JavaScript编码规范

1 前言

JavaScript在百度一直有着广泛的应用，特别是在浏览器端的行为管理。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。

虽然本文档是针对JavaScript设计的，但是在使用各种JavaScript的预编译语言时(如TypeScript等)时，适用的部分也应尽量遵循本文档的约定。

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：

```
// 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;
}
```

2.2.4 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 () {
});
```

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。**

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
9. AMD 模块

[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。

2.4.4 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

类型定义	语法示例	解释
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

2.4.5 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)
 */
```

2.4.6 命名空间注释

[建议] 命名空间使用 `@namespace` 标识。

示例：

```
/**
 * @namespace
 */
var util = {};
```

2.4.7 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 () {
};
```

2.4.8 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。

2.4.9 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'
        }
    );
};
```

2.4.10 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';
```

2.4.11 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
    }
];
```

2.4.12 AMD 模块注释

[强制] AMD 模块使用 `@module` 或 `@exports` 标识。

解释:

@exports 与 @module 都可以用来标识模块，区别在于 @module 可以省略模块名称。而只使用 @exports 时在 namepaths 中可以省略 module: 前缀。

示例:

```
define(
    function (require) {

        /**
         * foo description
         *
         * @exports Foo
         */
        var foo = {
            // TODO
        };

        /**
         * baz description
         *
         * @return {boolean} return description
         */
        foo.baz = function () {
            // TODO
        };

        return foo;
```

```
    }
);
```

也可以在 exports 变量前使用 @module 标识：

```
define(
    function (require) {

        /**
         * module description.
         *
         * @module foo
         */
        var exports = {};


        /**
         * bar description
         *
         */
        exports.bar = function () {
            // TODO
        };

        return exports;
    }
);
```

如果直接使用 factory 的 exports 参数，还可以：

```
/**
 * module description.
 *
 * @module
 */
define(
    function (require, exports) {

        /**
         * bar description
         *
         */
        exports.bar = function () {
            // TODO
        };
        return exports;
    }
);
```

[强制] 对于已使用 `@module` 标识为 AMD模块 的引用，在 `namepaths` 中必须增加 `module:` 作前缀。

解释：

namepaths 没有 module: 前缀时，生成的文档中将无法正确生成链接。

示例：

```
/**
 * 点击处理
 *
 * @fires module:Select#change
 * @private
 */
Select.prototype.clickHandler = function () {
    /**
     * 值变更时触发
     *
     * @event module: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'
        }
    );
};
```

[建议] 对于类定义的模块，可以使用 `@alias` 标识构建函数。

示例：

```
/**
 * A module representing a jacket.
 * @module jacket
 */
define(
    function () {

        /**
         * @class
         * @alias module:jacket
         */
        var Jacket = function () {
        };
```

```
        return Jacket;
    }
);
```

[建议] 多模块定义时，可以使用 `@exports` 标识各个模块。

示例：

```
// one module
define('html/utils',
    /**
     * Utility functions to ease working with DOM elements.
     * @exports html/utils
     */
    function () {
        var exports = {
        };

        return exports;
    }
);

// another module
define('tag',
    /** @exports tag */
    function () {
        var exports = {
        };

        return exports;
    }
);
```

[建议] 对于 exports 为 Object 的模块，可以使用`@namespace`标识。

解释：

使用 @namespace 而不是 @module 或 @exports 时，对模块的引用可以省略 module: 前缀。

[建议] 对于 exports 为类名的模块，使用 `@class` 和 `@exports` 标识。

示例：

```
// 只使用 @class Bar 时，类方法和属性都必须增加 @name Bar#methodName 来标识，与
@exports 配合可以免除这一麻烦，并且在引用时可以省去 module: 前缀。
// 另外需要注意类名需要使用 var 定义的方式。
```

```
/**
 * Bar description
 *
 * @see foo
 * @exports  Bar
 * @class
 */
var Bar = function () {
    // TODO
};

/**
 * baz description
 *
 * @return {(string|Array)} return description
 */
Bar.prototype.baz = function () {
    // TODO
};
```

2.4.13 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, opt_p3) {
    // 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。

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。

解释：

一个 var 声明多个变量，容易导致较长的行长度，并且在修改时容易造成逗号和分号的混淆。

示例：

```
// good
var hangModules = [];
var missModules = [];
var visited = {};

// bad
var hangModules = [],
    missModules = [],
    visited = {};
```

[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] 尽可能使用简洁的表达式。

示例：

```
// 字符串为空

// good
if (!name) {
    // ......
}

// bad
if (name === '') {
    // ......
}
```

```
// 字符串非空

// good
if (name) {
    // ......
}

// bad
if (name !== '') {
    // ......
}
```

```
// 数组非空

// good
if (collection.length) {
    // ......
}

// bad
if (collection.length > 0) {
    // ......
}
```

```
// 布尔不成立

// good
if (!notTrue) {
```

```
    // ......
}

// bad
if (notTrue === false) {
    // ......
}
```

```
// null 或 undefined

// good
if (noValue == null) {
  // ......
}

// bad
if (noValue === null || typeof noValue === 'undefined') {
  // ......
}
```

[建议] 按执行频率排列分支的顺序。

解释：

按执行频率排列分支的顺序好处是：

1. 阅读的人容易找到最常见的情况，增加可读性。
2. 提高执行效率。

[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') {
    // ......
}
```

[建议] 如果函数或全局中的 `else` 块后没有任何语句，可以删除 `else`。

示例：

```
// good
function getName() {
    if (name) {
        return name;
    }

    return 'unnamed';
}

// bad
function getName() {
    if (name) {
        return name;
    }
    else {
        return 'unnamed';
    }
}
```

3.3 循环

[建议] 不要在循环体中包含函数表达式，事先将函数提取到循环体外。

解释：

循环体中的函数表达式，运行过程中会生成循环次数个函数对象。

示例：

```
// good
function clicker() {
    // ......
}

for (var i = 0, len = elements.length; i < len; i++) {
    var element = elements[i];
    addListener(element, 'click', clicker);
}
```

```
// bad
for (var i = 0, len = elements.length; i < len; i++) {
    var element = elements[i];
    addListener(element, 'click', function () {});
}
```

[建议] 对循环内多次使用的不变值，在循环外用变量缓存。

示例：

```
// good
var width = wrap.offsetWidth + 'px';
for (var i = 0, len = elements.length; i < len; i++) {
    var element = elements[i];
    element.style.width = width;
    // ......
}


// bad
for (var i = 0, len = elements.length; i < len; i++) {
    var element = elements[i];
    element.style.width = wrap.offsetWidth + 'px';
    // ......
}
```

[建议] 对有序集合进行遍历时，缓存 `length`。

解释：

虽然现代浏览器都对数组长度进行了缓存，但对于一些宿主对象和老旧浏览器的数组对象，在每次 length 访问时会动态计算元素个数，此时缓存 length 能有效提高程序性能。

示例：

```
for (var i = 0, len = elements.length; i < len; i++) {
    var element = elements[i];
    // ......
}
```

[建议] 对有序集合进行顺序无关的遍历时，使用逆序遍历。

解释：

逆序遍历可以节省变量，代码比较优化。

示例：

```
var len = elements.length;
while (len--) {
    var element = elements[len];
    // ......
}
```

3.4 类型

3.4.1 类型检测

[建议] 类型检测优先使用 `typeof`。对象类型检测使用 `instanceof`。`null` 或 `undefined` 的检测使用 `== null`。

示例：

```
// string
typeof variable === 'string'

// number
typeof variable === 'number'

// boolean
typeof variable === 'boolean'

// Function
typeof variable === 'function'

// Object
typeof variable === 'object'

// RegExp
variable instanceof RegExp

// Array
variable instanceof Array

// null
variable === null

// null or undefined
variable == null

// undefined
typeof variable === 'undefined'
```

3.4.2 类型转换

[建议] 转换成 `string` 时，使用 `+ ''`。

示例：

```
// good
num + '';

// bad
new String(num);
num.toString();
String(num);
```

[建议] 转换成 `number` 时，通常使用 `+`。

示例：

```
// good
+str;

// bad
Number(str);
```

[建议] `string` 转换成 `number`，要转换的字符串结尾包含非数字并期望忽略时，使用 `parseInt`。

示例：

```
var width = '200px';
parseInt(width, 10);
```

[强制] 使用 `parseInt` 时，必须指定进制。

示例：

```
// good
parseInt(str, 10);

// bad
parseInt(str);
```

[建议] 转换成 `boolean` 时，使用 `!!`。

示例：

```
var num = 3.14;
!!num;
```

[建议] `number` 去除小数点，使用 `Math.floor / Math.round / Math.ceil`，不使用 `parseInt`。

示例：

```
// good
var num = 3.14;
Math.ceil(num);

// bad
var num = 3.14;
parseInt(num, 10);
```

3.5 字符串

[强制] 字符串开头和结束使用单引号 `'`。

解释：

1. 输入单引号不需要按住 shift，方便输入。
2. 实际使用中，字符串经常用来拼接 HTML。为方便 HTML 中包含双引号而不需要转义写法。

示例：

```
var str = '我是一个字符串';
var html = '<div class="cls">拼接HTML可以省去双引号转义</div>';
```

[建议] 使用 `数组` 或 `+` 拼接字符串。

解释：

1. 使用 + 拼接字符串，如果拼接的全部是 StringLiteral，压缩工具可以对其进行自动合并的优化。所以，静态字符串建议使用 + 拼接。
2. 在现代浏览器下，使用 + 拼接字符串，性能较数组的方式要高。
3. 如需要兼顾老旧浏览器，应尽量使用数组拼接字符串。

示例：

```
// 使用数组拼接字符串
var str = [
    // 推荐换行开始并缩进开始第一个字符串，对齐代码，方便阅读.
    '<ul>',
        '<li>第一项</li>',
```

```
        '<li>第二项</li>',
    '</ul>'
].join('');

// 使用 + 拼接字符串
var str2 = '' // 建议第一个为空字符串，第二个换行开始并缩进开始，对齐代码，方便阅读
    + '<ul>',
    +    '<li>第一项</li>',
    +    '<li>第二项</li>',
    + '</ul>';
```

[建议] 复杂的数据到视图字符串的转换过程，选用一种模板引擎。

解释：

使用模板引擎有如下好处：

1. 在开发过程中专注于数据，将视图生成的过程由另外一个层级维护，使程序逻辑结构更清晰。
2. 优秀的模板引擎，通过模板编译技术和高质量的编译产物，能获得比手工拼接字符串更高的性能。

- artTemplate: 体积较小，在所有环境下性能高，语法灵活。
- dot.js: 体积小，在现代浏览器下性能高，语法灵活。
- etpl: 体积较小，在所有环境下性能高，模板复用性高，语法灵活。
- handlebars: 体积大，在所有环境下性能高，扩展性高。
- hogon: 体积小，在现代浏览器下性能高。
- nunjucks: 体积较大，性能一般，模板复用性高。

3.6 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
};
```

[强制] 对象创建时，如果任何一个 属性 需要添加引号，则所有 属性 必须添加 '。

解释：

如果属性不符合 Identifier 和 NumberLiteral 的形式，就需要以 StringLiteral 的形式提供。

示例：

```
// good
var info = {
    'name': 'someone',
    'age': 28,
    'more-info': '...'
};

// bad
var info = {
    name: 'someone',
    age: 28,
    'more-info': '...'
};
```

[强制] 不允许修改和扩展任何原生对象和宿主对象的原型。

示例：

```
// 以下行为绝对禁止
String.prototype.trim = function () {
};
```

[建议] 属性访问时，尽量使用 .。

解释：

属性名符合 Identifier 的要求，就可以通过 . 来访问，否则就只能通过 [expr] 方式访问。

通常在 JavaScript 中声明的对象，属性命名是使用 Camel 命名法，用 . 来访问更清晰简洁。部分特殊的属性(比如来自后端的JSON)，可能采用不寻常的命名方式，可以通过 [expr] 方式访问。

示例：

```
info.age;
info['more-info'];
```

[建议] for in 遍历对象时，使用 hasOwnProperty 过滤掉原型中的属性。

示例：

```
var newInfo = {};
for (var key in info) {
    if (info.hasOwnProperty(key)) {
        newInfo[key] = info[key];
    }
}
```

3.7 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();
```

[强制] 遍历数组不使用 `for in`。

解释：

数组对象可能存在数字以外的属性, 这种情况下 for in 不会得到正确结果.

示例：

```
var arr = ['a', 'b', 'c'];
arr.other = 'other things'; // 这里仅作演示，实际中应使用Object类型

// 正确的遍历方式
for (var i = 0, len = arr.length; i < len; i++) {
    console.log(i);
}

// 错误的遍历方式
for (i in arr) {
    console.log(i);
}
```

[建议] 不因为性能的原因自己实现数组排序功能，尽量使用数组的 `sort` 方法。

解释：

自己实现的常规排序算法，在性能上并不优于数组默认的 sort 方法。以下两种场景可以自己实现排序：

1. 需要稳定的排序算法，达到严格一致的排序结果。
2. 数据特点鲜明，适合使用桶排。

[建议] 清空数组使用 `.length = 0`。

3.8 函数

3.8.1 函数长度

[建议] 一个函数的长度控制在 `50` 行以内。

解释：

将过多的逻辑单元混在一个大函数中，易导致难以维护。一个清晰易懂的函数应该完成单一的逻辑单元。复杂的操作应进一步抽取，通过函数的调用来体现流程。

特定算法等不可分割的逻辑允许例外。

示例：

```
function syncViewStateOnUserAction() {
    if (x.checked) {
        y.checked = true;
        z.value = '';
    }
    else {
        y.checked = false;
    }

    if (!a.value) {
        warning.innerText = 'Please enter it';
        submitButton.disabled = true;
    }
    else {
        warning.innerText = '';
        submitButton.disabled = false;
    }
}

// 直接阅读该函数会难以明确其主线逻辑，因此下方是一种更合理的表达方式：

function syncViewStateOnUserAction() {
    syncXStateToView();
    checkAAvailability();
}

function syncXStateToView() {
    if (x.checked) {
        y.checked = true;
        z.value = '';
```

```
    }
    else {
        y.checked = false;
    }
}

function checkAAvailability() {
    if (!a.value) {
        displayWarningForAMissing();
    }
    else {
        clearWarnignForA();
    }
}
```

3.8.2 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。

[建议] 通过 options 参数传递非数据输入型参数。

解释：

有些函数的参数并不是作为算法的输入，而是对算法的某些分支条件判断之用，此类参数建议通过一个 options 参数传递。

如下函数：

```
/**
 * 移除某个元素
 *
 * @param {Node} element 需要移除的元素
 * @param {boolean} removeEventListeners 是否同时将所有注册在元素上的事件移除
 */
function removeElement(element, removeEventListeners) {
    element.parent.removeChild(element);
    if (removeEventListeners) {
        element.clearEventListeners();
    }
}
```

可以转换为下面的签名：

```
/**
 * 移除某个元素
 *
 * @param {Node} element 需要移除的元素
 * @param {Object} options 相关的逻辑配置
 * @param {boolean} options.removeEventListeners 是否同时将所有注册在元素上的事件移除
 */
function removeElement(element, options) {
    element.parent.removeChild(element);
    if (options.removeEventListeners) {
        element.clearEventListeners();
    }
}
```

这种模式有几个显著的优势：

- boolean 型的配置项具备名称，从调用的代码上更易理解其表达的逻辑意义。
- 当配置项有增长时，无需无休止地增加参数个数，不会出现 removeElement(element, true, false, false, 3) 这样难以理解的调用代码。
- 当部分配置参数可选时，多个参数的形式非常难处理重载逻辑，而使用一个 options 对象只需判断属性是否存在，实现得以简化。

3.8.3 闭包

[建议] 在适当的时候将闭包内大对象置为 `null`。

解释：

在 JavaScript 中，无需特别的关键词就可以使用闭包，一个函数可以任意访问在其定义的作用域外的变量。需要注意的是，函数的作用域是静态的，即在定义时决定，与调用的时机和方式没有任何关系。

闭包会阻止一些变量的垃圾回收，对于较老旧的JavaScript引擎，可能导致外部所有变量均无法回收。

首先一个较为明确的结论是，以下内容会影响到闭包内变量的回收：

- 嵌套的函数中是否有使用该变量。
- 嵌套的函数中是否有 **直接调用eval。**
- 是否使用了 with 表达式。

Chakra、V8 和 SpiderMonkey 将受以上因素的影响，表现出不尽相同又较为相似的回收策略，而JScript.dll和Carakan则完全没有这方面的优化，会完整保留整个 LexicalEnvironment 中的所有变量绑定，造成一定的内存消耗。

由于对闭包内变量有回收优化策略的 Chakra、V8 和 SpiderMonkey 引擎的行为较为相似，因此可以总结如下，当返回一个函数 fn 时：

1. 如果 fn 的 [[Scope]] 是ObjectEnvironment（with 表达式生成 ObjectEnvironment，函数和 catch 表达式生成 DeclarativeEnvironment），则：
    1. 如果是 V8 引擎，则退出全过程。
    2. 如果是 SpiderMonkey，则处理该 ObjectEnvironment 的外层 LexicalEnvironment。

2. 获取当前 LexicalEnvironment 下的所有类型为 Function 的对象，对于每一个 Function 对象，分析其 FunctionBody：
    1. 如果 FunctionBody 中含有 **直接调用eval**，则退出全过程。
    2. 否则得到所有的 Identifier。
    3. 对于每一个 Identifier，设其为 name，根据查找变量引用的规则，从 LexicalEnvironment 中找出名称为 name 的绑定 binding。
    4. 对 binding 添加 notSwap 属性，其值为 true。
3. 检查当前 LexicalEnvironment 中的每一个变量绑定，如果该绑定有 notSwap 属性且值为 true，则：
    1. 如果是V8引擎，删除该绑定。
    2. 如果是SpiderMonkey，将该绑定的值设为 undefined，将删除 notSwap 属性。

对于Chakra引擎，暂无法得知是按 V8 的模式还是按 SpiderMonkey 的模式进行。

如果有 **非常庞大** 的对象，且预计会在 **老旧的引擎** 中执行，则使用闭包时，注意将闭包不需要的对象置为空引用。

[建议] 使用 `IIFE` 避免 `Lift 效应`。

解释：

在引用函数外部变量时，函数执行时外部变量的值由运行时决定而非定义时，最典型的场景如下：

```
var tasks = [];
for (var i = 0; i < 5; i++) {
    tasks[tasks.length] = function () {
        console.log('Current cursor is at ' + i);
    };
}

var len = tasks.length;
while (len--) {
    tasks[len]();
}
```

以上代码对 tasks 中的函数的执行均会输出 `Current cursor is at 5`，往往不符合预期。

此现象称为 **Lift 效应** 。解决的方式是通过额外加上一层闭包函数，将需要的外部变量作为参数传递来解除变量的绑定关系：

```
var tasks = [];
for (var i = 0; i < 5; i++) {
    // 注意有一层额外的闭包
    tasks[tasks.length] = (function (i) {
        return function () {
            console.log('Current cursor is at ' + i);
        };
    })(i);
}
```

```
var len = tasks.length;
while (len--) {
    tasks[len]();
}
```

3.8.4 空函数

[建议] 空函数不使用 `new Function()` 的形式。

示例：

```
var emptyFunction = function () {};
```

[建议] 对于性能有高要求的场合，建议存在一个空函数的常量，供多处使用共享。

示例：

```
var EMPTY_FUNCTION = function () {};

function MyClass() {
}

MyClass.prototype.abstractMethod = EMPTY_FUNCTION;
MyClass.prototype.hooks.before = EMPTY_FUNCTION;
MyClass.prototype.hooks.after = EMPTY_FUNCTION;
```

3.9 面向对象

[强制] 类的继承方案，实现时需要修正 `constructor`。

解释：

通常使用其他 library 的类继承方案都会进行 constructor 修正。如果是自己实现的类继承方案，需要进行 constructor 修正。

示例：

```
/**
 * 构建类之间的继承关系
 *
 * @param {Function} subClass 子类函数
 * @param {Function} superClass 父类函数
 */
function inherits(subClass, superClass) {
    var F = new Function();
    F.prototype = superClass.prototype;
```

```
    subClass.prototype = new F();
    subClass.prototype.constructor = subClass;
}
```

[建议] 声明类时，保证 `constructor` 的正确性。

示例：

```
function Animal(name) {
    this.name = name;
}

// 直接prototype等于对象时，需要修正constructor
Animal.prototype = {
    constructor: Animal,

    jump: function () {
        alert('animal ' + this.name + ' jump');
    }
};

// 这种方式扩展prototype则无需理会constructor
Animal.prototype.jump = function () {
    alert('animal ' + this.name + ' jump');
};
```

[建议] 属性在构造函数中声明，方法在原型中声明。

解释：

原型对象的成员被所有实例共享，能节约内存占用。所以编码时我们应该遵守这样的原则：原型对象包含程序不会修改的成员，如方法函数或配置项。

```
function TextNode(value, engine) {
    this.value = value;
    this.engine = engine;
}

TextNode.prototype.clone = function () {
    return this;
};
```

[强制] 自定义事件的 事件名 必须全小写。

解释：

在 JavaScript 广泛应用的浏览器环境，绝大多数 DOM 事件名称都是全小写的。为了遵循大多数 JavaScript 开发者的习惯，在设计自定义事件时，事件名也应该全小写。

[强制] 自定义事件只能有一个 `event` 参数。如果事件需要传递较多信息，应仔细设计事件对象。

解释：

一个事件对象的好处有：

1. 顺序无关，避免事件监听者需要记忆参数顺序。
2. 每个事件信息都可以根据需要提供或者不提供，更自由。
3. 扩展方便，未来添加事件信息时，无需考虑会破坏监听器参数形式而无法向后兼容。

[建议] 设计自定义事件时，应考虑禁止默认行为。

解释：

常见禁止默认行为的方式有两种：

1. 事件监听函数中 return false。
2. 事件对象中包含禁止默认行为的方法，如 preventDefault。

3.10 动态特性

3.10.1 eval

[强制] 避免使用直接 `eval` 函数。

解释：

直接 eval，指的是以函数方式调用 eval 的调用方法。直接 eval 调用执行代码的作用域为本地作用域，应当避免。

如果有特殊情况需要使用直接 eval，需在代码中用详细的注释说明为何必须使用直接 eval，不能使用其它动态执行代码的方式，同时需要其他资深工程师进行 Code Review。

[建议] 尽量避免使用 `eval` 函数。

3.10.2 动态执行代码

[建议] 使用 `new Function` 执行动态代码。

解释：

通过 new Function 生成的函数作用域是全局使用域，不会影响当当前的本地作用域。如果有动态代码执行的需求，建议使用 new Function。

示例：

```
var handler = new Function('x', 'y', 'return x + y;');
var result = handler($('#x').val(), $('#y').val());
```

3.10.3 with

[建议] 尽量不要使用 `with`。

解释：

使用 with 可能会增加代码的复杂度，不利于阅读和管理；也会对性能有影响。大多数使用 with 的场景都能使用其他方式较好的替代。所以，尽量不要使用 with。

3.10.4 delete

[建议] 减少 `delete` 的使用。

解释：

如果没有特别的需求，减少或避免使用`delete`。`delete`的使用会破坏部分 JavaScript 引擎的性能优化。

[建议] 处理 `delete` 可能产生的异常。

解释：

对于有被遍历需求，且值 null 被认为具有业务逻辑意义的值的对象，移除某个属性必须使用 delete 操作。

在严格模式或IE下使用 delete 时，不能被删除的属性会抛出异常，因此在不确定属性是否可以删除的情况下，建议添加 try-catch 块。

示例：

```
try {
    delete o.x;
}
catch (deleteError) {
    o.x = null;
}
```

3.10.5 对象属性

[建议] 避免修改外部传入的对象。

解释：

JavaScript 因其脚本语言的动态特性，当一个对象未被 seal 或 freeze 时，可以任意添加、删除、修改属性值。

但是随意地对 非自身控制的对象 进行修改，很容易造成代码在不可预知的情况下出现问题。因此，设计良好的组件、函数应该避免对外部传入的对象的修改。

下面代码的 selectNode 方法修改了由外部传入的 datasource 对象。如果 datasource 用在其它场合（如另一个 Tree 实例）下，会造成状态的混乱。

```
function Tree(datasource) {
    this.datasource = datasource;
}

Tree.prototype.selectNode = function (id) {
    // 从datasource中找出节点对象
    var node = this.findNode(id);
    if (node) {
        node.selected = true;
        this.flushView();
    }
};
```

对于此类场景，需要使用额外的对象来维护，使用由自身控制，不与外部产生任何交互的 selectedNodeIndex 对象来维护节点的选中状态，不对 datasource 作任何修改。

```
function Tree(datasource) {
    this.datasource = datasource;
    this.selectedNodeIndex = {};
}

Tree.prototype.selectNode = function (id) {
    // 从datasource中找出节点对象
    var node = this.findNode(id);
    if (node) {
        this.selectedNodeIndex[id] = true;
        this.flushView();
    }
};
```

除此之外，也可以通过 deepClone 等手段将自身维护的对象与外部传入的分离，保证不会相互影响。

[建议] 具备强类型的设计。

解释：

- 如果一个属性被设计为 boolean 类型，则不要使用 1 / 0 作为其值。对于标识性的属性，如对代码体积有严格要求，可以从一开始就设计为 number 类型且将 0 作为否定值。
- 从 DOM 中取出的值通常为 string 类型，如果有对象或函数的接收类型为 number 类型，提前作好转换，而不是期望对象、函数可以处理多类型的值。

4 浏览器环境

4.1 模块化

4.1.1 AMD

[强制] 使用 `AMD` 作为模块定义。

解释：

AMD 作为由社区认可的模块定义形式，提供多种重载提供灵活的使用方式，并且绝大多数优秀的 Library 都支持 AMD，适合作为规范。

目前，比较成熟的 AMD Loader 有：

- 官方实现的 requirejs
- 百度自己实现的 esl

[强制] 模块 `id` 必须符合标准。

解释：

模块 id 必须符合以下约束条件：

1. 类型为 string，并且是由 `/` 分割的一系列 terms 来组成。例如：`this/is/a/module`。
2. term 应该符合 [a-zA-Z0-9_-]+ 规则。
3. 不应该有 .js 后缀。
4. 跟文件的路径保持一致。

4.1.2 define

[建议] 定义模块时不要指明 `id` 和 `dependencies`。

解释：

在 AMD 的设计思想里，模块名称是和所在路径相关的，匿名的模块更利于封包和迁移。模块依赖应在模块定义内部通过 local require 引用。

所以，推荐使用 define(factory) 的形式进行模块定义。

示例：

```
define(
    function (require) {
    }
);
```

[建议] 使用 `return` 来返回模块定义。

解释：

使用 return 可以减少 factory 接收的参数（不需要接收 exports 和 module），在没有 AMD Loader 的场景下也更容易进行简单的处理来伪造一个 Loader。

示例：

```
define(
    function (require) {
        var exports = {};

        // ...

        return exports;
    }
);
```

4.1.3 require

[强制] 全局运行环境中，`require` 必须以 `async require` 形式调用。

解释：

模块的加载过程是异步的，同步调用并无法保证得到正确的结果。

示例：

```
// good
require(['foo'], function (foo) {
});

// bad
var foo = require('foo');
```

[强制] 模块定义中只允许使用 `local require`，不允许使用 `global require`。

解释：

1. 在模块定义中使用 global require，对封装性是一种破坏。
2. 在 AMD 里，global require 是可以被重命名的。并且 Loader 甚至没有全局的 require 变量，而是用 Loader 名称做为 global require。模块定义不应该依赖使用的 Loader。

[强制] Package在实现时，内部模块的 `require` 必须使用 `relative id`。

解释：

对于任何可能通过 发布-引入 的形式复用的第三方库、框架、包，开发者所定义的名称不代表使用者使用的名称。因此不要基于任何名称的假设。在实现源码中，require 自身的其它模块时使用 relative id。

示例：

```
define(
    function (require) {
        var util = require('./util');
    }
);
```

[建议] 不会被调用的依赖模块，在 `factory` 开始处统一 `require`。

解释：

有些模块是依赖的模块，但不会在模块实现中被直接调用，最为典型的是 css / js / tpl 等 Plugin 所引入的外部内容。此类内容建议放在模块定义最开始处统一引用。

示例：

```
define(
    function (require) {
        require('css!foo.css');
        require('tpl!bar.tpl.html');

        // ...
    }
);
```

4.2 DOM

4.2.1 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。

示例：

```
<div></div>
<span></span>

<script>
var elements = document.getElementsByTagName('*');
```

```
// 显示为 DIV
alert(elements[0].tagName);

var div = elements[0];
var p = document.createElement('p');
document.body.insertBefore(p, div);

// 显示为 P
alert(elements[0].tagName);
</script>
```

[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。

4.2.2 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。

4.2.3 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。

4.2.4 DOM 操作

[建议] 操作 `DOM` 时，尽量减少页面 `reflow`。

解释：

页面 reflow 是非常耗时的行为，非常容易导致性能瓶颈。下面一些场景会触发浏览器的reflow：

- DOM元素的添加、修改（内容）、删除。
- 应用新的样式或者修改任何影响元素布局的属性。
- Resize浏览器窗口、滚动页面。
- 读取元素的某些属性（offsetLeft、offsetTop、offsetHeight、offsetWidth、scrollTop/Left/Width/Height、clientTop/Left/Width/Height、getComputedStyle()、currentStyle(in IE)）。

[建议] 尽量减少 `DOM` 操作。

解释：

DOM 操作也是非常耗时的一种操作，减少 DOM 操作有助于提高性能。举一个简单的例子，构建一个列表。我们可以用两种方式：

1. 在循环体中 createElement 并 append 到父元素中。
2. 在循环体中拼接 HTML 字符串，循环结束后写父元素的 innerHTML。

第一种方法看起来比较标准，但是每次循环都会对 DOM 进行操作，性能极低。在这里推荐使用第二种方法。

4.2.5 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。