
  });
}
```

[↑ 返回顶部](#)

## 移除冗余代码

竭尽你的全力去避免冗余代码。冗余代码是不好的，因为它意味着当你需要修改一些逻辑时会有多个地方需要修改。

想象一下你在经营一家餐馆，你需要记录所有的库存西红柿，洋葱，大蒜，各种香料等等。如果你有多

个记录列表，当你用西红柿做一道菜时你得更新多个列表。如果你只有一个列表，就只有一个地方需要更新！

你有冗余代码通常是因为你有两个或多个稍微不同的东西，它们共享大部分，但是它们的不同之处迫使你使用两个或更多独立的函数来处理大部分相同的东西。移除冗余代码意味着创建一个可以处理这些不同之处的抽象的函数/模块/类。

让这个抽象正确是关键，这是为什么要你遵循 *Classes* 那一章的 SOLID 的原因。不好的抽象比冗余代码更差，所以要谨慎行事。既然已经这么说了，如果你能够做出一个好的抽象，才去做。不要重复你自己，否则你会发现当你要修改一个东西时时刻需要修改多个地方。

**不好的：**

```
function showDeveloperList(developers) {
  developers.forEach((developer) => {
    const expectedSalary = developer.calculateExpectedSalary();
    const experience = developer.getExperience();
    const githubLink = developer.getGithubLink();
    const data = {
      expectedSalary,
      experience,
      githubLink
    };

    render(data);
  });
}

function showManagerList(managers) {
  managers.forEach((manager) => {
    const expectedSalary = manager.calculateExpectedSalary();
    const experience = manager.getExperience();
    const portfolio = manager.getMBAPProjects();
    const data = {
      expectedSalary,
      experience,
      portfolio
    };

    render(data);
  });
}
```

```
});  
}
```

**好的：**

```
function showList(employees) {  
  employees.forEach((employee) => {  
    const expectedSalary = employee.calculateExpectedSalary();  
    const experience = employee.getExperience();  
  
    let portfolio = employee.getGithubLink();  
  
    if (employee.type === 'manager') {  
      portfolio = employee.getMBAProjects();  
    }  
  
    const data = {  
      expectedSalary,  
      experience,  
      portfolio  
    };  
  
    render(data);  
  });  
}
```

[↑ 返回顶部](#)

## 使用 Object.assign 设置默认对象

**不好的：**

```
const 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  
}
```

```
createMenu(menuConfig);
```

**好的：**

```
const 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
  // ...
}

createMenu(menuConfig);
```

[↑ 返回顶部](#)

## 不要使用标记位做为函数参数

标记位是告诉你的用户这个函数做了不只一件事情。函数应该只做一件事情。如果你的函数因为一个布尔值出现不同的代码路径，请拆分它们。

**不好的：**

```
function createFile(name, temp) {
  if (temp) {
    fs.create(`./temp/${name}`);
  } else {
    fs.create(name);
  }
}
```

好的：

```
function createFile(name) {
  fs.create(name);
}

function createTempFile(name) {
  createFile(`./temp/${name}`);
}
```

[↑ 返回顶部](#)

## 避免副作用

如果一个函数做了除接受一个值然后返回一个值或多个值之外的任何事情，它将会产生副作用，它可能是写入一个文件，修改一个全局变量，或者意外的把你所有的钱连接到一个陌生人那里。

现在在你的程序中确实偶尔需要副作用，就像上面的代码，你也许需要写入到一个文件，你需要做的是集中化你要做的事情，不要让多个函数或者类写入一个特定的文件，用一个服务来实现它，一个并且只有一个。

重点是避免这些常见的易犯的错误，比如在对象之间共享状态而不使用任何结构，使用任何地方都可以写入的可变的数据类型，没有集中化导致副作用。如果你能做到这些，那么你将比其它的码农大军更加幸福。

不好的：

```
// Global variable referenced by following function.
// 全局变量被下面的函数引用
// If we had another function that used this name, now it'd be a
// could break it.
// 如果有另一个函数使用这个 name，现在它应该是一个数组，这可能会出现错误
let name = 'Ryan McDermott';

function splitIntoFirstAndLastName() {
  name = name.split(' ');
}
```

```
splitIntoFirstAndLastName();

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

**好的：**

```
function splitIntoFirstAndLastName(name) {
  return name.split(' ');
}

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

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

[↑ 返回顶部](#)

## 不要写入全局函数

污染全局在 JavaScript 中是一个不好的做法，因为你可能会和另外一个类库冲突，你的 API 的用户

可能不够聪明，直到他们得到在生产环境得到一个异常。让我们来考虑这样一个例子：假设你要扩展

JavaScript 的原生 `Array`，添加一个可以显示两个数组的不同之处的 `diff` 方法，你可以在

`Array.prototype` 中写一个新的方法，但是它可能会和尝试做相同事情的其它类库发生冲突。如果有

另外一个类库仅仅使用 `diff` 方法来查找数组的第一个元素和最后一个元素之间的不同之处呢？这就是

为什么使用 ES2015/ES6 的类是一个更好的做法的原因，只要简单的扩展全局的 `Array` 即可。

**不好的：**

```
Array.prototype.diff = function diff(comparisonArray) {
  const hash = new Set(comparisonArray);
  return this.filter(elem => !hash.has(elem));
};
```

**好的：**

```
class SuperArray extends Array {
  diff(comparisonArray) {
    const hash = new Set(comparisonArray);
    return this.filter(elem => !hash.has(elem));
  }
}
```

[↑ 返回顶部](#)

## 函数式编程优于指令式编程

JavaScript 不是 Haskell 那种方式的函数式语言，但是它有它的函数式风格。函数式语言更加简洁

并且更容易进行测试，当你可以使用函数式编程风格时请尽情使用。

**不好的：**

```
const programmerOutput = [
  {
    name: 'Uncle Bobby',
    linesOfCode: 500
  }, {
    name: 'Suzie Q',
    linesOfCode: 1500
  }, {
    name: 'Jimmy Gosling',
    linesOfCode: 150
  }, {
    name: 'Gracie Hopper',
    linesOfCode: 1000
  }
];

let totalOutput = 0;

for (let i = 0; i < programmerOutput.length; i++) {
  totalOutput += programmerOutput[i].linesOfCode;
}
```

**好的：**

```
const programmerOutput = [
  {
    name: 'Uncle Bobby',
```



```

    linesOfCode: 500
  }, {
    name: 'Suzie Q',
    linesOfCode: 1500
  }, {
    name: 'Jimmy Gosling',
    linesOfCode: 150
  }, {
    name: 'Gracie Hopper',
    linesOfCode: 1000
  }
];

const totalOutput = programmerOutput
  .map((programmer) => programmer.linesOfCode)
  .reduce((acc, linesOfCode) => acc + linesOfCode, 0);

```

[↑ 返回顶部](#)

## 封装条件语句

不好的：

```

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

```

好的：

```

function shouldShowSpinner(fsm, listNode) {
  return fsm.state === 'fetching' && isEmpty(listNode);
}

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {
  // ...
}

```

[↑ 返回顶部](#)

## 避免负面条件

不好的：

```
function isDOMNodeNotPresent(node) {  
    // ...  
}  
  
if (!isDOMNodeNotPresent(node)) {  
    // ...  
}
```

**好的：**

```
function isDOMNodePresent(node) {  
    // ...  
}  
  
if (isDOMNodePresent(node)) {  
    // ...  
}
```

[↑ 返回顶部](#)

## 避免条件语句

这看起来似乎是一个不可能的任务。第一次听到这个时，多数人会说：“没有 `if` 语句还能期望我干啥呢”，答案是多数情况下你可以使用多态来完成同样的任务。第二个问题通常是“好了，那么做很棒，但是我为什么想要那样做呢”，答案是我们学到的上一条代码整洁之道的理念：一个函数应当只做一件事情。当你有使用 `if` 语句的类/函数是，你在告诉你的用户你的函数做了不止一件事情。记住：只做一件事情。

**不好的：**

```
class Airplane {  
    // ...  
    getCruisingAltitude() {  
        switch (this.type) {  
            case '777':  
                return this.getMaxAltitude() - this.getPassengerCount();  
            case 'Air Force One':  
                return this.getMaxAltitude();  
        }  
    }  
}
```

```
        case 'Cessna':
            return this.getMaxAltitude() - this.getFuelExpenditure()
        }
    }
}
```

**好的：**

```
class Airplane {
    // ...
}

class Boeing777 extends Airplane {
    // ...
    getCruisingAltitude() {
        return this.getMaxAltitude() - this.getPassengerCount();
    }
}

class AirForceOne extends Airplane {
    // ...
    getCruisingAltitude() {
        return this.getMaxAltitude();
    }
}

class Cessna extends Airplane {
    // ...
    getCruisingAltitude() {
        return this.getMaxAltitude() - this.getFuelExpenditure();
    }
}
```

[↑ 返回顶部](#)

## 避免类型检查 (part 1)

JavaScript 是无类型的，这意味着你的函数能接受任何类型的参数。但是有时又会被这种自由咬伤，于是又尝试在你的函数中做类型检查。有很多方式来避免这个，第一个要考虑的是一致的 API。

**不好的：**

```
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'));
  }
}
```

**好的：**

```
function travelToTexas(vehicle) {
  vehicle.move(this.currentLocation, new Location('texas'));
}
```

[↑ 返回顶部](#)

## 避免类型检查 (part 2)

如果你使用原始的字符串、整数和数组，并且你不能使用多态，但是你依然感觉到有类型检查的需要，你应该考虑使用 TypeScript。它是一个常规 JavaScript 的优秀替代品，因为它在标准的 JavaScript 语法之上为你提供静态类型。对常规 JavaScript 做人工类型检查的问题是需要大量的冗词来仿造类型安全而不缺失可读性。保持你的 JavaScript 简洁，编写良好的测试，并有良好的代码审阅，否则使用 TypeScript（就像我说的，它是一个伟大的替代品）来完成这些。

**不好的：**

```
function combine(val1, val2) {
  if (typeof val1 === 'number' && typeof val2 === 'number' ||
      typeof val1 === 'string' && typeof val2 === 'string') {
    return val1 + val2;
  }

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

**好的：**

```
function combine(val1, val2) {  
  return val1 + val2;  
}
```

[↑ 返回顶部](#)

## 不要过度优化

现代化浏览器运行时在幕后做大量的优化，在大多数的时间，做优化就是在浪费你的时间。 [这些是好的](#)

[资源](#)，用来

查看那些地方需要优化。为这些而优化，直到他们被修正。

不好的：

```
// On old browsers, each iteration with uncached `list.length` w  
// because of `list.length` recomputation. In modern browsers, t  
// 在旧的浏览器上，每次循环 `list.length` 都没有被缓存，会导致不必要的开销  
// 算 `list.length`。在现代化浏览器上，这个已经被优化了。  
for (let i = 0, len = list.length; i < len; i++) {  
  // ...  
}
```

好的：

```
for (let i = 0; i < list.length; i++) {  
  // ...  
}
```

[↑ 返回顶部](#)

## 移除僵尸代码

僵死代码和冗余代码同样糟糕。没有理由在代码库中保存它。如果它不会被调用，就删掉它。当你需要它时，它依然保存在版本历史记录中。

不好的：

```
function oldRequestModule(url) {
  // ...
}

function newRequestModule(url) {
  // ...
}

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

好的：

```
function newRequestModule(url) {
  // ...
}

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

[↑ 返回顶部](#)

## 对象和数据结构

### 使用 getters 和 setters

JavaScript 没有接口或类型，所以坚持这个模式是非常困难的，因为我们没有 `public` 和 `private`

关键字。正因为如此，使用 getters 和 setters 来访问对象上的数据比简单的在一个对象上查找属性

要好得多。“为什么？”你可能会问，好吧，原因请看下面的列表：

- 当你在获取一个对象属性的背后做更多的事情时，你不需要在代码库中查找和修改每一处访问；
- 使用 `set` 可以让添加验证变得容易；
- 封装内部实现；
- 使用 getting 和 setting 时，容易添加日志和错误处理；
- 继承这个类，你可以重写默认功能；
- 你可以延迟加载对象的属性，比如说从服务器获取。

**不好的：**

```
class BankAccount {
    constructor() {
        this.balance = 1000;
    }
}

const bankAccount = new BankAccount();

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

**好的：**

```
class BankAccount {
    constructor(balance = 1000) {
        this._balance = balance;
    }

    // It doesn't have to be prefixed with `get` or `set` to be a
    set balance(amount) {
        if (verifyIfAmountCanBeSetted(amount)) {
            this._balance = amount;
        }
    }

    get balance() {
        return this._balance;
    }

    verifyIfAmountCanBeSetted(val) {
        // ...
    }
}

const bankAccount = new BankAccount();

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

// Get balance
let balance = bankAccount.balance;
```

[↑ 返回顶部](#)

## 让对象拥有私有成员

这个可以通过闭包来实现（针对 ES5 或更低）。

不好的：

```
const Employee = function(name) {
  this.name = name;
};

Employee.prototype.getName = function getName() {
  return this.name;
};

const employee = new Employee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Employee
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Employee
```

好的：

```
const Employee = function (name) {
  this.getName = function getName() {
    return name;
  };
};

const employee = new Employee('John Doe');
console.log(`Employee name: ${employee.getName()}`); // Employee
delete employee.name;
console.log(`Employee name: ${employee.getName()}`); // Employee
```

[↑ 返回顶部](#)

## 类

### ES2015/ES6 类优先与 ES5 纯函数

很难为经典的 ES5 类创建可读的的继承、构造和方法定义。如果你需要继承（并且感到奇怪为啥你不需要），则优先用 ES2015/ES6 的类。不过，短小的函数优先于类，直到你



发现你需要更大并且更复杂的对象。

**不好的：**

```
const Animal = function(age) {
  if (!(this instanceof Animal)) {
    throw new Error('Instantiate Animal with `new`');
  }

  this.age = age;
};

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

const 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 liveBirth() {};

const 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 speak() {};
```

**好的：**

```
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() { /* ... */ }
}

```

[↑ 返回顶部](#)

## 使用方法链

这个模式在 JavaScript 中是非常有用的，并且你可以在许多类库比如 jQuery 和 Lodash 中见到。

它使你的代码变得富有表现力，并减少啰嗦。因为这个原因，我说，使用方法链然后再看看你的代码

会变得多么简洁。在你的类 / 方法中，简单的在每个方法的最后返回 `this`，然后你就能把这个类的其它方法链在一起。

**不好的：**

```

class Car {
  constructor() {
    this.make = 'Honda';
    this.model = 'Accord';
    this.color = 'white';
  }

  setMake(make) {

```

```

        this.make = make;
    }

    setModel(model) {
        this.model = model;
    }

    setColor(color) {
        this.color = color;
    }

    save() {
        console.log(this.make, this.model, this.color);
    }
}

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

```

**好的：**

```

class Car {
    constructor() {
        this.make = 'Honda';
        this.model = 'Accord';
        this.color = 'white';
    }

    setMake(make) {
        this.make = make;
        // 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);
  // NOTE: Returning this for chaining
  return this;
}
}

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

[↑ 返回顶部](#)

## 组合优先于继承

正如[设计模式四人帮](#)所述，如果可能，你应该优先使用组合而不是继承。有许多好的理由去使用继承，也有许多好的理由去使用组合。这个格言的重点是，如果你本能的观点是继承，那么请想一下组合能否更好的为你的问题建模。很多情况下它真的可以。

那么你也许会这样想，“我什么时候改使用继承？”这取决于你手上的问题，不过这儿有一个像样的列表说明什么时候继承比组合更好用：

1. 你的继承表示"是一个"的关系而不是"有一个"的关系（人类->动物 vs 用户->用户详情）；
2. 你可以重用来自基类的代码（人可以像所有动物一样行动）；
3. 你想通过基类对子类进行全局的修改（改变所有动物行动时的热量消耗）；

**不好的：**

```
class Employee {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }
}
```

```

    // ...
}

// 不好是因为雇员“有”税率数据, EmployeeTaxData 不是一个 Employee 类型。
class EmployeeTaxData extends Employee {
    constructor(ssn, salary) {
        super();
        this.ssn = ssn;
        this.salary = salary;
    }

    // ...
}

```

**好的：**

```

class EmployeeTaxData {
    constructor(ssn, salary) {
        this.ssn = ssn;
        this.salary = salary;
    }

    // ...
}

class Employee {
    constructor(name, email) {
        this.name = name;
        this.email = email;
    }

    setTaxData(ssn, salary) {
        this.taxData = new EmployeeTaxData(ssn, salary);
    }
    // ...
}

```

[↑ 返回顶部](#)

# SOLID

---

## 单一职责原则 (SRP)

正如代码整洁之道所述，“永远不要有超过一个理由来修改一个类”。给一个类塞满许多功能，就像你在航班上只能带一个行李箱一样，这样做的问题你的类不会有理想的内聚性，将会有太多的理由来对它进行修改。

最小化需要修改一个类的次数时很重要的，因为如果一个类拥有太多的功能，一旦你修改它的一小部分，将会很难弄清楚会对代码库中的其它模块造成什么影响。

**不好的：**

```
class UserSettings {
    constructor(user) {
        this.user = user;
    }

    changeSettings(settings) {
        if (this.verifyCredentials()) {
            // ...
        }
    }

    verifyCredentials() {
        // ...
    }
}
```

**好的：**

```
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()) {
      // ...
    }
  }
}

```

[↑ 返回顶部](#)

## 开闭原则 (OCP)

Bertrand Meyer 说过，“软件实体 (类，模块，函数等) 应该为扩展开放，但是为修改关闭。”这

是什么意思呢？这个原则基本上说明了你应该允许用户添加功能而不必修改现有的代码。

**不好的：**

```

class AjaxAdapter extends Adapter {
  constructor() {
    super();
    this.name = 'ajaxAdapter';
  }
}

class NodeAdapter extends Adapter {
  constructor() {
    super();
    this.name = 'nodeAdapter';
  }
}

class HttpRequester {
  constructor(adapter) {
    this.adapter = adapter;
  }

  fetch(url) {
    if (this.adapter.name === 'ajaxAdapter') {

```

```

        return makeAjaxCall(url).then((response) => {
            // transform response and return
        });
    } else if (this.adapter.name === 'httpNodeAdapter') {
        return makeHttpRequest(url).then((response) => {
            // transform response and return
        });
    }
}

function makeAjaxCall(url) {
    // request and return promise
}

function makeHttpRequest(url) {
    // request and return promise
}

```

**好的：**

```

class AjaxAdapter extends Adapter {
    constructor() {
        super();
        this.name = 'ajaxAdapter';
    }

    request(url) {
        // request and return promise
    }
}

class NodeAdapter extends Adapter {
    constructor() {
        super();
        this.name = 'nodeAdapter';
    }

    request(url) {
        // request and return promise
    }
}

class HttpRequister {
    constructor(adapter) {
        this.adapter = adapter;
    }
}

```



```
fetch(url) {  
  return this.adapter.request(url).then((response) => {  
    // transform response and return  
  });  
}
```

[↑ 返回顶部](#)

## 里氏代换原则 (LSP)

这是针对一个非常简单的里面的一个恐怖意图，它的正式定义是：“如果 S 是 T 的一个子类型，那么类型为 T 的对象可以被类型为 S 的对象替换（例如，类型为 S 的对象可作为类型为 T 的替代品）而不需要修改目标程序的期望性质（正确性、任务执行性等）。”这甚至是个恐怖的定义。

最好的解释是，如果你又一个基类和一个子类，那个基类和子类可以互换而不会产生不正确的结果。这可能还有些疑惑，让我们来看一下这个经典的正方形与矩形的例子。从数学上说，一个正方形是一个矩形，但是你用 "is-a" 的关系用继承来实现，你将很快遇到麻烦。

**不好的：**

```
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 {
  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);
    const area = rectangle.getArea(); // BAD: Will return 25 for
    rectangle.render(area);
  });
}

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

```

**好的：**

```

class Shape {
  setColor(color) {
    // ...
  }

  render(area) {
    // ...
  }
}

class Rectangle extends Shape {

```

```

    constructor(width, height) {
      super();
      this.width = width;
      this.height = height;
    }

    getArea() {
      return this.width * this.height;
    }
  }

  class Square extends Shape {
    constructor(length) {
      super();
      this.length = length;
    }

    getArea() {
      return this.length * this.length;
    }
  }

  function renderLargeShapes(shapes) {
    shapes.forEach((shape) => {
      const area = shape.getArea();
      shape.render(area);
    });
  }

  const shapes = [new Rectangle(4, 5), new Rectangle(4, 5), new Square(4, 5)];
  renderLargeShapes(shapes);

```

[↑ 返回顶部](#)

## 接口隔离原则 (ISP)

JavaScript 没有接口，所以这个原则不像其它语言那么严格。不过，对于 JavaScript 这种缺少类类型的语言来说，它依然是重要并且有意义的。

接口隔离原则说的是“客户端不应该强制依赖他们不需要的接口。”在 JavaScript 这种弱类型语言中，接口是隐式的契约。

在 JavaScript 中能比较好的说明这个原则的是一个类需要一个巨大的配置对象。不需要客户端去设置大

量的选项是有益的，因为多数情况下他们不需要全部的设置。让它们变成可选的有助于防止出现一个“胖接口”。

**不好的：**

```
class DOMTraverser {
  constructor(settings) {
    this.settings = settings;
    this.setup();
  }

  setup() {
    this.rootNode = this.settings.rootNode;
    this.animationModule.setup();
  }

  traverse() {
    // ...
  }
}

const $ = new DOMTraverser({
  rootNode: document.getElementsByTagName('body'),
  animationModule() {} // Most of the time, we won't need to ani
  // ...
});
```

**好的：**

```
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() {
    // ...
  }
}

const $ = new DOMTraverser({
  rootNode: document.getElementsByTagName('body'),
  options: {
    animationModule() {}
  }
});

```

[↑ 返回顶部](#)

## 依赖反转原则 (DIP)

这个原则阐述了两个重要的事情：

1. 高级模块不应该依赖于低级模块，两者都应该依赖与抽象；
2. 抽象不应当依赖于具体实现，具体实现应当依赖于抽象。

这个一开始会很难理解，但是如果你使用过 Angular.js，你应该已经看到过通过依赖注入来实现的这个原则，虽然他们不是相同的概念，依赖反转原则让高级模块远离低级模块的细节和创建，可以通过 DI 来实现。这样做的巨大益处是降低模块间的耦合。耦合是一个非常糟糕的开发模式，因为会导致代码难于重构。

如上所述，JavaScript 没有接口，所以被依赖的抽象是隐式契约。也就是说，一个对象/类的方法和属性直接暴露给另外一个对象/类。在下面的例子中，任何一个 Request 模块的隐式契约 `InventoryTracker` 将有一个 `requestItems` 方法。

**不好的：**

```

class InventoryRequester {
  constructor() {
    this.REQ_METHODS = ['HTTP'];

```

```

    }

    requestItem(item) {
        // ...
    }
}

class InventoryTracker {
    constructor(items) {
        this.items = items;

        // 不好的： 我们已经创建了一个对请求的具体实现的依赖， 我们只有一个 request
        // 赖一个请求方法 'request'
        this.requester = new InventoryRequester();
    }

    requestItems() {
        this.items.forEach((item) => {
            this.requester.requestItem(item);
        });
    }
}

const inventoryTracker = new InventoryTracker(['apples', 'banana', 'oranges']);
inventoryTracker.requestItems();

```

**好的：**

```

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) {
    // ...
  }
}

// 通过外部创建依赖项并将它们注入, 我们可以轻松的用一个崭新的使用 WebSocket
// 替换。
const inventoryTracker = new InventoryTracker(['apples', 'banana']);
inventoryTracker.requestItems();
```

[↑ 返回顶部](#)

## 测试

测试比发布更加重要。如果你没有测试或者测试不够充分，每次发布时你就不能确认没有破坏任何事情。

测试的量由你的团队决定，但是拥有 100% 的覆盖率(包括所有的语句和分支)是你为什么能达到高度自信

和内心的平静。这意味着需要一个额外的伟大的测试框架，也需要一个好的[覆盖率工具](#)。

没有理由不写测试。这里有[大量的优秀的 JS 测试框架](#)，选一个适合你的团队的即可。当为团队选择了测试框架之后，接下来的目标是为生产的每一个新的功能 / 模块编写测试。如果你倾向于测试驱动开发(TDD)，那就太棒了，但是要点是确认你在线上任何功能或者重构一个现有功能之前，达到了需要的目标覆盖率。

### 一个测试一个概念

不好的：

```
const assert = require('assert');
```

```
describe('MakeMomentJSGreatAgain', () => {
  it('handles date boundaries', () => {
    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);
  });
});
```

**好的：**

```
const assert = require('assert');

describe('MakeMomentJSGreatAgain', () => {
  it('handles 30-day months', () => {
    const date = new MakeMomentJSGreatAgain('1/1/2015');
    date.addDays(30);
    date.shouldEqual('1/31/2015');
  });

  it('handles leap year', () => {
    const date = new MakeMomentJSGreatAgain('2/1/2016');
    date.addDays(28);
    assert.equal('02/29/2016', date);
  });

  it('handles non-leap year', () => {
    const date = new MakeMomentJSGreatAgain('2/1/2015');
    date.addDays(28);
    assert.equal('03/01/2015', date);
  });
});
```

[↑ 返回顶部](#)



# 并发

---

## 使用 Promises, 不要使用回调

回调不够简洁，因为他们会产生过多的嵌套。在 ES2015/ES6 中，Promises 已经是内置的全局类型了，使用它们吧！

**不好的：**

```
require('request').get('https://en.wikipedia.org/wiki/Robert_Cec
  if (requestErr) {
    console.error(requestErr);
  } else {
    require('fs').writeFile('article.html', response.body, (writ
      if (writeErr) {
        console.error(writeErr);
      } else {
        console.log('File written');
      }
    });
  }
});
```

**好的：**

```
require('request-promise').get('https://en.wikipedia.org/wiki/Ro
  .then((response) => {
    return require('fs-promise').writeFile('article.html', respo
  })
  .then(() => {
    console.log('File written');
  })
  .catch((err) => {
    console.error(err);
  });
```

[↑ 返回顶部](#)

## Async/Await 比 Promises 更加简洁

Promises 是回调的一个非常简洁的替代品，但是 ES2017/ES8 带来的 `async` 和 `await` 提供了一个更加简洁的解决方案。你需要的只是一个前缀为 `async` 关键字的函数，接下来就可以不需要 `then` 函数链来编写逻辑了。如果你能使用 ES2017/ES8 的高级功能的话，今天就使用它吧！

**不好的：**

```
require('request-promise').get('https://en.wikipedia.org/wiki/Ro')
  .then((response) => {
    return require('fs-promise').writeFile('article.html', respo
  })
  .then(() => {
    console.log('File written');
  })
  .catch((err) => {
    console.error(err);
  });
```

**好的：**

```
async function getCleanCodeArticle() {
  try {
    const response = await require('request-promise').get('https
    await require('fs-promise').writeFile('article.html', respon
    console.log('File written');
  } catch(err) {
    console.error(err);
  }
}
```

[↑ 返回顶部](#)

## 错误处理

---

抛出错误是一件好事情！他们意味着当你的程序有错时运行时可以成功确认，并且通过停止执行当前堆栈

上的函数来让你知道，结束当前进程（在 Node 中），在控制台中用一个堆栈跟踪提示你。

## 不要忽略捕捉到的错误

对捕捉到的错误不做任何处理不能给你修复错误或者响应错误的能力。向控制台记录错误（`console.log`）也不怎么好，因为往往会丢失在海量的控制台输出中。如果你把任意一段代码用 `try/catch` 包装那就意味着你想到这里可能会错，因此你应该有个修复计划，或者当错误发生时有一个代码路径。

不好的：

```
try {
  functionThatMightThrow();
} catch (error) {
  console.log(error);
}
```

好的：

```
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 promise

与你不应忽略来自 `try/catch` 的错误的原因相同。

不好的：

```
getdata()
  .then((data) => {
```

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

**好的：**

```
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!
});
```

[↑ 返回顶部](#)

## 格式化

格式化是主观的。就像其它规则一样，没有必须让你遵守的硬性规则。重点是不要因为格式去争论，这里有[大量的工具](#)来自动格式化，使用其中的一个即可！因为做为工程师去争论格式化就是在浪费时间和金钱。

针对自动格式化工具不能涵盖的问题（缩进、制表符还是空格、双引号还是单引号等），这里有一些指南。

## 使用一致的大小写

JavaScript 是无类型的，所以大小写告诉你关于你的变量、函数等的很多事情。这些规则是主观的，所以你的团队可以选择他们想要的。重点是，不管你们选择了什么，保持一致。

**不好的：**

```
const DAYS_IN_WEEK = 7;
const daysInMonth = 30;

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

function eraseDatabase() {}
function restore_database() {}

class animal {}
class Alpaca {}
```

**好的：**

```
const DAYS_IN_WEEK = 7;
const DAYS_IN_MONTH = 30;

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

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

class Animal {}
class Alpaca {}
```

[↑ 返回顶部](#)

## 函数的调用方与被调用方应该靠近

如果一个函数调用另一个，则在代码中这两个函数的竖直位置应该靠近。理想情况下，保持被调用函数在被调用函数的正上方。我们倾向于从上到下阅读代码，就像读一章报纸。由于这个原因，保持你的代码可以按照这种方式阅读。

**不好的：**

```
class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }
}
```

```

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

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

getPeerReviews() {
  const peers = this.lookupPeers();
  // ...
}

perfReview() {
  this.getPeerReviews();
  this.getManagerReview();
  this.getSelfReview();
}

getManagerReview() {
  const manager = this.lookupManager();
}

getSelfReview() {
  // ...
}
}

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

```

**好的：**

```

class PerformanceReview {
  constructor(employee) {
    this.employee = employee;
  }

  perfReview() {
    this.getPeerReviews();
    this.getManagerReview();
    this.getSelfReview();
  }

  getPeerReviews() {
    const peers = this.lookupPeers();
    // ...
  }
}

```

```

    }

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

    getManagerReview() {
        const manager = this.lookupManager();
    }

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

    getSelfReview() {
        // ...
    }
}

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

```

[↑ 返回顶部](#)

## 注释

---

### 仅仅对包含复杂业务逻辑的东西进行注释

注释是代码的辩解，不是要求。多数情况下，好的代码就是文档。

**不好的：**

```

function hashIt(data) {
    // The hash
    let hash = 0;

    // Length of string
    const length = data.length;

    // Loop through every character in data
    for (let i = 0; i < length; i++) {
        // Get character code.
        const char = data.charCodeAt(i);
        // Make the hash
        hash = ((hash << 5) - hash) + char;
    }
}

```

```
    // Convert to 32-bit integer
    hash &= hash;
  }
}
```

**好的：**

```
function hashIt(data) {
  let hash = 0;
  const length = data.length;

  for (let i = 0; i < length; i++) {
    const char = data.charCodeAt(i);
    hash = ((hash << 5) - hash) + char;

    // Convert to 32-bit integer
    hash &= hash;
  }
}
```

[↑ 返回顶部](#)

## 不要在代码库中保存注释掉的代码

因为有版本控制，把旧的代码留在历史记录即可。

**不好的：**

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

**好的：**

```
doStuff();
```

[↑ 返回顶部](#)



## 不要有日志式的注释

记住，使用版本控制！不需要僵尸代码，注释掉的代码，尤其是日志式的注释。使用 `git log` 来获取历史记录。

不好的：

```
/**
 * 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;
}
```

好的：

```
function combine(a, b) {
  return a + b;
}
```

[↑ 返回顶部](#)

## 避免占位符

它们仅仅添加了干扰。让函数和变量名称与合适的缩进和格式化为你的代码提供视觉结构。

不好的：

```
////////////////////////////////////
// Scope Model Instantiation
////////////////////////////////////
$scope.model = {
  menu: 'foo',
  nav: 'bar'
};

////////////////////////////////////
```

```
// Action setup
////////////////////////////////////
const actions = function() {
  // ...
};
```

好的：

```
$scope.model = {
  menu: 'foo',
  nav: 'bar'
};







const actions = function() {
  // ...
};
```

[↑ 返回顶部](#)

## Translation

---

This is also available in other languages:

-  **Brazilian Portuguese:** [fesnt/clean-code-javascript](#)
-  **Chinese:**
  - [alivebao/clean-code-js](#)
  - [beginor/clean-code-js](#)
-  **German:** [marcbruederlin/clean-code-javascript](#)
-  **Korean:** [qkraudghgh/clean-code-javascript-ko](#)
-  **Russian:**
  - [BoryaMogila/clean-code-javascript-ru/](#)
  - [maksugr/clean-code-javascript](#)
-  **Vietnamese:** [hienvd/clean-code-javascript/](#)

[↑ 返回顶部](#)