**代码风格**

**1 文件**

**【建议】 JavaScript 文件使用无 BOM 的 UTF-8 编码。**

解释：

UTF-8 编码具有更广泛的适应性。BOM 在使用程序或工具处理文件时可能造成不必要的干扰。

**【建议】 在文件结尾处，保留一个空行。**

**2 结构**

**2.1 缩进**

**【强制】 使用 4 个空格做为一个缩进层级，不允许使用 2 个空格 或 tab 字符。**

**【强制】 switch 下的 case 和 default 必须增加一个缩进层级。**

示例：

// good

switch (variable) {

case '1':

// do...

break;

case '2':

// do...

break;

default:

// do...

}

// bad

switch (variable) {

case '1':

// do...

break;

case '2':

// do...

break;

default:

// do...

}

**2.2 空格**

**【强制】 二元运算符两侧必须有一个空格，一元运算符与操作对象之间不允许有空格。**

示例：

var a = !arr.length;

a++;

a = b + c;

**【强制】 用作代码块起始的左花括号 { 前必须有一个空格。**

示例：

// good

if (condition) {

}

while (condition) {

}

function funcName() {

}

// bad

if (condition){

}

while (condition){

}

function funcName(){

}

**【强制】 if / else / for / while / function / switch / do / try / catch / finally 关键字后，必须有一个空格。**

示例：

// good

if (condition) {

}

while (condition) {

}

(function () {

})();

// bad

if(condition) {

}

while(condition) {

}

(function() {

})();

**【强制】 在对象创建时，属性中的 : 之后必须有空格，: 之前不允许有空格。**

示例：

// good

var obj = {

a: 1,

b: 2,

c: 3

};

// bad

var obj = {

a : 1,

b:2,

c :3

};

**【强制】 函数声明、具名函数表达式、函数调用中，函数名和 ( 之间不允许有空格。**

示例：

// good

function funcName() {

}

var funcName = function funcName() {

};

funcName();

// bad

function funcName () {

}

var funcName = function funcName () {

};

funcName ();

**【强制】 , 和 ; 前不允许有空格。**

示例：

// good

callFunc(a, b);

// bad

callFunc(a , b) ;

**【强制】 在函数调用、函数声明、括号表达式、属性访问、if / for / while / switch / catch 等语句中，() 和 [] 内紧贴括号部分不允许有空格。**

示例：

// good

callFunc(param1, param2, param3);

save(this.list[this.indexes[i]]);

needIncream && (variable += increament);

if (num > list.length) {

}

while (len--) {

}

// bad

callFunc( param1, param2, param3 );

save( this.list[ this.indexes[ i ] ] );

needIncreament && ( variable += increament );

if ( num > list.length ) {

}

while ( len-- ) {

}

**【强制】 单行声明的数组与对象，如果包含元素，{} 和 [] 内紧贴括号部分不允许包含空格。**

解释：

声明包含元素的数组与对象，只有当内部元素的形式较为简单时，才允许写在一行。元素复杂的情况，还是应该换行书写。

示例：

// good

var arr1 = [];

var arr2 = [1, 2, 3];

var obj1 = {};

var obj2 = {name: 'obj'};

var obj3 = {

name: 'obj',

age: 20,

sex: 1

};

// bad

var arr1 = [ ];

var arr2 = [ 1, 2, 3 ];

var obj1 = { };

var obj2 = { name: 'obj' };

var obj3 = {name: 'obj', age: 20, sex: 1};

**【强制】 行尾不得有多余的空格。**

**2.3 换行**

**【强制】 每个独立语句结束后必须换行。**

**【强制】 每行不得超过 120 个字符。**

解释：

超长的不可分割的代码允许例外，比如复杂的正则表达式。长字符串不在例外之列。

**【强制】 运算符处换行时，运算符必须在新行的行首。**

示例：

// good

if (user.isAuthenticated()

&& user.isInRole('admin')

&& user.hasAuthority('add-admin')

|| user.hasAuthority('delete-admin')

) {

// Code

}

var result = number1 + number2 + number3

+ number4 + number5;

// bad

if (user.isAuthenticated() &&

user.isInRole('admin') &&

user.hasAuthority('add-admin') ||

user.hasAuthority('delete-admin')) {

// Code

}

var result = number1 + number2 + number3 +

number4 + number5;

**【强制】 在函数声明、函数表达式、函数调用、对象创建、数组创建、for语句等场景中，不允许在 , 或 ; 前换行。**

示例：

// good

var obj = {

a: 1,

b: 2,

c: 3

};

foo(

aVeryVeryLongArgument,

anotherVeryLongArgument,

callback

);

// bad

var obj = {

a: 1

, b: 2

, c: 3

};

foo(

aVeryVeryLongArgument

, anotherVeryLongArgument

, callback

);

**【建议】 不同行为或逻辑的语句集，使用空行隔开，更易阅读。**

示例：

// 仅为按逻辑换行的示例，不代表setStyle的最优实现

function setStyle(element, property, value) {

if (element == null) {

return;

}

element.style[property] = value;

}

**【建议】 在语句的行长度超过 120 时，根据逻辑条件合理缩进。**

示例：

// 较复杂的逻辑条件组合，将每个条件独立一行，逻辑运算符放置在行首进行分隔，或将部分逻辑按逻辑组合进行分隔。

// 建议最终将右括号 ) 与左大括号 { 放在独立一行，保证与 if 内语句块能容易视觉辨识。

if (user.isAuthenticated()

&& user.isInRole('admin')

&& user.hasAuthority('add-admin')

|| user.hasAuthority('delete-admin')

) {

// Code

}

// 按一定长度截断字符串，并使用 + 运算符进行连接。

// 分隔字符串尽量按语义进行，如不要在一个完整的名词中间断开。

// 特别的，对于HTML片段的拼接，通过缩进，保持和HTML相同的结构。

var html = '' // 此处用一个空字符串，以便整个HTML片段都在新行严格对齐

+ '<article>'

+ '<h1>Title here</h1>'

+ '<p>This is a paragraph</p>'

+ '<footer>Complete</footer>'

+ '</article>';

// 也可使用数组来进行拼接，相对 + 更容易调整缩进。

var html = [

'<article>',

'<h1>Title here</h1>',

'<p>This is a paragraph</p>',

'<footer>Complete</footer>',

'</article>'

];

html = html.join('');

// 当参数过多时，将每个参数独立写在一行上，并将结束的右括号 ) 独立一行。

// 所有参数必须增加一个缩进。

foo(

aVeryVeryLongArgument,

anotherVeryLongArgument,

callback

);

// 也可以按逻辑对参数进行组合。

// 最经典的是baidu.format函数，调用时将参数分为“模板”和“数据”两块

baidu.format(

dateFormatTemplate,

year, month, date, hour, minute, second

);

// 当函数调用时，如果有一个或以上参数跨越多行，应当每一个参数独立一行。

// 这通常出现在匿名函数或者对象初始化等作为参数时，如setTimeout函数等。

setTimeout(

function () {

alert('hello');

},

200

);

order.data.read(

'id=' + me.model.id,

function (data) {

me.attchToModel(data.result);

callback();

},

300

);

// 链式调用较长时采用缩进进行调整。

$('#items')

.find('.selected')

.highlight()

.end();

// 三元运算符由3部分组成，因此其换行应当根据每个部分的长度不同，形成不同的情况。

var result = thisIsAVeryVeryLongCondition

? resultA : resultB;

var result = condition

? thisIsAVeryVeryLongResult

: resultB;

// 数组和对象初始化的混用，严格按照每个对象的 { 和结束 } 在独立一行的风格书写。

var array = [

{

// ...

},

{

// ...

}

];

**【建议】 对于 if...else...、try...catch...finally 等语句，推荐使用在 } 号后添加一个换行 的风格，使代码层次结构更清晰，阅读性更好。**

示例：

if (condition) {

// some statements;

}

else {

// some statements;

}

try {

// some statements;

}

catch (ex) {

// some statements;

}

**2.4 语句**

**【强制】 不得省略语句结束的分号。**

**【强制】 在 if / else / for / do / while 语句中，即使只有一行，也不得省略块 {...}。**

示例：

// good

if (condition) {

callFunc();

}

// bad

if (condition) callFunc();

if (condition)

callFunc();

**【强制】 函数定义结束不允许添加分号。**

示例：

// good

function funcName() {

}

// bad

function funcName() {

};

// 如果是函数表达式，分号是不允许省略的。

var funcName = function () {

};

**【强制】 IIFE 必须在函数表达式外添加 (，非 IIFE 不得在函数表达式外添加 (。**

解释：

IIFE = Immediately-Invoked Function Expression.

额外的 ( 能够让代码在阅读的一开始就能判断函数是否立即被调用，进而明白接下来代码的用途。而不是一直拖到底部才恍然大悟。

示例：

// good

var task = (function () {

// Code

return result;

})();

var func = function () {

};

// bad

var task = function () {

// Code

return result;

}();

var func = (function () {

});

**3 命名**

**【强制】 变量 使用 Camel命名法。**

示例：

var loadingModules = {};

**【强制】 常量 使用 全部字母大写，单词间下划线分隔 的命名方式。**

示例：

var HTML\_ENTITY = {};

**【强制】 函数 使用 Camel命名法。**

示例：

function stringFormat(source) {

}

**【强制】 函数的 参数 使用 Camel命名法。**

示例：

function hear(theBells) {

}

**【强制】 类 使用 Pascal命名法。**

示例：

function TextNode(options) {

}

**【强制】 类的 方法 / 属性 使用 Camel命名法。**

示例：

function TextNode(value, engine) {

this.value = value;

this.engine = engine;

}

TextNode.prototype.clone = function () {

return this;

};

**【强制】 枚举变量 使用 Pascal命名法，枚举的属性 使用 全部字母大写，单词间下划线分隔 的命名方式。**

示例：

var TargetState = {

READING: 1,

READED: 2,

APPLIED: 3,

READY: 4

};

**【强制】 命名空间 使用 Camel命名法。**

示例：

equipments.heavyWeapons = {};

**【强制】 由多个单词组成的缩写词，在命名中，根据当前命名法和出现的位置，所有字母的大小写与首字母的大小写保持一致。**

示例：

function XMLParser() {

}

function insertHTML(element, html) {

}

var httpRequest = new HTTPRequest();

**【强制】 类名 使用 名词。**

示例：

function Engine(options) {

}

**【建议】 函数名 使用 动宾短语。**

示例：

function getStyle(element) {

}

**【建议】 boolean 类型的变量使用 is 或 has 开头。**

示例：

var isReady = false;

var hasMoreCommands = false;

**【建议】 Promise对象 用 动宾短语的进行时 表达。**

示例：

var loadingData = ajax.get('url');

loadingData.then(callback);

**4 注释**

**4.1 单行注释**

**【强制】 必须独占一行。// 后跟一个空格，缩进与下一行被注释说明的代码一致。**

**4.2 多行注释**

**【建议】 避免使用 /\*...\*/ 这样的多行注释。有多行注释内容时，使用多个单行注释。**

**4.3 文档化注释**

**【强制】 为了便于代码阅读和自文档化，以下内容必须包含以 /\*\*...\*/ 形式的块注释中。**

解释：

1. 文件
2. namespace
3. 类
4. 函数或方法
5. 类属性
6. 事件
7. 全局变量
8. 常量

**【强制】 文档注释前必须空一行。**

**【建议】 自文档化的文档说明 what，而不是 how。**

**4.4 类型定义**

**【强制】 类型定义都是以{开始, 以}结束。**

解释：

常用类型如：{string}, {number}, {boolean}, {Object}, {Function}, {RegExp}, {Array}, {Date}。

类型不仅局限于内置的类型，也可以是自定义的类型。比如定义了一个类 Developer，就可以使用它来定义一个参数和返回值的类型。

**【强制】 对于基本类型 {string}, {number}, {boolean}，首字母必须小写。**

|  |  |  |
| --- | --- | --- |
| **类型定义** | **语法示例** | **解释** |
| String | {string} | -- |
| Number | {number} | -- |
| Boolean | {boolean} | -- |
| Object | {Object} | -- |
| Function | {Function} | -- |
| RegExp | {RegExp} | -- |
| Array | {Array} | -- |
| Date | {Date} | -- |
| 单一类型集合 | {Array.<string>} | string 类型的数组 |
| 多类型 | {(number｜boolean)} | 可能是 number 类型, 也可能是 boolean 类型 |
| 允许为null | {?number} | 可能是 number, 也可能是 null |
| 不允许为null | {!Object} | Object 类型, 但不是 null |
| Function类型 | {function(number, boolean)} | 函数, 形参类型 |
| Function带返回值 | {function(number, boolean):string} | 函数, 形参, 返回值类型 |
| 参数可选 | @param {string=} name | 可选参数, =为类型后缀 |
| 可变参数 | @param {...number} args | 变长参数, ...为类型前缀 |
| 任意类型 | {\*} | 任意类型 |
| 可选任意类型 | @param {\*=} name | 可选参数，类型不限 |
| 可变任意类型 | @param {...\*} args | 变长参数，类型不限 |

**4.5 文件注释**

**【强制】 文件顶部必须包含文件注释，用 @file 标识文件说明。**

示例：

/\*\*

\* @file Describe the file

\*/

**【建议】 文件注释中可以用 @author 标识开发者信息。**

解释：

开发者信息能够体现开发人员对文件的贡献，并且能够让遇到问题或希望了解相关信息的人找到维护人。通常情况文件在被创建时标识的是创建者。随着项目的进展，越来越多的人加入，参与这个文件的开发，新的作者应该被加入 @author 标识。

@author 标识具有多人时，原则是按照 责任 进行排序。通常的说就是如果有问题，就是找第一个人应该比找第二个人有效。比如文件的创建者由于各种原因，模块移交给了其他人或其他团队，后来因为新增需求，其他人在新增代码时，添加 @author 标识应该把自己的名字添加在创建人的前面。

@author 中的名字不允许被删除。任何劳动成果都应该被尊重。

业务项目中，一个文件可能被多人频繁修改，并且每个人的维护时间都可能不会很长，不建议为文件增加 @author 标识。通过版本控制系统追踪变更，按业务逻辑单元确定模块的维护责任人，通过文档与wiki跟踪和查询，是更好的责任管理方式。

对于业务逻辑无关的技术型基础项目，特别是开源的公共项目，应使用 @author 标识。

示例：

/\*\*

\* @file Describe the file

\* @author author-name(mail-name@domain.com)

\* author-name2(mail-name2@domain.com)

\*/

**4.6 命名空间注释**

**【建议】 命名空间使用 @namespace 标识。**

示例：

/\*\*

\* @namespace

\*/

var util = {};

**4.7 类注释**

**【建议】 使用 @class 标记类或构造函数。**

解释：

对于使用对象 constructor 属性来定义的构造函数，可以使用 @constructor 来标记。

示例：

/\*\*

\* 描述

\*

\* @class

\*/

function Developer() {

// constructor body

}

**【建议】 使用 @extends 标记类的继承信息。**

示例：

/\*\*

\* 描述

\*

\* @class

\* @extends Developer

\*/

function Fronteer() {

Developer.call(this);

// constructor body

}

util.inherits(Fronteer, Developer);

**【强制】 使用包装方式扩展类成员时， 必须通过 @lends 进行重新指向。**

解释：

没有 @lends 标记将无法为该类生成包含扩展类成员的文档。

示例：

/\*\*

\* 类描述

\*

\* @class

\* @extends Developer

\*/

function Fronteer() {

Developer.call(this);

// constructor body

}

util.extend(

Fronteer.prototype,

/\*\* @lends Fronteer.prototype \*/{

\_getLevel: function () {

// TODO

}

}

);

**【强制】 类的属性或方法等成员信息使用 @public / @protected / @private 中的任意一个，指明可访问性。**

解释：

生成的文档中将有可访问性的标记，避免用户直接使用非 public 的属性或方法。

示例：

/\*\*

\* 类描述

\*

\* @class

\* @extends Developer

\*/

var Fronteer = function () {

Developer.call(this);

/\*\*

\* 属性描述

\*

\* @type {string}

\* @private

\*/

this.\_level = 'T12';

// constructor body

};

util.inherits(Fronteer, Developer);

/\*\*

\* 方法描述

\*

\* @private

\* @return {string} 返回值描述

\*/

Fronteer.prototype.\_getLevel = function () {

};

**4.8 函数/方法注释**

**【强制】 函数/方法注释必须包含函数说明，有参数和返回值时必须使用注释标识。**

**【强制】 参数和返回值注释必须包含类型信息和说明。**

**【建议】 当函数是内部函数，外部不可访问时，可以使用 @inner 标识。**

示例：

/\*\*

\* 函数描述

\*

\* @param {string} p1 参数1的说明

\* @param {string} p2 参数2的说明，比较长

\* 那就换行了.

\* @param {number=} p3 参数3的说明（可选）

\* @return {Object} 返回值描述

\*/

function foo(p1, p2, p3) {

var p3 = p3 || 10;

return {

p1: p1,

p2: p2,

p3: p3

};

}

**【强制】 对 Object 中各项的描述， 必须使用 @param 标识。**

示例：

/\*\*

\* 函数描述

\*

\* @param {Object} option 参数描述

\* @param {string} option.url option项描述

\* @param {string=} option.method option项描述，可选参数

\*/

function foo(option) {

// TODO

}

**【建议】 重写父类方法时， 应当添加 @override 标识。如果重写的形参个数、类型、顺序和返回值类型均未发生变化，可省略 @param、@return，仅用 @override 标识，否则仍应作完整注释。**

解释：

简而言之，当子类重写的方法能直接套用父类的方法注释时可省略对参数与返回值的注释。

**4.9 事件注释**

**【强制】 必须使用 @event 标识事件，事件参数的标识与方法描述的参数标识相同。**

示例：

/\*\*

\* 值变更时触发

\*

\* @event

\* @param {Object} e e描述

\* @param {string} e.before before描述

\* @param {string} e.after after描述

\*/

onchange: function (e) {

}

**【强制】 在会广播事件的函数前使用 @fires 标识广播的事件，在广播事件代码前使用 @event 标识事件。**

**【建议】 对于事件对象的注释，使用 @param 标识，生成文档时可读性更好。**

示例：

/\*\*

\* 点击处理

\*

\* @fires Select#change

\* @private

\*/

Select.prototype.clickHandler = function () {

/\*\*

\* 值变更时触发

\*

\* @event Select#change

\* @param {Object} e e描述

\* @param {string} e.before before描述

\* @param {string} e.after after描述

\*/

this.fire(

'change',

{

before: 'foo',

after: 'bar'

}

);

};

**4.10 常量注释**

**【强制】 常量必须使用 @const 标记，并包含说明和类型信息。**

示例：

/\*\*

\* 常量说明

\*

\* @const

\* @type {string}

\*/

var REQUEST\_URL = 'myurl.do';

**4.11 复杂类型注释**

**【建议】 对于类型未定义的复杂结构的注释，可以使用 @typedef 标识来定义。**

示例：

// `namespaceA~` 可以换成其它 namepaths 前缀，目的是为了生成文档中能显示 `@typedef` 定义的类型和链接。

/\*\*

\* 服务器

\*

\* @typedef {Object} namespaceA~Server

\* @property {string} host 主机

\* @property {number} port 端口

\*/

/\*\*

\* 服务器列表

\*

\* @type {Array.<namespaceA~Server>}

\*/

var servers = [

{

host: '1.2.3.4',

port: 8080

},

{

host: '1.2.3.5',

port: 8081

}

];

**4.12 细节注释**

对于内部实现、不容易理解的逻辑说明、摘要信息等，我们可能需要编写细节注释。

**【建议】 细节注释遵循单行注释的格式。说明必须换行时，每行是一个单行注释的起始。**

示例：

function foo(p1, p2) {

// 这里对具体内部逻辑进行说明

// 说明太长需要换行

for (...) {

....

}

}

**【强制】 有时我们会使用一些特殊标记进行说明。特殊标记必须使用单行注释的形式。下面列举了一些常用标记：**

解释：

1. TODO: 有功能待实现。此时需要对将要实现的功能进行简单说明。
2. FIXME: 该处代码运行没问题，但可能由于时间赶或者其他原因，需要修正。此时需要对如何修正进行简单说明。
3. HACK: 为修正某些问题而写的不太好或者使用了某些诡异手段的代码。此时需要对思路或诡异手段进行描述。
4. XXX: 该处存在陷阱。此时需要对陷阱进行描述。