

变量、函数、对象和数据结构、类、单一原则、测试、并发性、错误处理、格式、注释

翻译

变量

更加可读

**Bad:**

const yyyymmdstr = moment().format("YYYY/MM/DD");

**Good:**

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

同类型用相同的单词表示

**Bad:**

getUserInfo();

getClientData();

getCustomerRecord();

**Good:**

getUser();

将常量提出来

**Bad:**

// What the heck is 86400000 for?

setTimeout(blastOff, 86400000);

**Good:**

// Declare them as capitalized named constants.

const MILLISECONDS\_IN\_A\_DAY = 86400000;

setTimeout(blastOff, MILLISECONDS\_IN\_A\_DAY);

入参变量的解释清晰，多点变量死不了

**Bad:**

const address = "One Infinite Loop, Cupertino 95014";

const cityZipCodeRegex = /^[^,**\\**]+[,**\\**\s]+(.+?)\s\*(\d{5})?$/;

saveCityZipCode(

address.match(cityZipCodeRegex)[1],

address.match(cityZipCodeRegex)[2]

);

**Good:**

const address = "One Infinite Loop, Cupertino 95014";

const cityZipCodeRegex = /^[^,**\\**]+[,**\\**\s]+(.+?)\s\*(\d{5})?$/;

const [, city, zipCode] = address.match(cityZipCodeRegex) || [];

saveCityZipCode(city, zipCode);

明显的单词表现的效果更明显

**Bad:**

const locations = ["Austin", "New York", "San Francisco"];

locations.forEach(l => {

doStuff();

doSomeOtherStuff();

// ...

// ...

// ...

// Wait, what is `l` for again?

dispatch(l);

});

**Good:**

const locations = ["Austin", "New York", "San Francisco"];

locations.forEach(location => {

doStuff();

doSomeOtherStuff();

// ...

// ...

// ...

dispatch(location);

});

不要加入无用的上下文，除非。。。

**Bad:**

const Car = {

carMake: "Honda",

carModel: "Accord",

carColor: "Blue"

};

function paintCar(car) {

car.carColor = "Red";

}

**Good:**

const Car = {

make: "Honda",

model: "Accord",

color: "Blue"

};

function paintCar(car) {

car.color = "Red";

}

注意假值判断，参数使用默认值处理

**Bad:**

function createMicrobrewery(name) {

const breweryName = name || "Hipster Brew Co.";

// ...

}

**Good:**

function createMicrobrewery(name = "Hipster Brew Co.") {

// ...

}

函数

函数的参数尽可能少

**Bad:**

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

// ...

}

**Good:**

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

// ...

}

createMenu({

title: "Foo",

body: "Bar",

buttonText: "Baz",

cancellable: true

});

函数只做一件事（印度人的代码）

**Bad:**

function emailClients(clients) {

clients.forEach(client => {

const clientRecord = database.lookup(client);

if (clientRecord.isActive()) {

email(client);

}

});

}

**Good:**

function emailActiveClients(clients) {

clients.filter(isActiveClient).forEach(email);

}

function isActiveClient(client) {

const clientRecord = database.lookup(client);

return clientRecord.isActive();

}

函数的命名要提现它做啥（最高境界，命名即注释）

**Bad:**

function addToDate(date, month) {

// ...

}

const date = new Date();

// It's hard to tell from the function name what is added

addToDate(date, 1);

**Good:**

function addMonthToDate(month, date) {

// ...

}

const date = new Date();

addMonthToDate(1, date);

一个函数抽象一个功能（同上面）

**Bad:**

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...

});

}

**Good:**

function parseBetterJSAlternative(code) {

const tokens = tokenize(code);

const syntaxTree = parse(tokens);

syntaxTree.forEach(node => {

// parse...

});

}

function tokenize(code) {

const REGEXES = [

// ...

];

const statements = code.split(" ");

const tokens = [];

REGEXES.forEach(REGEX => {

statements.forEach(statement => {

tokens.push(/\* ... \*/);

});

});

return tokens;

}

function parse(tokens) {

const syntaxTree = [];

tokens.forEach(token => {

syntaxTree.push(/\* ... \*/);

});

return syntaxTree;

}

去除重复的代码

**Bad:**

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.getMBAProjects();

const data = {

expectedSalary,

experience,

portfolio

};

render(data);

});

}

**Good:**

function showEmployeeList(employees) {

employees.forEach(employee => {

const expectedSalary = employee.calculateExpectedSalary();

const experience = employee.getExperience();

const data = {

expectedSalary,

experience

};

switch (employee.type) {

case "manager":

data.portfolio = employee.getMBAProjects();

break;

case "developer":

data.githubLink = employee.getGithubLink();

break;

}

render(data);

});

}

对象的默认值使用assign 或者 …操作符

**Bad:**

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.cancellable : true;

}

createMenu(menuConfig);

**Good:**

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: "Send", cancellable: true}

// ...

}

createMenu(menuConfig);

不要使用一个标记来让函数提供多个功能

**Bad:**

function createFile(name, temp) {

if (temp) {

fs.create(`./temp/${name}`);

} else {

fs.create(name);

}

}

**Good:**

function createFile(name) {

fs.create(name);

}

function createTempFile(name) {

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

}

避免函数的副作用（1）

**Bad:**

// 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.

let name = "Ryan McDermott";

function splitIntoFirstAndLastName() {

name = name.split(" ");

}

splitIntoFirstAndLastName();

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

**Good:**

function splitIntoFirstAndLastName(name) {

return name.split(" ");

}

const name = "Ryan McDermott";

const newName = splitIntoFirstAndLastName(name);

console.log(name); // 'Ryan McDermott';

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

避免函数的副作用（2），引用传值

**Bad:**

const addItemToCart = (cart, item) => {

cart.push({ item, date: Date.now() });

};

**Good:**

const addItemToCart = (cart, item) => {

return [...cart, { item, date: Date.now() }];

};

减少全局变量

**Bad:**

Array.prototype.diff = function diff(comparisonArray) {

const hash = new Set(comparisonArray);

return this.filter(elem => !hash.has(elem));

};

**Good:**

class SuperArray extends Array {

diff(comparisonArray) {

const hash = new Set(comparisonArray);

return this.filter(elem => !hash.has(elem));

}

}

优先使用函数式编程而不是命令式编程（更容易被单元测试）

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;

}

**Good:**

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.reduce(

(totalLines, output) => totalLines + output.linesOfCode,

0

);

封装条件

**Bad:**

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

// ...

}

**Good:**

function shouldShowSpinner(fsm, listNode) {

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

}

if (shouldShowSpinner(fsmInstance, listNodeInstance)) {

// ...

}

避免否条件判断

**Bad:**

function isDOMNodeNotPresent(node) {

// ...

}

if (!isDOMNodeNotPresent(node)) {

// ...

}

**Good:**

function isDOMNodePresent(node) {

// ...

}

if (isDOMNodePresent(node)) {

// ...

}

避免条件分支（这个很难，但不是不能优化）

**Bad:**

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();

}

}

}

**Good:**

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();

}

}

避免类型判断（1）

**Bad:**

function travelToTexas(vehicle) {

if (vehicle instanceof Bicycle) {

vehicle.pedal(this.currentLocation, new Location("texas"));

} else if (vehicle instanceof Car) {

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

}

}

**Good:**

function travelToTexas(vehicle) {

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

}

避免类型判断（2）如果使用原始值操作，不要考虑太多情况

**Bad:**

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");

}

**Good:**

function combine(val1, val2) {

return val1 + val2;

}

不要过度优化（同上面多写点变量，一般情况下清晰比性能更重要）

**Bad:**

// On old browsers, each iteration with uncached `list.length` would be costly

// because of `list.length` recomputation. In modern browsers, this is optimized.

for (let i = 0, len = list.length; i < len; i++) {

// ...

}

**Good:**

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

// ...

}

删除老旧代码

**Bad:**

function oldRequestModule(url) {

// ...

}

function newRequestModule(url) {

// ...

}

const req = newRequestModule;

inventoryTracker("apples", req, "www.inventory-awesome.io");

**Good:**

function newRequestModule(url) {

// ...

}

const req = newRequestModule;

inventoryTracker("apples", req, "www.inventory-awesome.io");

对象和数据结构

用getters 和 setters 来操作对象属性

**Bad:**

function makeBankAccount() {

// ...

return {

balance: 0

// ...

};

}

const account = makeBankAccount();

account.balance = 100;

**Good:**

function makeBankAccount() {

// this one is private

let balance = 0;

// a "getter", made public via the returned object below

function getBalance() {

return balance;

}

// a "setter", made public via the returned object below

function setBalance(amount) {

// ... validate before updating the balance

balance = amount;

}

return {

// ...

getBalance,

setBalance

};

}

const account = makeBankAccount();

account.setBalance(100);

使对象拥有私有成员，oop思路，js一直支持得不好，我们可以使用闭包来做，或TS或node12以上

**Bad:**

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 name: John Doe

delete employee.name;

console.log(`Employee name: ${employee.getName()}`); // Employee name: undefined

**Good:**

function makeEmployee(name) {

return {

getName() {

return name;

}

};

}

const employee = makeEmployee("John Doe");

console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe

delete employee.name;

console.log(`Employee name: ${employee.getName()}`); // Employee name: John Doe

js类

使用es2016/es6 代替es5

**Bad:**

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() {};

**Good:**

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() {

/\* ... \*/

}

}

使用方法链（核心是使返回的数值可以连续的操作，更好的被理解）

**Bad:**

class Car {

constructor(make, model, color) {

this.make = make;

this.model = model;

this.color = color;

}

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("Ford", "F-150", "red");

car.setColor("pink");

car.save();

**Good:**

class Car {

constructor(make, model, color) {

this.make = make;

this.model = model;

this.color = color;

}

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("Ford", "F-150", "red").setColor("pink").save();

用组合代替继承（搞清关系而不是 有关系）

 an "is-a" relationship and not a "has-a" relationship (Human->Animal vs. User->UserDetails)

**Bad:**

class Employee {

constructor(name, email) {

this.name = name;

this.email = email;

}

// ...

}

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

class EmployeeTaxData extends Employee {

constructor(ssn, salary) {

super();

this.ssn = ssn;

this.salary = salary;

}

// ...

}

**Good:**

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);

}

// ...

}

一些设计原则SOLID

Single Responsibility Principle：单一职责原则

Open Closed Principle：开闭原则

Liskov Substitution Principle：里氏替换原则

Law of Demeter：迪米特法则，最少知道原则

Interface Segregation Principle：接口隔离原则

Dependence Inversion Principle：依赖倒置原则

### Single Responsibility Principle 单一责任准则 (SRP)

**Bad:**

class UserSettings {

constructor(user) {

this.user = user;

}

changeSettings(settings) {

if (this.verifyCredentials()) {

// ...

}

}

verifyCredentials() {

// ...

}

}

**Good:**

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()) {

// ...

}

}

}

### Open/Closed Principle (OCP)

允许用户加入代码而不改变已经存在的代码

**Bad:**

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 === "nodeAdapter") {

return makeHttpCall(url).then(response => {

// transform response and return

});

}

}

}

function makeAjaxCall(url) {

// request and return promise

}

function makeHttpCall(url) {

// request and return promise

}

**Good:**

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 HttpRequester {

constructor(adapter) {

this.adapter = adapter;

}

fetch(url) {

return this.adapter.request(url).then(response => {

// transform response and return

});

}

}

### Liskov Substitution Principle (LSP) Liskov（里氏）替换原则

说人话：

应当尽量从抽象类继承，而不从具体类继承。

一般而言,如果有两个具体类A、B有继承关系，那么一个最简单的修改方案是建立一个抽象类C，然后让类A和B成为抽象类C的子类

**Bad:**

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: Returns 25 for Square. Should be 20.

rectangle.render(area);

});

}

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

renderLargeRectangles(rectangles);

**Good:**

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(5)];

renderLargeShapes(shapes);

### 接口隔离原则Interface Segregation Principle (ISP)

**Bad:**

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 animate when traversing.

// ...

});

**Good:**

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() {}

}

});

### 依赖倒置原则Dependency Inversion Principle (DIP)

**1 高层级的模块不应该依赖于低层次的模块，它应该依赖于低层次模块的抽象**

**2 抽象不应该依赖于具体，具体应该依赖于抽象**

**Bad:**

class InventoryRequester {

constructor() {

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

}

requestItem(item) {

// ...

}

}

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);

});

}

}

const inventoryTracker = new InventoryTracker(["apples", "bananas"]);

inventoryTracker.requestItems();

**Good:**

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.

const inventoryTracker = new InventoryTracker(

["apples", "bananas"],

new InventoryRequesterV2()

);

inventoryTracker.requestItems();

测试

每个测试足够单一

**Bad:**

import assert from "assert";

describe("MakeMomentJSGreatAgain", () => {

it("handles date boundaries", () => {

let date;

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

date.addDays(30);

assert.equal("1/31/2015", date);

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);

});

});

**Good:**

import assert from "assert";

describe("MakeMomentJSGreatAgain", () => {

it("handles 30-day months", () => {

const date = new MakeMomentJSGreatAgain("1/1/2015");

date.addDays(30);

assert.equal("1/31/2015", date);

});

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);

});

});

并发

### Use Promises, not callbacks

### Async/Await 比 Promises 清晰

### 错误处理

### 不能忽略抛出的错误

**Bad:**

try {

functionThatMightThrow();

} catch (error) {

console.log(error);

}

**Good:**

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!

}

Promise错误的捕获

**Bad:**

getdata()

.then(data => {

functionThatMightThrow(data);

})

.catch(error => {

console.log(error);

});

**Good:**

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!

});

代码格式

使用一致的大小写

**Bad:**

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 {}

**Good:**

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 {}

函数的调用方和被调用方尽可能靠近

如果函数调用另一个函数，请在源文件中保持这些函数垂直关闭。理想情况下，将调用者保持在被调用者的正上方

**Bad:**

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(employee);

review.perfReview();

**Good:**

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();

注释

只注释具有复杂业务逻辑的代码

好的注解就是代码本身

**Bad:**

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;

}

}

**Good:**

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;

}

}

不要将注释掉的代码留在代码库中

**Bad:**

doStuff();

// doOtherStuff();

// doSomeMoreStuff();

// doSoMuchStuff();

**Good:**

doStuff();

不要有日志形式的代码注解

**Bad:**

/\*\*

\* 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;

}

**Good:**

function combine(a, b) {

return a + b;

}

避免花哨的标记

**Bad:**

////////////////////////////////////////////////////////////////////////////////

// Scope Model Instantiation

////////////////////////////////////////////////////////////////////////////////

$scope.model = {

menu: "foo",

nav: "bar"

};

////////////////////////////////////////////////////////////////////////////////

// Action setup

////////////////////////////////////////////////////////////////////////////////

const actions = function() {

// ...

};

**Good:**

$scope.model = {

menu: "foo",

nav: "bar"

};

const actions = function() {

// ...

};