HTML代码规范

整体结构

1. 统一使用HTML5文档类型：<!DOCTYPE html></li>
2. 启用 IE Edge 模式 <meta http-equiv="X-UA-Compatible" content="IE=Edge">
3. 在 head 中引入页面需要的所有 CSS 资源。

在页面渲染的过程中，新的CSS可能导致元素的样式重新计算和绘制，页面闪烁。

1. 必须在mate中声明编码类型：<charset="utf-8"/>
2. 自定义属性统一通过 data-\* 方式来添加：<div data-userid=…>
3. CSS文件引入必须放在 head 中， script 必须放在 body 底部
4. 对于非跳转的a标签: href="javascript:;"
5. script标签不要添加 type="text/javascript" 类型属性
6. link标签不要添加 type="text/css" 类型属性

代码格式

1. 每行不得超过屏幕换行
2. 标签和属性必须统一使用小写
3. 标签必须按照标准嵌套，比如内联元素不能包含块级元素等等
4. 合理的使用标签，如 标题根据重要性用h1~h6(同一页面只能有一个h1), 段落标记用p, 列表用ul，参考后面的常用标签
5. 对于无需自闭合的标签，不允许自闭合。
6. 在 CSS 可以实现相同需求的情况下不得使用表格进行布局。
7. 标签的使用应尽量简洁，减少不必要的标签。

Copy

<!-- good --> <img class="avatar" src="image.png"> <!-- bad --> <span class="avatar"> <img src="image.png"> </span>

注释规范

1. 采用类似标签闭合的写法，与HTML统一格式；注释文案两头空格，与CSS注释统一格式。
   * 开始注释：<!-- 注释文案 -->（文案两头空格）。
   * 结束注释：<!-- /注释文案 -->
   * 允许只有开始注释！

Copy

<!-- 头部 --> <div class="header"> <!-- LOGO --> <h1 class="logo"><a href="#">LOGO</a></h1> <!-- /LOGO --> <!-- 导航 --> <ul class="nav"> <li><a href="#">NAV1</a></li> <li><a href="#">NAV2</a></li> <!-- 更多导航项 --> </ul> <!-- /导航 --> </div> <!-- /头部 -->

命名

1. class 必须单词全字母小写，单词间以 - 分隔。
2. class 必须代表相应模块或部件的内容或功能，不得以样式信息进行命名。
3. 元素 id 必须保证页面唯一。
4. id 建议单词全字母小写，单词间以 - 分隔。同项目必须保持风格一致。
5. id、class 命名，在避免冲突并描述清楚的前提下尽可能短。

Copy

<!-- good --> <div id="nav"></div> <!-- bad --> <div id="navigation"></div> <!-- good --> <p class="comment"></p> <!-- bad --> <p class="com"></p> <!-- good --> <span class="author"></span> <!-- bad --> <span class="red"></span>

1. 禁止为了 hook 脚本，创建无样式信息的 class。

不允许 class 只用于让 JavaScript 选择某些元素，class 应该具有明确的语义和样式。否则容易导致 css class 泛滥。 使用 id、属性选择作为 hook 是更好的方式。

1. 同一页面，应避免使用相同的 name 与 id。

CSS代码规范

样式命名规范

1. 命名中禁止出现大写字母
2. 使用 - 对单词进行分割，如 .box-head{ ….}
3. 多个选择器之间必须换行

Copy

/\* 推荐 \*/ a.btn, input.btn, input[type="button"] { ... }

1. 连接伪类样式按照顺序书写

Copy

a.link {} /\* 正常情况下的样式 \*/ a.visited {} /\* 已访问的样式 \*/ a.hover {} /\* 鼠标移入时的样式 \*/ a.active {} /\* 鼠标按下时的样式 \*/

1. 命名避免使用中文拼音，应该采用更简明有语义的英文单词进行组合
2. 命名注意缩写，但是不能盲目缩写，具体请参见最后的常用CSS命名规则
3. 禁止使用id选择器
4. 命名中尽量不要使用数字，例如 .list2 {..}
5. 不要将选择器写的过于冗长，这会额外增加文件大小并且限制了太小范围的选择器，使树形结构过于严格应用范围过于局限，建议3-4个长度之内写完。选择器并不需要完整反映结构嵌套顺序，相反，能简则简。
6. 样式针对特定html结构时，使用元素名+类名 ul.nav {….}

属性规范

1. 建议属性按顺序书写

Copy

.box-content { /\* 定位, 常见的有: position / top / right / bottom / left / float / display / overflow 等 \*/ display: block; position: absolute; left: 0; top: 0; overflow-x: hidden; /\* 盒模型, 常见的有: border / margin / padding / width / height 等 \*/ width: 50px; height: 50px; margin: 10px; padding: 5px; border: 1px solid black; /\* 排版, 常见的有: font / line-height / text-align / word-wrap 等 \*/ font-size: 14px; line-height: 20px; /\* 视觉, 常见的有: background / color / transition / list-style 等 \*/ background: #f5f5f5; color: #333; -webkit-transition: color 1s; -moz-transition: color 1s; transition: color 1s; /\* 按照这样的顺序书写可以提神浏览器渲染性能 \*/ }

1. 每个属性后面必须要有分号
2. 必须去除零值后面单位，例如 { padding:0; }
3. 对于需要拖动的元素，必须禁用选中文本
4. 不建议使用hack修复兼容性问题，尽量通过合理的布局来解决
5. 不建议使用filter，对于圆角这类效果采用优雅降级策略
6. 禁止使用expression
7. 单位必须统一使用px（像素）
8. url(image.jpg) 路径中不需要引号，同时文件名中不能有空格
9. 能简写的尽量采用简写，特殊样式重用的情况下除外

Copy

body { margin-top: 2px; margin-right: 5px; margin-bottom: 2px; margin-left: 15px; /\* 简写 \*/ margin: 2px 5px 2px 15px; border-width: 1px; border-style: solid; border-color: #000; /\* 简写 \*/ border: 1px solid #000; }

注释规范

1. 头部注释：文件头部需要说明当前css文件的功能及作用范围

Copy

/\*\* \* 公用样式，样式充值、公用规则、miniui样式修正等最常用的样式 \* 适用范围：所有页面 \*/

1. 区块注释：针对功能或者布局的代码块使用region区域注释包裹起来（在VS中能够使用折叠功能）

Copy

/\* #region 按项目定制通用样式 \*/ .blue { color: #05a; } .red { color: #f00; } .gray { color: #a7a7a7; } /\* #endregion \*/

其他

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。
2. Css文件中禁止使用 @charset
3. 如果CSS可以做到的效果，就不要使用JS
   * 让CSS做更多的事，减轻JS开发量
   * 用CSS控制交互或视觉的变化，JS只需要更改className
   * 利用CSS一次性更改多个节点样式，避免多次渲染，提高渲染效率
4. 禁止使用 import 引入 css文件
5. 建议小图标使用矢量字体或sprite合并
6. 使用统一的reset样式
7. 如果引用字体库，需要考虑各个浏览器兼容问题
8. 禁止使用 !important
9. 注意css是从右到左匹配的
10. 选择器的效率是： class > tag > \*。
11. 素材图片格式尽量采用png， 大幅图片采用jpg

常用命名参考

1. 页面结构
   * 容器：container
   * 页头：header
   * 内容：content/container
   * 页面主体：main
   * 页尾：footer
   * 导航：nav
   * 侧栏：sidebar
   * 栏目：column
   * 页面外围控制整体布局宽度：wrapper
   * 左右中：left right center
2. 导航
   * 导航：nav
   * 主导航：nav-main
   * 子导航：nav-sub
   * 顶导航：nav-top
   * 边导航：sidebar
   * 左导航：sidebar-left
   * 右导航：sidebar-right
   * 菜单：menu
   * 子菜单：menu-sub
   * 标题: title
   * 摘要: summary
3. 功能
   * 标志：logo
   * 广告：banner
   * 登陆：login
   * 登录条：login-bar
   * 注册：register
   * 搜索：search
   * 功能区：shop
   * 标题：title
   * 加入：joinus
   * 状态：status
   * 按钮：btn
   * 滚动：scroll
   * 标签页：tab
   * 文章列表：list
   * 提示信息：msg
   * 当前的: current
   * 小技巧： tips
   * 图标：icon
   * 注释：note
   * 指南：guild
   * 服务：service
   * 热点：hot
   * 新闻：news
   * 下载：download
   * 投票：vote
   * 合作伙伴：partner
   * 友情链接：link
   * 版权：copyright
4. 状态
   * 选中 selected
   * 当前 current
   * 显示 show
   * 隐藏 hide
   * 打开 open
   * 关闭 close
   * 出错 error
   * 禁用 disabled

浏览器支持

| **浏览器** | **版本** |
| --- | --- |
| IE | ie8+ |
| Firefox | 最新版 |
| Safari | 最新版 |
| Chrome | 最新版 |

浏览器特性支持

* <http://fmbip.com/litmus>
* <http://caniuse.com/>

# JavaScript代码规范

### 代码风格

#### 文件

1. JavaScript 文件使用无 BOM 的 UTF-8 编码

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。

1. 在文件结尾处，保留一个空行

#### 结构

1. 缩进

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

1. switch 下的 case 和 default 必须增加一个缩进层级

Copy

// 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... }

#### 空格

1. 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

Copy

var a = !arr.length; a++; a = b + c;

1. 用作代码块起始的左花括号 { 前必须有一个空格

Copy

// good if (condition) { } while (condition) { } function funcName() { } // bad if (condition){ } while (condition){ } function funcName(){ }

1. if / else / for / while / function / switch / do / try / catch / finally 关键字后，必须有一个空格

Copy

// good if (condition) { } while (condition) { } (function () { })(); // bad if(condition) { } while(condition) { } (function() { })();

1. 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

Copy

// good var obj = { a: 1, b: 2, c: 3 }; // bad var obj = { a : 1, b:2, c :3 };

1. 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

Copy

// good function funcName() { } var funcName = function funcName() { }; funcName(); // bad function funcName () { } var funcName = function funcName () { }; funcName ();

1. ,和; 前不允许有空格

Copy

// good callFunc(a, b); // bad callFunc(a , b) ;

1. 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

Copy

// 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-- ) { }

1. 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
2. 行尾不得有多余的空格

#### 换行

1. 每个独立语句结束后必须换行
2. 每行不得超过 120 个字符

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

1. 运算符处换行时，运算符必须在新行的行首

Copy

// 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;

1. 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

Copy

// 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 );

1. 不同行为或逻辑的语句集，使用空行隔开，更易阅读

Copy

// 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; }

1. 在语句的行长度超过 120 时，根据逻辑条件合理缩进

Copy

// 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>' ].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 = [ { // ... }, { // ... } ];

#### 语句

1. 不得省略语句结束的分号
2. 在 if / else / for / do / while 语句中，即使只有一行，也不得省略块 {...}
3. 函数定义结束不允许添加分号

Copy

// 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 () { };

1. 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 (

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。

Copy

// good var task = (function () { // Code return result; })(); var func = function () { }; // bad var task = function () { // Code return result; }(); var func = (function () { });

#### 命名

1. 变量 使用 Camel 命名法

Copy

var loadingModules = {};

1. 常量 使用 全部字母大写，单词间下划线分隔 的命名方式

Copy

var HTML\_ENTITY = {};

1. 函数 使用 Camel命名法

Copy

function stringFormat(source) {}

1. 函数的 参数 使用 Camel命名法

Copy

function hear(theBells) {}

1. 类 使用 Pascal命名法

Copy

function TextNode(options) {}

1. 类的 方法 / 属性 使用 Camel命名法

Copy

function TextNode(value, engine) { this.value = value; this.engine = engine; } TextNode.prototype.clone = function () { return this; };

1. 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

Copy

var TargetState = { READING: 1, READED: 2, APPLIED: 3, READY: 4 };

1. 命名空间 使用 Camel命名法

Copy

equipments.heavyWeapons = {};

1. 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

Copy

function XMLParser() { } function insertHTML(element, html) { } var httpRequest = new HTTPRequest();

1. 类名 使用 名词

Copy

function Engine(options) {}

1. 函数名 使用 动宾短语

Copy

function getStyle(element) {}

1. boolean 类型的变量使用 is 或 has 开头

Copy

var isReady = false; var hasMoreCommands = false;

1. Promise对象 用 动宾短语的进行时 表达

Copy

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

### 语言特性

#### 变量

1. 变量在使用前必须通过 var 定义
2. 每个 var 只能声明一个变量
3. 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

#### 条件

1. 在 Equality Expression 中使用类型严格的 ===。仅当判断 null 或 undefined 时，允许使用 == null
2. 尽可能使用简洁的表达式

Copy

// 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') { // ...... }

1. 按执行频率排列分支的顺序

按执行频率排列分支的顺序好处是

* + 阅读的人容易找到最常见的情况，增加可读性
  + 提高执行效率

1. 对于相同变量或表达式的多值条件，用 switch 代替 if

Copy

// 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') { // ...... }

1. 如果函数或全局中的 else 块后没有任何语句，可以删除 else

Copy

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

#### 循环

1. 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

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

Copy

// 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 () {}); }

1. 对循环内多次使用的不变值，在循环外用变量缓存

Copy

// 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'; // ...... }

1. 对有序集合进行遍历时，缓存 length

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。

Copy

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

1. 对有序集合进行顺序无关的遍历时，使用逆序遍历

逆序遍历可以节省变量，代码比较优化

Copy

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

#### 类型

1. 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

Copy

// 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'

#### 类型转换

1. 转换成 string 时，使用 + ''

Copy

// good num + ''; // bad new String(num); num.toString(); String(num);

1. 转换成 number 时，通常使用 +

Copy

// good +str; // bad Number(str);

1. 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

Copy

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

1. 使用 parseInt 时，必须指定进制

Copy

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

1. 转换成 boolean 时，使用 !!

Copy

var num = 3.14; !!num;

1. number 去除小数点，使用 Math.floor / Math.round / Math.ceil，不使用 parseInt

Copy

// good var num = 3.14; Math.ceil(num); // bad var num = 3.14; parseInt(num, 10);

#### 字符串

1. 字符串开头和结束使用单引号 '
   * 输入单引号不需要按住 shift，方便输入
   * 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

Copy

var str = '我是一个字符串'; var html = '

拼接HTML可以省去双引号转义

';

1. 使用 数组 或 + 拼接字符串
   * 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
   * 在现代浏览器下，使用 + 拼接字符串，性能较数组的方式要高
   * 如需要兼顾老旧浏览器，应尽量使用数组拼接字符串

Copy

// 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>';

1. 复杂的数据到视图字符串的转换过程，选用一种模板引擎

#### 对象

1. 使用对象字面量 {} 创建新 Object
2. 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
3. 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 '

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

1. 不允许修改和扩展任何原生对象和宿主对象的原型

Copy

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

1. 属性访问时，尽量使用 .

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

1. for in 遍历对象时, 使用 hasOwnProperty 过滤掉原型中的属性

Copy

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

### 数组

1. 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

Copy

// good var arr = []; // bad var arr = new Array();

1. 遍历数组不使用 for in

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

Copy

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); }

1. 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

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：

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

1. 清空数组使用 .length = 0

#### 函数

1. 一个函数的长度控制在 50 行以内

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

#### 参数设计

1. 一个函数的参数控制在 6 个以内

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

1. 通过 options 参数传递非数据输入型参数

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。

#### 闭包

1. 在适当的时候将闭包内大对象置为 null
2. 使用 IIFE 避免 Lift 效应

#### 空函数

1. 空函数不使用 new Function() 的形式

Copy

var emptyFunction = function () {};

1. 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

Copy

var EMPTY\_FUNCTION = function () {}; function MyClass() { } MyClass.prototype.abstractMethod = EMPTY\_FUNCTION; MyClass.prototype.hooks.before = EMPTY\_FUNCTION; MyClass.prototype.hooks.after = EMPTY\_FUNCTION;

### 面向对象

1. 类的继承方案，实现时需要修正 constructor

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。

Copy

/\*\* \* 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; }

1. 声明类时，保证 constructor 的正确性

Copy

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'); };

1. 属性在构造函数中声明，方法在原型中声明

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

Copy

function TextNode(value, engine) { this.value = value; this.engine = engine; } TextNode.prototype.clone = function () { return this; };

1. 自定义事件的 事件名 必须全小写

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

1. 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

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

1. 计自定义事件时，应考虑禁止默认行为：

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

### 动态特性

#### eval

1. 避免使用直接 eval 函数

#### 动态执行代码

1. 使用 new Function 执行动态代码

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

Copy

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

#### with

1. 尽量不要使用 with

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

#### delete

1. 减少 delete 的使用

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

1. 处理 delete 可能产生的异常

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

Copy

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

#### 对象属性

1. 避免修改外部传入的对象

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

Copy

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

Copy

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。

1. 具备强类型的设计

# JavaScript注释规范

### 注释

#### 单行注释

1. 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

#### 多行注释

1. 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

#### 文档化注释

1. 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
   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
2. 文档注释前必须空一行
3. 自文档化的文档说明 what，而不是 how

#### 类型定义

1. 类型定义都是以{开始, 以}结束

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

1. 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 |

#### 文件注释

1. 文件顶部必须包含文件注释，用 @file 标识文件说明

Copy

/\*\* \* @file Describe the file \*/

1. 文件注释中可以用 @author 标识开发者信息

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。

Copy

/\*\* \* @file Describe the file \* @author author-name(mail-name@domain.com) \* author-name2(mail-name2@domain.com) \*/

#### 命名空间注释

1. 命名空间使用 @namespace 标识

Copy

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

#### 类注释

1. 使用 @class 标记类或构造函数

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。

Copy

/\*\* \* 描述 \* \* @class \*/ function Developer() { // constructor body }

1. 使用 @extends 标记类的继承信息

Copy

/\*\* \* 描述 \* \* @class \* @extends Developer \*/ function Fronteer() { Developer.call(this); // constructor body } util.inherits(Fronteer, Developer);

1. 使用包装方式扩展类成员时， 必须通过 @lends 进行重新指向

没有 @lends 标记将无法为该类生成包含扩展类成员的文档

Copy

/\*\* \* 类描述 \* \* @class \* @extends Developer \*/ function Fronteer() { Developer.call(this); // constructor body } utility.extend( Fronteer.prototype, /\*\* @lends Fronteer.prototype \*/{ \_getLevel: function () { // TODO } } );

1. 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

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

Copy

/\*\* \* 类描述 \* \* @class \* @extends Developer \*/ var Fronteer = function () { Developer.call(this); /\*\* \* 属性描述 \* \* @type {string} \* @private \*/ this.\_level = 'T12'; // constructor body }; /\*\* \* 方法描述 \* \* @private \* @return {string} 返回值描述 \*/ Fronteer.prototype.\_getLevel = function () { };

#### 函数/方法注释

1. 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
2. 参数和返回值注释必须包含类型信息和说明
3. 当函数是内部函数，外部不可访问时，可以使用 @inner 标识

Copy

/\*\* \* 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 }; }

1. 对 Object 中各项的描述， 必须使用 @param 标识

Copy

/\*\* \* 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 }

1. 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

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

#### 事件注释

1. 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

Copy

/\*\* \* 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) { }

1. 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
2. 对于事件对象的注释，使用 @param 标识，生成文档时可读性更好

Copy

/\*\* \* 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' } ); };

#### 常量注释

1. 常量必须使用 @const 标记，并包含说明和类型信息

Copy

/\*\* \* 常量说明 \* \* @const \* @type {string} \*/ var REQUEST\_URL = 'myurl.do';

#### 复杂类型注释

1. 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

Copy

// `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 } ];

#### AMD 模块注释

1. AMD 模块使用 @module 或 @exports 标识

@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

Copy

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

Copy

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：

Copy

/\*\* \* module description. \* \* @module \*/ define( function (require, exports) { /\*\* \* bar description \* \*/ exports.bar = function () { // TODO }; return exports; } );

1. 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

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

Copy

/\*\* \* 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' } ); };

1. 对于类定义的模块，可以使用 @alias 标识构建函数

Copy

/\*\* \* A module representing a jacket. \* @module jacket \*/ define( function () { /\*\* \* @class \* @alias module:jacket \*/ var Jacket = function () { }; return Jacket; } );

1. 多模块定义时，可以使用 @exports 标识各个模块

Copy

// 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; } );

1. 对于 exports 为 Object 的模块，可以使用@namespace标识

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

1. 对于 exports 为类名的模块，使用 @class 和 @exports 标识

Copy

// 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 };

#### 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

1. 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

Copy

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 (...) { .... } }

1. 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
   * 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。
   * 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。
   * 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。
   * XXX: 该处存在陷阱。此时需要对陷阱进行描述。