# JavaScript编码规范

[1 前言](#_1 前言)

[2 JavaScript 语言规范](#_2 JavaScript 语言规范)

[2.1 变量](#_2.1 变量)

[2.1.1 必须 在声明变量时，加上 var 关键字](#_2.1.1 必须 在声明变量时，加上 var 关键字)

[2.1.2 建议 使用单一 var 模式](#_2.1.2 建议 使用单一 var 模式)

[2.2 分号](#_2.2 分号)

[2.2.1 必须 使用分号分隔语句](#_2.2.1 必须 使用分号分隔语句)

[2.3 函数](#_2.3 函数)

[2.3.1 可以 使用嵌套函数](#_2.3.1 可以使用嵌套函数)

[2.3.2 禁止 在块儿内声明函数](#_2.3.2 禁止在块儿内声明函数)

[2.3.3 禁止 使用arguments.callee](#_2.3.3 禁止使用arguments.callee)

[2.4 strict 模式](#_2.4 strict 模式)

[2.4.1 禁止 在块儿内声明函数](#_2.4.1 建议在函数作用域中，使用 use strict)

[2.5 异常和自定义异常](#_2.5 异常和自定义异常)

[2.5.1 可以 使用异常和自定义异常](#_2.5.1 可以使用异常和自定义异常)

[2.6 类型](#_2.6 类型)

[2.6.1 禁止 使用封装的基本类型](#_2.6.1 禁止使用封装的基本类型)

[2.7 Array 和 Object](#_2.7 Array 和 Object)

[2.7.1 必须 使用 Array 和 Object 直接量](#_2.7.1 必须 使用 Array 和 Object 直接量)

[2.7.2 禁止 使用关联数组](#_2.7.2 禁止 使用关联数组)

[2.8 原型](#_2.8 原型)

[2.8.1 不建议 使用多级原型结构](#_2.8.1 不建议 使用多级原型结构)

[2.8.2 禁止 修改内置对象的原型](#_2.8.2 禁止 修改内置对象的原型)

[2.9 with](#_2.9 with)

[2.9.1 禁止 使用 with() {}](#_2.9.1 禁止 使用 with() {})

[2.10 for-in 循环](#_2.10 for-in 循环)

[2.10.1 仅限于 object/map/hash 的遍历 禁止 使用 for-in 遍历 Array](#_2.10.1 仅限于 object/map/hash 的遍历 禁止 使用 for-in 遍历 Array)

[2.11 eval()](#_2.11 eval())

[2.11.1 仅限于 解析 JSON](#_2.11.1 仅限于 解析 JSON)

[2.12 == 和 ===](#_2.12 == 和 ===)

[2.12.1 禁止 使用隐式类型转换](#_2.12.1 禁止 使用隐式类型转换)

[3 JavaScript 代码风格](#_3 JavaScript 代码风格)

[3.1 命名](#_3.1 命名)

[3.1.1 推荐 使用 functionNamesLikeThis，variableNamesLikeThis， ClassNamesLikeThis，EnumNamesLikeThis， methodNamesLikeThis，SYMBOLIC\_CONSTANTS\_LIKE\_THIS](#_3.1.1 推荐 使用 functionNamesLikeThis，variableNamesLikeThis，ClassNamesLikeThis，EnumNamesLikeThis，methodNamesLikeThis，SYMBOLIC_CONSTANTS_LIKE_THIS)

[3.1.2 推荐 使用小写的文件名](#_3.1.2 推荐 使用小写的文件名)

[3.2 代码格式化](#_3.2 代码格式化)

[3.2.1 大括号](#_3.2.1 大括号)

[3.2.2 数组和对象的初始化](#_3.2.2 数组和对象的初始化)

[3.2.3 函数参数 和 函数调用](#_3.2.3 函数参数 和 函数调用)

[3.2.4 传递匿名函数](#_3.2.4 传递匿名函数)

[3.2.5 更多的缩进](#_3.2.5 更多的缩进)

[3.2.6 空行](#_3.2.6 空行)

[3.2.7 二元和三元操作符](#_3.2.7 二元和三元操作符)

[3.3 括号](#_3.3 括号)

[3.3.1 只在需要的时候使用](#_3.3.1 只在需要的时候使用)

[3.3.2 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 (。](#_3.3.2 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 (。)

[3.4 代码格式化](#_3.4 字符串)

[3.4.1 推荐使用 ’ 优于 "](#_3.4.1 推荐使用 ' 优于 \")

[3.5 函数定义](#_3.5 函数定义)

[3.5.1 推荐使用函数字面量定义函数](#_3.5.1 推荐使用函数字面量定义函数)

[3.6 注释](#_3.6 注释)

[3.6.1 顶层/文件注释](#_3.6.1 顶层/文件注释)

[3.6.2 类注释](#_3.6.2 类注释)

[3.6.3 方法与函数的注释](#_3.6.3 方法与函数的注释)

[3.6.4 属性注释](#_3.6.4 属性注释)

[3.6.5 类型转换的注释](#_3.6.5 类型转换的注释)

[3.6.6 JSDoc 缩进](#_3.6.6 JSDoc 缩进)

[3.6.6.1 枚举](#_3.6.6.1 枚举)

[3.6.6.2 Typedefs](#_3.6.6.2 Typedefs)

[3.6.7 JSDoc 中的 HTML](#_3.6.7 JSDoc 中的 HTML)

[3.7 缩进](#_3.7 缩进)

[3.7.1 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。](#_3.7.1 必须 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。)

[4 DOM](#_4 DOM)

[4.1 事件](#_4.1 事件)

[4.1.1 推荐使用事件代理](#_4.1.1 推荐 使用事件代理)

[4.1.2 必须事件绑定到DOM节点上，不允许使用css的class，如需选择一类 节点，自定义class，并添加 js- 的前缀](#_4.1.2 必须 事件绑定到DOM节点上，不允许使用css的class，如需选择一类节点，自定义class，并添加 js- 的前缀)

## 1 前言

JavaScript 是一门客户端脚本语言，在腾讯一直有着广泛的应用，特别是在浏览器端的行为管理。

这份指南列出了编写 JavaScript 时需要遵守的规则，目标是使 JavaScript 代码更加易于维护。

易维护的代码意味着代码具有如下特征：

* 可读性好
* 具有一致性
* 可预见性好
* 看起来好像同一个人编写的
* 有文档

以下文档中，每一条规范都会有一个实施级别，分别是：

* 必须： 必须执行
* 建议： 推荐不强制使用
* 不建议： 不推荐但也可以使用
* 可以： 可以使用
* 仅限于： 限制使用范围
* 禁止： 不可以使用

## 2 JavaScript 语言规范

### 2.1 变量

#### 2.1.1 **必须** 在声明变量时，加上 var 关键字

当你没有写 var，变量会暴露在全局上下文中，这样很可能会和现有变量冲突；另外，如果不加 var，很难明确该变量的作用域，变量很可能看起来像存在于在局部作用域中，但其实已经泄漏到 Document 或者 Window 中，所以务必用 var 去声明变量。

// bad

offset = 3;

// good

var offset = 3;

// bad

function sum(x ,y){

result = x + y;

return result;

}

// good

function sum(x ,y){

var result = x + y;

return result;

}

还有一种隐式创建全局变量的反模式，如下：

//bad

function foo(){

var a = b = 0; //相当于 var a = (b = 0); b 没有定义直接使用，会被创建为全局变量

}

//good

function(){

var a = 0, b = 0;

}

#### 2.1.2 **建议** 使用单一 var 模式

单一 var 可以在单一的位置看到所有定义的变量，还可以防止出现在变量定义以前就使用的问题。

//bad

function(){

var a;

...

var b;

...

var c;

...

}

//good

function(){

var a,

b,

c;

...

}

变量提升引起的一个问题：

//bad

var myname = "global";

function func(){

alert(myname); //undefined

var myname = "local";

alert(myname); // local

}

//good

var myname = "global";

function func(){

var myname;

alert(myname); //undefined

myname = "local"

alert(myname); // local

}

#### 2.1.3 **仅限于**在 commonJS 标准下，宿主回调或者Flash文件必须访问全局变量时使用全局变量

如果一个方法或变量，必须是全局变量才能会被宿主访问，或者供 Flash 对象访问，可以使用 window.global.xx 的方式定义全局变量，统一放到 global 命名空间下。

### 2.2 分号

#### 2.2.1 **必须** 使用分号分隔语句

如果仅依靠语句间的隐式分隔，有时会很麻烦。使用分号，你自己更能清楚哪里是语句的起止。另外，在代码压缩合并的过程中会去掉换行符，两行合并到一行也很可能会出现错误。

//bad

// 1.

MyClass.prototype.myMethod = function() {

return 42;

} // No semicolon here.

(function() { //A

// Some initialization code wrapped in a function to create a scope for locals.

})();

var x = {

'i': 1,

'j': 2

} // No semicolon here.

// 2. Trying to do one thing on Internet Explorer and another on Firefox.

// I know you'd never write code like this, but throw me a bone.

[normalVersion, ffVersion][isIE]();

var THINGS\_TO\_EAT = [apples, oysters, sprayOnCheese] // No semicolon here.

// 3. conditional execution a la bash

-1 == resultOfOperation() || die();

//good

// 1.

MyClass.prototype.myMethod = function() {

return 42;

}; // semicolon here.

(function() { //A

// Some initialization code wrapped in a function to create a scope for locals.

})();

var x = {

'i': 1,

'j': 2

}; // semicolon here.

// 2. Trying to do one thing on Internet Explorer and another on Firefox.

// I know you'd never write code like this, but throw me a bone.

[normalVersion, ffVersion][isIE]();

var THINGS\_TO\_EAT = [apples, oysters, sprayOnCheese]; //semicolon here.

// 3. conditional execution a la bash

-1 == resultOfOperation() || die();

以上代码的问题:

1. 报 JavaScript 错误 - 例1的语句会解释成, 函数function(){return 42;}被调用，匿名函数A作为其参数, 返回 42 后, 又一次被"调用", 这就导致了错误.
2. 例子2中, 你很可能会在运行时遇到 'no such property in undefined' 错误, 原因是代码试图这样 x[ffVersion][isIE](http://git.oa.com/omg-webfront/prepare/wikis/javascript-style-guide) 执行.
3. 当 resultOfOperation() 返回非 NaN 时, 就会调用die, 其结果也会赋给 THINGS\_TO\_EAT.

遗漏分号有时会出现很奇怪的结果, 所以确保语句以分号结束.

### 2.3 函数

#### 2.3.1 **可以**使用嵌套函数

使用嵌套函数，可以减少重复代码，隐藏函数，变量等。 可以随意使用。

#### 2.3.2 **禁止**在块儿内声明函数

虽然很多 JS 引擎都支持块内声明函数，但它不属于 ECMAScript 规范 (见 ECMA-262, 第13和14条)。各个浏览器糟糕的实现相互不兼容，有些也与未来 ECMAScript 草案相违背。ECMAScript 只允许在脚本的根语句或函数中声明函数。如果确实需要在块中定义函数，建议使用函数表达式来初始化变量:

//bad

if (x) {

function foo() {} //等同于在上下文最顶端声明

}

//good

if (x) {

var foo = function() {} //使用函数表达式初始化变量

}

#### 2.3.3 **禁止**使用arguments.callee

递归函数必须能够引用它本身。很典型的，函数通过自己的名字调用自己。然而，匿名函数 (通过 函数表达式 或者 函数构造器 创建) 没有名称。因此如果没有可访问的变量指向该函数，唯一能引用它的方式就是通过 arguments.callee。 例如，

function create() {

return function(n) {

if (n <= 1)

return 1;

return n \* arguments.callee(n - 1);

};

}

var result = create()(5); // returns 120 (5 \\* 4 \\* 3 \\* 2 \\* 1)

可以使用命名函数表达式解决此问题。

早期的JavaScript版本不允许命名函数表达式，也因为这个原因你不能构造一个递归的函数表达式。

例如，下边这个语法就是行的通的：

function factorial (n) {

return !(n > 1) ? 1 : factorial(n - 1) \* n;

}

[1,2,3,4,5].map(factorial);

但，这样就不行：

[1,2,3,4,5].map(function (n) {

return !(n > 1) ? 1 : /\* what goes here? \*/ (n - 1) \* n;

});

ECMAScript 3 通过命名函数表达式解决这些问题。如：

[1,2,3,4,5].map(function factorial (n) {

return !(n > 1) ? 1 : factorial(n-1)\*n;

});

### 2.4 strict 模式

#### 2.4.1 **建议**在函数作用域中，使用 use strict

function fo(){

"use strict";

...

}

strict 模式会被 use strict 触发，触发后，在新的支持此模式的浏览器中，函数中的代码在 ECMAScript 语言 strict 子集中运行。对于之前的浏览器，不会有任何影响。

strict 模式使程序更加简洁不易出错。例如，在 strict 模式中 with 会报错。

关于 strict 模式，详细参考：[MDN-Strict mode](http://git.oa.com/omg-webfront/prepare/wikis/javascript-style-guide" \l "3-2-4)

### 2.5 异常和自定义异常

#### 2.5.1 **可以**使用异常和自定义异常

正确的使用异常，可以增加代码的健壮性。

### 2.6 类型

#### 2.6.1 **禁止**使用封装的基本类型

没有任何理由去封装基本类型, 另外还存在一些风险。

//bad

var x = new Boolean(false);

if (x) {

alert('hi'); // Shows 'hi'.

}

var y = new String("Hi");

//good

var x = false;

if (x) {

alert('hi'); // Shows 'hi'.

}

var y = "Hi";

除非明确用于类型转换, 其他情况请千万不要这样做！

var x = Boolean(0);

if (x) {

alert('hi'); // This will never be alerted.

}

typeof Boolean(0) == 'boolean';

typeof new Boolean(0) == 'object';

### 2.7 Array 和 Object

### 2.7.1 **必须** 使用 Array 和 Object 直接量

使用 Array 和 Object 语法, 而不使用 Array 和 Object 构造器.

使用 Array 构造器很容易因为传参不恰当导致错误.

// Length is 3.

var a1 = new Array(x1, x2, x3);

// Length is 2.

var a2 = new Array(x1, x2);

// If x1 is a number and it is a natural number the length will be x1.

// If x1 is a number but not a natural number this will throw an exception.

// Otherwise the array will have one element with x1 as its value.

var a3 = new Array(x1);

// Length is 0.

var a4 = new Array();

如果传入一个参数而不是2个参数, 数组的长度很有可能就不是你期望的数值了.

为了避免这些歧义, 我们应该使用更易读的直接量来声明.

var a = \[x1, x2, x3];

var a2 = \[x1, x2];

var a3 = \[x1];

var a4 = \[];

虽然 Object 构造器没有上述类似的问题, 但鉴于可读性和一致性考虑, 最好还是在字面上更清晰地指明.

var o = new Object();

var o2 = new Object();

o2.a = 0;

o2.b = 1;

o2.c = 2;

o2\['strange key'] = 3;

应该写成:

var o = {};

var o2 = {

'a': 0,

'b': 1,

'c': 2,

'strange key': 3

};

### 2.7.2 **禁止** 使用关联数组

永远不要使用 Array 作为 map/hash/associative 数组。

数组中不允许使用非整型作为索引值, 所以也就不允许用关联数组。请使用 Object 来表示 map/hash 对象。 Array 仅仅是扩展自 Object (类似于其他 JS 中的对象, 就像 Date, RegExp 和 String)一样来使用。

//bad

var a=\[1,2,3,4];

a\['foo']='Hello World';

alert(a.length);

alert(a.foo);

//good

var a = {

0:1,

1:2,

2:3,

3:4,

'foo':'Hello World'

};

### 2.8 原型

### 2.8.1 **不建议** 使用多级原型结构

当你自定义一个 D 类, 且把 B 类作为其原型, 那么这就获得了一个多级原型结构，类似高级语言中的继承。这些原型结构会变得越来越复杂!

function D() {

goog.base(this)

}

goog.inherits(D, B);

D.prototype.method = function() {

...

};

### 2.8.2 **禁止** 修改内置对象的原型

千万不要修改内置对象, 如 Object.prototype 和 Array.prototype 的原型. 而修改内置对象, 如 Function.prototype 的原型, 虽然危险不大, 但仍会导致调试时的诡异现象.修改内置对象的原型，也可能造成 for in 循环将扩展的属性或方法遍历出来。所以，请避免修改内置对象原型.

仅在一种情况下可以修改内置对象原型：W3C 标准中有的原型方法或属性，现有的浏览器还未实现，这时，可以扩展该方法或属性，但要注意写法，如下：

if(Array.prototype.forEach === undefined){

Array.prototype.forEach = function(){

...

}

}

### 2.9 with

### 2.9.1 **禁止** 使用 with() {}

使用 with 让你的代码在语义上变得不清晰。 因为 with 的对象, 可能会与局部变量产生冲突, 从而改变你程序原本的用义；还会在运行时创建一层运行时上下文，影响性能。例如下面的代码：

with (foo) {

var x = 3;

return x;

}

局部变量 x 可能被 foo 的属性覆盖, 当它定义一个 setter 时, 在赋值 3 后会执行很多其他代码。 所以不要使用 with 语句。

### 2.10 for-in 循环

### 2.10.1 **仅限于** object/map/hash 的遍历 **禁止** 使用 for-in 遍历 Array

对 Array 用 for-in 循环有时会出错. 因为它并不是从 0 到 length - 1 进行遍历, 而是所有出现在对象及其原型链的键值. 下面就是一些失败的使用案例:

function printArray(arr) {

for (var key in arr) {

print(arr[key]);

}

}

printArray(\[0,1,2,3]); // This works.

var a = new Array(10);

printArray(a); // This is wrong.

a = document.getElementsByTagName('\\*');

printArray(a); // This is wrong.

a = \[0,1,2,3];

a.buhu = 'wine';

printArray(a); // This is wrong again.

a = new Array;

a[3] = 3;

printArray(a); // This is wrong again.

而遍历数组通常用最普通的 for 循环.

function printArray(arr) {

var l = arr.length;

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

print(arr[i]);

}

}

### 2.11 eval()

### 2.11.1 **仅限于** 解析 JSON

eval() 会让程序执行的比较混乱，当 eval() 执行的内容包含用户输入的话还可能有代码注入的安全问题。可以用其他更佳的，更清晰，更安全的方式写你的代码，所以一般情况下请不要使用 eval()。当碰到一些需要解析序列化串的情况下(如, 计算 RPC 响应)，使用 eval 很容易实现。

解析序列化串是指将字节流转换成内存中的数据结构. 比如, 你可能会将一个对象输出成文件形式:

users = \[

{

name: 'Eric',

id: 37824,

email: 'jellyvore@myway.com'

},

{

name: 'xtof',

id: 31337,

email: 'b4d455h4x0r@google.com'

},

...

];

使用 ajax 方式获取后，取得的是字符串，想要将字符串变成 JavaScript 中的对象，使用 eval 会非常方便。

但是，更推荐的解析 JSON 数据的方式是用 JSON.parse() 或使用 json 的解析库解析。

### 2.12 == 和 ===

### 2.12.1 **禁止** 使用隐式类型转换

使用 == != 比较会执行隐式类型转换，

false == 0; //true

"" == 0; //true

为了避免隐式类型转换导致的混淆不清，建议使用 === !== 对值和类型进行比较，

false == 0; //false

"" == 0; //false

0 === 0； //true

这样在做比较时，我们可以清晰的知道参数的值和类型，减少阅读代码是的脑力开销。

## 3 JavaScript 代码风格

### 3.1 命名

#### 3.1.1 **推荐** 使用 functionNamesLikeThis，variableNamesLikeThis，ClassNamesLikeThis，EnumNamesLikeThis，methodNamesLikeThis，SYMBOLIC\_CONSTANTS\_LIKE\_THIS

function functionNamesLikeThis(){//函数定义，驼峰式，首字母小写

}

var variableNamesLikeThis;//变量定义，驼峰式，首字母小写

function ClassNamesLikeThis(){//类名，驼峰式，首字母大写

}

var classInstance = new ClassNamesLikeThis();

EnumNamesLikeThis, //枚举变量，驼峰式，首字母大写

window.methodNamesLikeThis = function(){ //方法名，驼峰式，首字母小写

}

var SYMBOLIC\_CONSTANTS\_LIKE\_THIS = 3.14159265358979323846; //常量名，单词用下划线分割，全部字母大写

另外，

* 变量名中的单词必须为英文单词，不许出现混拼音，尤其中英文混合；
* 功能类似的两个函数，不许使用 fn1 fn2 这种名字加数字的方式命名；

#### 3.1.2 **推荐** 使用小写的文件名

文件名应该使用小写字符, 以避免在有些系统平台上不识别大小写的命名方式. 文件名以.js结尾, 不要包含除 - 和 \_ 外的标点符号(使用 - 优于 \_).

### 3.2 代码格式化

#### 3.2.1 大括号

分号会被隐式插入到代码中，所以你务必在插入大括号，哪怕只有一行。例如:

//bad

if (something)

// ...

else

// ...

//good

if (something) {

// ...

} else {

// ...

}

#### 3.2.2 数组和对象的初始化

如果初始值不是很长(少于 80 个字符), 就保持写在单行上: var arr = [1, 2, 3]; // No space after [ or before ]. var obj = {a: 1, b: 2, c: 3}; // No space after { or before }

初始值占用多行时, 缩进4个空格. // Object initializer. var inset = { top: 10, right: 20, bottom: 15, left: 12 };

// Array initializer.

this.rows\_ = [

'"Slartibartfast" <fjordmaster@magrathea.com>',

'"Zaphod Beeblebrox" <theprez@universe.gov>',

'"Ford Prefect" <ford@theguide.com>',

'"Arthur Dent" <has.no.tea@gmail.com>',

'"Marvin the Paranoid Android" <marv@googlemail.com>',

'the.mice@magrathea.com'

];

// Used in a method call.

goog.dom.createDom(goog.dom.TagName.DIV, {

id: 'foo',

className: 'some-css-class',

style: 'display:none'

}, 'Hello, world!');

比较长的标识符或者数值, 不要为了让代码好看些而手工对齐. 如:

//good

CORRECT\_Object.prototype = {

a: 0,

b: 1,

lengthyName: 2

};

//bad

WRONG\_Object.prototype = {

a : 0,

b : 1,

lengthyName: 2

};

#### 3.2.3 函数参数 和 函数调用

尽量让函数参数在同一行上. 如果一行超过 80 字符, 每个参数独占一行, 并以4个空格缩进, 或者与括号对齐, 以提高可读性. 尽可能不要让每行超过80个字符。比如下面这样:

// Four-space, wrap at 80. Works with very long function names, survives

// renaming without reindenting, low on space.

goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(

veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {

// ...

};

// Four-space, one argument per line. Works with long function names,

// survives renaming, and emphasizes each argument.

goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(

veryDescriptiveArgumentNumberOne,

veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy,

artichokeDescriptorAdapterIterator) {

// ...

};

// Parenthesis-aligned indentation, wrap at 80. Visually groups arguments,

// low on space.

function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {

// ...

}

// Parenthesis-aligned, one argument per line. Visually groups and

// emphasizes each individual argument.

function bar(veryDescriptiveArgumentNumberOne,

veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy,

artichokeDescriptorAdapterIterator) {

// ...

}

函数调用时同理，

// call function goog.foo.bar.doThingThatIsVeryDifficultToExplain

goog.foo.bar.doThingThatIsVeryDifficultToExplain(

veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {

// ...

};

#### 3.2.4 传递匿名函数

如果参数中有匿名函数, 函数体从调用该函数的左边开始缩进2个空格, 而不是从 function 这个关键字开始. 这让匿名函数更加易读 (不要增加很多没必要的缩进让函数体显示在屏幕的右侧).

//bad

var names = items.map(function(item) {

return item.name;

});

//good

prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {

if (a1.equals(a2)) {

someOtherLongFunctionName(a1);

} else {

andNowForSomethingCompletelyDifferent(a2.parrot);

}

});

#### 3.2.5 更多的缩进

事实上, 除了 初始化数组和对象 , 和传递匿名函数外, 所有被拆开的多行文本要么选择与之前的表达式左对齐, 要么以4个(而不是2个)空格作为一缩进层次.

//bad

someWonderfulHtml = '' +

getEvenMoreHtml(someReallyInterestingValues, moreValues,

evenMoreParams, 'a duck', true, 72,

slightlyMoreMonkeys(0xfff)) +

'';

//good

thisIsAVeryLongVariableName =

hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();

thisIsAVeryLongVariableName = 'expressionPartOne' + someMethodThatIsLong() +

thisIsAnEvenLongerOtherFunctionNameThatCannotBeIndentedMore();

someValue = this.foo(

shortArg,

'Some really long string arg - this is a pretty common case, actually.',

shorty2,

this.bar());

if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&

!ambientNotification.isActive() && (client.isAmbientSupported() ||

client.alwaysTryAmbientAnyways()) {

ambientNotification.activate();

}

#### 3.2.6 空行

使用空行来划分一组逻辑上相关联的代码片段.

doSomethingTo(x);

doSomethingElseTo(x);

andThen(x);

nowDoSomethingWith(y);

andNowWith(z);

#### 3.2.7 二元和三元操作符

操作符始终跟随着前行, 这样就不用顾虑分号的隐式插入问题. 如果一行实在放不下, 还是按照上述的缩进风格来换行.

var x = a ? b : c; // All on one line if it will fit.

// Indentation +4 is OK.

var y = a ?

longButSimpleOperandB : longButSimpleOperandC;

// Indenting to the line position of the first operand is also OK.

var z = a ?

moreComplicatedB :

moreComplicatedC;

### 3.3 括号

#### 3.3.1 只在需要的时候使用

不要滥用括号，只在必要的时候使用它。

对于一元操作符(如delete，typeof 和 void )， 或是在某些关键词(如 return，throw，case，new )之后，不要使用括号。

#### 3.3.2 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 () {

});

### 3.4 字符串

#### 3.4.1 **推荐**使用 ' 优于 "

单引号 (') 优于双引号 (")。

var msg = 'This is some HTML';

当你创建一个包含 HTML 代码的字符串时就知道它的好处了。 var msgCode = ' This is msg code ';

### 3.5 函数定义

#### 3.5.1 **推荐**使用函数字面量定义函数

使用 function fnName(){} 的方式定义函数会自动提升到作用域顶端。

render();

function render(){

// your code here.

}

render();

推荐使用函数字面量的方式，定义之后函数才可用，能更好的控制时序。 render(); // undefined is not a function var render = function(){ // your code here. } render();

### 3.6 注释

我们使用 [JSDoc](http://usejsdoc.org/) 中的注释风格. 行内注释使用 // 变量 的形式. 另外, 我们也遵循 C++ 代码注释风格 . 这也就是说你需要:

* 版权和著作权的信息,
* 文件注释中应该写明该文件的基本信息(如, 这段代码的功能摘要, 如何使用, 与哪些东西相关), 来告诉那些不熟悉代码的读者.
* 类, 函数, 变量和必要的注释,
* 期望在哪些浏览器中执行,
* 正确的大小写, 标点和拼写. 为了避免出现句子片段, 请以合适的大/小写单词开头, 并以合适的标点符号结束这个句子. 现在假设维护这段代码的是一位初学者. 这可能正好是这样的! 目前很多编译器可从 JSDoc 中提取类型信息, 来对代码进行验证, 删除和压缩. 因此, 你很有必要去熟悉正确完整的 JSDoc .

#### 3.6.1 顶层/文件注释

顶层注释用于告诉不熟悉这段代码的读者这个文件中包含哪些东西. 应该提供文件的大体内容, 它的作者, 依赖关系和兼容性信息. 如下:

// Copyright 2009 Google Inc. All Rights Reserved.

/\*\*

\* @fileoverview Description of file, its uses and information

\* about its dependencies.

\* @author user@google.com (Firstname Lastname)

\*/

#### 3.6.2 类注释

每个类的定义都要附带一份注释, 描述类的功能和用法. 也需要说明构造器参数. 如果该类继承自其它类, 应该使用 @extends 标记. 如果该类是对接口的实现, 应该使用 @implements 标记.

/\*\*

\* Class making something fun and easy.

\* @param {string} arg1 An argument that makes this more interesting.

\* @param {Array.<number>} arg2 List of numbers to be processed.

\* @constructor

\* @extends {goog.Disposable}

\*/

project.MyClass = function(arg1, arg2) {

// ...

};

goog.inherits(project.MyClass, goog.Disposable);

#### 3.6.3 方法与函数的注释

提供参数的说明. 使用完整的句子, 并用第三人称来书写方法说明.

/\*\*

\* Converts text to some completely different text.

\* @param {string} arg1 An argument that makes this more interesting.

\* @return {string} Some return value.

\*/

project.MyClass.prototype.someMethod = function(arg1) {

// ...

};

/\*\*

\* Operates on an instance of MyClass and returns something.

\* @param {project.MyClass} obj Instance of MyClass which leads to a long

\* comment that needs to be wrapped to two lines.

\* @return {boolean} Whether something occured.

\*/

function PR\_someMethod(obj) {

// ...

}

对于一些简单的, 不带参数的 getters, 说明可以忽略.

/\*\*

\* @return {Element} The element for the component.

\*/

goog.ui.Component.prototype.getElement = function() {

return this.element\_;

};

#### 3.6.4 属性注释

也需要对属性进行注释.

/\*\*

\* Maximum number of things per pane.

\* @type {number}

\*/

project.MyClass.prototype.someProperty = 4;

#### 3.6.5 类型转换的注释

有时, 类型检查不能很准确地推断出表达式的类型, 所以应该给它添加类型标记注释来明确之, 并且必须在表达式和类型标签外面包裹括号.

/\*\* @type {number} \*/ (x)

(/\*\* @type {number} \*/ x)

#### 3.6.6 JSDoc 缩进

如果你在 @param, @return, @supported, @this 或 @deprecated 中断行, 需要像在代码中一样, 使用4个空格作为一个缩进层次.

/\*\*

\* Illustrates line wrapping for long param/return descriptions.

\* @param {string} foo This is a param with a description too long to fit in

\* one line.

\* @return {number} This returns something that has a description too long to

\* fit in one line.

\*/

project.MyClass.prototype.method = function(foo) {

return 5;

};

不要在 @fileoverview 标记中进行缩进. 虽然不建议, 但也可对说明文字进行适当的排版对齐. 不过, 这样带来一些负面影响, 就是当你每次修改变量名时, 都得重新排版说明文字以保持和变量名对齐.

/\*\*

\* This is NOT the preferred indentation method.

\* @param {string} foo This is a param with a description too long to fit in

\* one line.

\* @return {number} This returns something that has a description too long to

\* fit in one line.

\*/

project.MyClass.prototype.method = function(foo) {

return 5;

};

##### 3.6.6.1 枚举

/\*\*

\* Enum for tri-state values.

\* @enum {number}

\*/

project.TriState = {

TRUE: 1,

FALSE: -1,

MAYBE: 0

};

注意一下, 枚举也具有有效类型, 所以可以当成参数类型来用.

/\*\*

\* Sets project state.

\* @param {project.TriState} state New project state.

\*/

project.setState = function(state) {

// ...

};

##### 3.6.6.2 Typedefs

有时类型会很复杂. 比如下面的函数, 接收 Element 参数:

/\*\*

\* @param {string} tagName

\* @param {(string|Element|Text|Array.<Element>|Array.<Text>)} contents

\* @return {Element}

\*/

goog.createElement = function(tagName, contents) {

...

};

你可以使用 @typedef 标记来定义个常用的类型表达式.

/\*\* @typedef {(string|Element|Text|Array.<Element>|Array.<Text>)} \*/

goog.ElementContent;

/\*\*

\* @param {string} tagName

\* @param {goog.ElementContent} contents

\* @return {Element}

\*/

goog.createElement = function(tagName, contents) {

...

};

#### 3.6.7 JSDoc 中的 HTML

似于 JavaDoc, JSDoc 支持许多 HTML 标签, 如 <code>, <pre>, <tt>, <strong>, <ul>, <ol>, <li>, <a>, 等等.

这就是说 JSDoc 不会完全依照纯文本中书写的格式. 所以, 不要在 JSDoc 中, 使用空白字符来做格式化:

/\*\*

\* Computes weight based on three factors:

\* items sent

\* items received

\* last timestamp

\*/

上面的注释, 出来的结果是:

Computes weight based on three factors: items sent items received items received

应该这样写:

/\*\*

\* Computes weight based on three factors:

\* <ul>

\* <li>items sent

\* <li>items received

\* <li>last timestamp

\* </ul>

\*/

另外, 也不要包含任何 HTML 或类 HTML 标签, 除非你就想让它们解析成 HTML 标签.

/\*\*

\* Changes <b> tags to <span> tags.

\*/

出来的结果是:

Changes tags to tags.

另外, 也应该在源代码文件中让其他人更可读, 所以不要过于使用 HTML 标签:

/\*\*

\* Changes &lt;b&gt; tags to &lt;span&gt; tags.

\*/

上面的代码中, 其他人就很难知道你想干嘛, 直接改成下面的样子就清楚多了:

/\*\*

\* Changes 'b' tags to 'span' tags.

\*/

### 3.7 缩进

#### 3.7.1 **必须** 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。

仅限于在引入的 js 库中允许使用 2 个空格的缩进或 tab 字符。源代码中必须使用 4 个空格缩进。

## 4 DOM

### 4.1 事件

#### 4.1.1 **推荐** 使用事件代理

事件代理提高性能，降低内存溢出风险，DOM更新后无需重新绑定事件处理器。

#### 4.1.2 **必须** 事件绑定到DOM节点上，不允许使用css的class，如需选择一类节点，自定义class，并添加 js- 的前缀

此条标准旨在减少js和css之间的依赖，防止css中class名称改变造成功能失效的情况。