JavaScript编码规范

1 前言

本文档的目标是使JavaScript代码风格保持一致，容易被理解和被维护。

2 代码风格

2.1 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。

2.2 结构

2.2.1 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...

}
```

2.2.2 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。

2.2.3 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：

```
// 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;
}
```

2.2.4 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();
```

2.3 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);
```

2.4 注释

2.4.1 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。

2.4.2 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。

2.4.3 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。

2.4.4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);
```

2.4.5 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。

2.4.6 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'
        }
    );
};
```

3 语言特性

3.1 变量

[强制] 变量在使用前必须通过 `var` 定义。

解释：

不通过 var 定义变量将导致变量污染全局环境。

示例：

```
// good
var name = 'MyName';

// bad
name = 'MyName';
```

[强制] 每个 `var` 只能声明一个变量。

[强制] 变量必须 `即用即声明`，不得在函数或其它形式的代码块起始位置统一声明所有变量。

解释：

变量声明与使用的距离越远，出现的跨度越大，代码的阅读与维护成本越高。虽然JavaScript的变量是函数作用域，还是应该根据编程中的意图，缩小变量出现的距离空间。

示例：

```
// good
function kv2List(source) {
    var list = [];

    for (var key in source) {
        if (source.hasOwnProperty(key)) {
            var item = {
                k: key,
                v: source[key]
            };
            list.push(item);
        }
    }

    return list;
}

// bad
function kv2List(source) {
    var list = [];
    var key;
    var item;

    for (key in source) {
        if (source.hasOwnProperty(key)) {
            item = {
                k: key,
                v: source[key]
            };
            list.push(item);
        }
    }

    return list;
}
```

3.2 条件

[强制] 在 **Equality Expression** 中使用类型严格的 `===`。仅当判断 **null** 或 **undefined** 时，允许使用 `== null`。

解释：

使用 === 可以避免等于判断中隐式的类型转换。

示例：

```
// good
if (age === 30) {
    // ......
}

// bad
if (age == 30) {
    // ......
}
```

[建议] 对于相同变量或表达式的多值条件，用 `switch` 代替 `if`。

示例：

```
// good
switch (typeof variable) {
    case 'object':
        // ......
        break;
    case 'number':
    case 'boolean':
    case 'string':
        // ......
        break;
}

// bad
var type = typeof variable;
if (type === 'object') {
    // ......
}
else if (type === 'number' || type === 'boolean' || type === 'string') {
    // ......
}
```

3.3 对象

[强制] 使用对象字面量 `{}` 创建新 `Object`。

示例：

```
// good
var obj = {};

// bad
var obj = new Object();
```

[强制] 对象创建时，如果一个对象的所有 `属性` 均可以不添加引号，则所有 `属性` 不得添加引号。

示例：

```
var info = {
    name: 'someone',
    age: 28
};
```

3.4 数组

[强制] 使用数组字面量 `[]` 创建新数组，除非想要创建的是指定长度的数组。

示例：

```
// good
var arr = [];

// bad
var arr = new Array();
```

3.5 函数

3.5.1 函数长度

[建议] 一个函数的长度控制在 `50` 行以内。

3.5.2 参数设计

[建议] 一个函数的参数控制在 `6` 个以内。

解释：

除去不定长参数以外，函数具备不同逻辑意义的参数建议控制在 6 个以内，过多参数会导致维护难度增大。

某些情况下，如使用 AMD Loader 的 require 加载多个模块时，其 callback 可能会存在较多参数，因此对函数参数的个数不做强制限制。

4. DOM

4.1 元素获取

[建议] 对于单个元素，尽可能使用 `document.getElementById` 获取，避免使用 `document.all`。

[建议] 对于多个元素的集合，尽可能使用 `context.getElementsByTagName` 获取。其中 `context` 可以为 `document` 或其他元素。指定 `tagName` 参数为 `*` 可以获得所有子元素。

[建议] 遍历元素集合时，尽量缓存集合长度。如需多次操作同一集合，则应将集合转为数组。

解释：

原生获取元素集合的结果并不直接引用 DOM 元素，而是对索引进行读取，所以 DOM 结构的改变会实时反映到结果中。

[建议] 获取元素的直接子元素时使用 `children`。避免使用 `childNodes`，除非预期是需要包含文本、注释和属性类型的节点。

4.2 样式获取

[建议] 获取元素实际样式信息时，应使用 `getComputedStyle` 或 `currentStyle`。

解释：

通过 style 只能获得内联定义或通过 JavaScript 直接设置的样式。通过 CSS class 设置的元素样式无法直接通过 style 获取。

4.3 样式设置

[建议] 尽可能通过为元素添加预定义的 **className** 来改变元素样式，避免直接操作 **style** 设置。

[强制] 通过 **style** 对象设置元素样式时，对于带单位非 **0** 值的属性，不允许省略单位。

解释：

除了 IE，标准浏览器会忽略不规范的属性值，导致兼容性问题。

4.4 DOM 事件

[建议] 优先使用 `addEventListener / attachEvent` 绑定事件，避免直接在 **HTML** 属性中或 **DOM** 的 `expando` 属性绑定事件处理。

解释：

expando 属性绑定事件容易导致互相覆盖。

[建议] 使用 `addEventListener` 时第三个参数使用 `false`。

解释：

标准浏览器中的 addEventListener 可以通过第三个参数指定两种时间触发模型：冒泡和捕获。而 IE 的 attachEvent 仅支持冒泡的事件触发。所以为了保持一致性，通常 addEventListener 的第三个参数都为 false。

[建议] 在没有事件自动管理的框架支持下，应持有监听器函数的引用，在适当时候（元素释放、页面卸载等）移除添加的监听器。