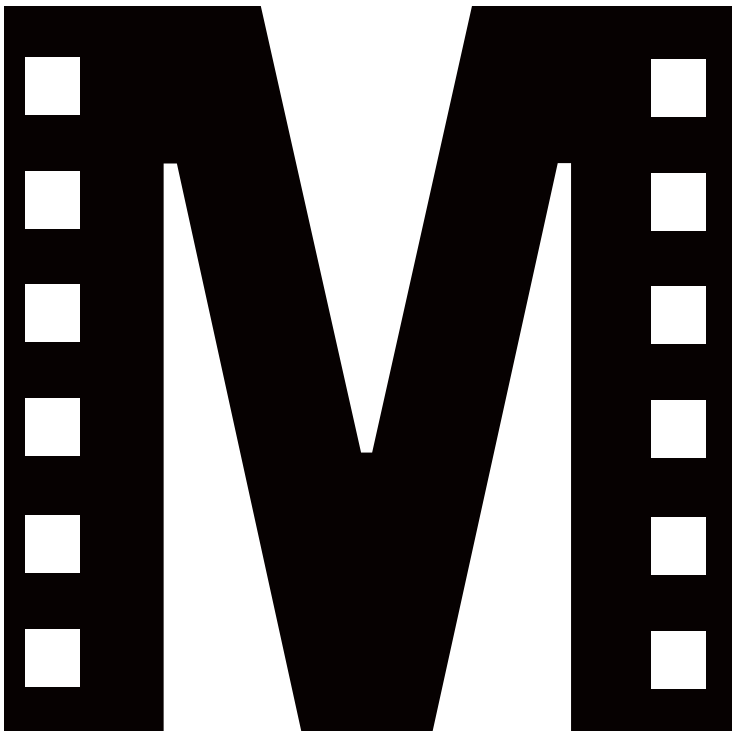

GDPLS Movie

Grand Duke of Programming Language Script

2017 June

Code Style

Generated & Compiled By XeLaTeX

HTML编码规范

1 前言

HTML作为描述网页结构的超文本标记语言，在百度一直有着广泛的应用。本文档的目标是使HTML代码风格保持一致，容易被理解和被维护。

2 代码风格

2.1 缩进与换行

[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。

示例：

```
<ul>
    <li>first</li>
    <li>second</li>
</ul>
```

[建议] 每行不得超过 120 个字符。

解释：

过长的代码不容易阅读与维护。但是考虑到 HTML 的特殊性，不做硬性要求。

2.2 命名

[强制] class 必须单词全字母小写，单词间以 - 分隔。

[强制] 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。

示例：

```
<!-- good -->
<div class="sidebar"></div>

<!-- bad -->
<div class="left"></div>
```

[强制] 元素 id 必须保证页面唯一。

解释：

同一个页面中，不同的元素包含相同的 id，不符合 id 的属性含义。并且使用 document.getElementById 时可能导致难以追查的问题。

[建议] 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。

[建议] id、class 命名，在避免冲突并描述清楚的前提下尽可能短。

示例：

```
<!-- 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>
```

[强制] 禁止为了 hook 脚本，创建无样式信息的 class。

解释：

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。

[强制] 同一页面，应避免使用相同的 name 与 id。

解释：

IE 浏览器会混淆元素的 id 和 name 属性， document.getElementById
可能获得不期望的元素。所以在对元素的 id 与 name 属性的命名需要非常小心。

一个比较好的实践是，为 id 和 name 使用不同的命名法。

示例：

```
<input name="foo">
<div id="foo"></div>
<script>
// IE6 将显示 INPUT
alert(document.getElementById('foo').tagName);
</script>
```

2.3 标签

[强制] 标签名必须使用小写字母。

示例：

```
<!-- good -->
<p>Hello StyleGuide!</p>

<!-- bad -->
<P>Hello StyleGuide!</P>
```

[强制] 对于无需自闭合的标签，不允许自闭合。

解释：

常见无需自闭合标签有input、br、img、hr等。

示例：

```
<!-- good -->
<input type="text" name="title">

<!-- bad -->
<input type="text" name="title" />
```

[强制] 对 HTML5 中规定允许省略的闭合标签，不允许省略闭合标签。

解释：

对代码体积要求非常严苛的场景，可以例外。比如：第三方页面使用的投放系统。

示例：

```
<!-- good -->
<ul>
    <li>first</li>
    <li>second</li>
</ul>

<!-- bad -->
<ul>
    <li>first
    <li>second
</ul>
```

[强制] 标签使用必须符合标签嵌套规则。

解释：

比如 div 不得置于 p 中，tbody 必须置于 table 中。

详细的标签嵌套规则参见HTML DTD中的 `Elements` 定义部分。

[建议] HTML 标签的使用应该遵循标签的语义。

解释：

下面是常见标签语义

- p - 段落
- h1,h2,h3,h4,h5,h6 - 层级标题

- strong,em - 强调
- ins - 插入
- del - 删除
- abbr - 缩写
- code - 代码标识
- cite - 引述来源作品的标题
- q - 引用
- blockquote - 一段或长篇引用
- ul - 无序列表
- ol - 有序列表
- dl,dt,dd - 定义列表

示例：

```
<!-- good -->
<p>Esprima serves as an important <strong>building block</strong> for
some JavaScript language tools.</p>

<!-- bad -->
<div>Esprima serves as an important <span class="strong">building
block</span> for some JavaScript language tools.</div>
```

[建议] 在 CSS 可以实现相同需求的情况下不得使用表格进行布局。

解释：

在兼容性允许的情况下应尽量保持语义正确性。对网格对齐和拉伸性有严格要求的场景允许例外，如多列复杂表单。

[建议] 标签的使用应尽量简洁，减少不必要的标签。

示例：

```
<!-- good -->
<img class="avatar" src="image.png">

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

2.4 属性

[强制] 属性名必须使用小写字母。

示例：

```
<!-- good -->
<table cellspacing="0">...</table>

<!-- bad -->
<table cellSpacing="0">...</table>
```

[强制] 属性值必须用双引号包围。

解释：

不允许使用单引号，不允许不使用引号。

示例：

```
<!-- good -->
<script src="esl.js"></script>

<!-- bad -->
<script src='esl.js'></script>
<script src=esl.js></script>
```

[建议] 布尔类型的属性，建议不添加属性值。

示例：

```
<input type="text" disabled>
<input type="checkbox" value="1" checked>
```

[建议] 自定义属性建议以 xxx- 为前缀，推荐使用 data-。

解释：

使用前缀有助于区分自定义属性和标准定义的属性。

示例：

```
<ol data-ui-type="Select"></ol>
```

3 通用

3.1 DOCTYPE

[强制] 使用 HTML5 的 doctype 来启用标准模式，建议使用大写的 DOCTYPE。

示例：

```
<!DOCTYPE html>
```

[建议] 启用 IE Edge 模式。

示例：

```
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
```

[建议] 在 html 标签上设置正确的 lang 属性。

解释：

有助于提高页面的可访问性，如：让语音合成工具确定其所应该采用的发音，令翻译工具确定其翻译语言等。

示例：

```
<html lang="zh-CN">
```

3.2 编码

[强制] 页面必须使用精简形式，明确指定字符编码。指定字符编码的 meta 必须是 head 的第一个直接子元素。

解释：

见 HTML5 Charset能用吗 一文。

示例：

```
<html>
    <head>
        <meta charset="UTF-8">
        ......
    </head>
    <body>
        ......
    </body>
</html>
```

[建议] HTML 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。

3.3 CSS和JavaScript引入

[强制] 引入 CSS 时必须指明 rel="stylesheet"。

示例：

```
<link rel="stylesheet" src="page.css">
```

[建议] 引入 CSS 和 JavaScript 时无须指明 type 属性。

解释：

text/css 和 text/javascript 是 type 的默认值。

[建议] 展现定义放置于外部 CSS 中，行为定义放置于外部 JavaScript 中。

解释：

结构-样式-行为的代码分离，对于提高代码的可阅读性和维护性都有好处。

[建议] 在 head 中引入页面需要的所有 CSS 资源。

解释：

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。

[建议] JavaScript 应当放在页面末尾，或采用异步加载。

解释：

将 script
放在页面中间将阻断页面的渲染。出于性能方面的考虑，如非必要，请遵守此条建议。

示例：

```
<body>
    <!-- a lot of elements -->
    <script src="init-behavior.js"></script>
</body>
```

[建议] 移动环境或只针对现代浏览器设计的 Web 应用，如果引用外部资源的 URL 协议部分与页面相同，建议省略协议前缀。

解释：

使用 protocol-relative URL 引入 CSS，在 IE7/8 下，会发两次请求。是否使用 protocol-relative URL 应充分考虑页面针对的环境。

示例：

```
<script src="//s1.bdstatic.com/cache/static/jquery-
1.10.2.min_f2fb5194.js"></script>
```

4 head

4.1 title

[强制] 页面必须包含 title 标签声明标题。

[强制] title 必须作为 head 的直接子元素，并紧随 charset 声明之后。

解释：

title 中如果包含 ascii
之外的字符，浏览器需要知道字符编码类型才能进行解码，否则可能导致乱码。

示例：

```
<head>
    <meta charset="UTF-8">
    <title>页面标题</title>
</head>
```

4.2 favicon

[强制] 保证 favicon 可访问。

解释：

在未指定 favicon 时，大多数浏览器会请求 Web Server 根目录下的 favicon.ico
。为了保证favicon可访问，避免404，必须遵循以下两种方法之一：

1. 在 Web Server 根目录放置 favicon.ico 文件。
2. 使用 link 指定 favicon。

示例：

```
<link rel="shortcut icon" href="path/to/favicon.ico">
```

4.3 viewport

[建议] 若页面欲对移动设备友好，需指定页面的 viewport。

解释：

viewport meta
tag可以设置可视区域的宽度和初始缩放大小，避免在移动设备上出现页面展示不正常。

比如，在页面宽度小于 980px 时，若需 iOS 设备友好，应当设置 viewport 的
width

值来适应你的页面宽度。同时因为不同移动设备分辨率不同，在设置时，应当使用 device-width 和 device-height 变量。

另外，为了使 viewport
正常工作，在页面内容样式布局设计上也要做相应调整，如避免绝对定位等。关于 viewport 的更多介绍，可以参见 Safari Web Content Guide的介绍

5 图片

[强制] 禁止 `img` 的 `src` 取值为空。延迟加载的图片也要增加默认的 `src`。

解释：

src
取值为空，会导致部分浏览器重新加载一次当前页面，参考：https://developer.yahoo.com/performance/rules.html#emptysrc

[建议] 避免为 `img` 添加不必要的 `title` 属性。

解释：

多余的 title 影响看图体验，并且增加了页面尺寸。

[建议] 为重要图片添加 `alt` 属性。

解释：

可以提高图片加载失败时的用户体验。

[建议] 添加 `width` 和 `height` 属性，以避免页面抖动。

[建议] 有下载需求的图片采用 `img` 标签实现，无下载需求的图片采用 `CSS` 背景图实现。

解释：

1. 产品 logo、用户头像、用户产生的图片等有潜在下载需求的图片，以 img 形式实现，能方便用户下载。
2. 无下载需求的图片，比如：icon、背景、代码使用的图片等，尽可能采用 css 背景图实现。

6 表单

6.1 控件标题

[强制] 有文本标题的控件必须使用 `label` 标签将其与其标题相关联。

解释：

有两种方式：

1. 将控件置于 label 内。
2. label 的 for 属性指向控件的 id。

推荐使用第一种，减少不必要的 id。如果 DOM 结构不允许直接嵌套，则应使用第二种。

示例：

```
<label><input type="checkbox" name="confirm" value="on">
我已确认上述条款</label>

<label for="username">用户名：</label> <input type="textbox"
name="username" id="username">
```

6.2 按钮

[强制] 使用 button 元素时必须指明 type 属性值。

解释：

button 元素的默认 type 为 submit，如果被置于 form 元素中，点击后将导致表单提交。为显示区分其作用方便理解，必须给出 type 属性。

示例：

```
<button type="submit">提交</button>
<button type="button">取消</button>
```

[建议] 尽量不要使用按钮类元素的 name 属性。

解释：

由于浏览器兼容性问题，使用按钮的 name 属性会带来许多难以发现的问题。具体情况可参考此文。

6.3 可访问性 (A11Y)

[建议] 负责主要功能的按钮在 DOM 中的顺序应靠前。

解释：

负责主要功能的按钮应相对靠前，以提高可访问性。如果在 CSS 中指定了 `float: right` 则可能导致视觉上主按钮在前，而 DOM 中主按钮靠后的情况。

示例：

```
<!-- good -->
<style>
.buttons .button-group {
    float: right;
}
</style>

<div class="buttons">
    <div class="button-group">
        <button type="submit">提交</button>
        <button type="button">取消</button>
    </div>
</div>

<!-- bad -->
<style>
.buttons button {
    float: right;
}
</style>

<div class="buttons">
    <button type="button">取消</button>
    <button type="submit">提交</button>
</div>
```

[建议] 当使用 JavaScript 进行表单提交时，如果条件允许，应使原生提交功能正常工作。

解释：

当浏览器 JS 运行错误或关闭 JS 时，提交功能将无法工作。如果正确指定了 form 元素的 action 属性和表单控件的 name 属性时，提交仍可继续进行。

示例：

```
<form action="/login" method="post">
    <p><input name="username" type="text" placeholder="用户名"></p>
    <p><input name="password" type="password" placeholder="密码"></p>
</form>
```

[建议] 在针对移动设备开发的页面时，根据内容类型指定输入框的 type 属性。

解释：

根据内容类型指定输入框类型，能获得能友好的输入体验。

示例：

```
<input type="date">
```

7 多媒体

[建议] 当在现代浏览器中使用 audio 以及 video 标签来播放音频、视频时，应当注意格式。

解释：

音频应尽可能覆盖到如下格式：

- MP3
- WAV
- Ogg

视频应尽可能覆盖到如下格式：

- MP4
- WebM
- Ogg

[建议] 在支持 HTML5 的浏览器中优先使用 audio 和 video 标签来定义音视频元素。

[建议] 使用退化到插件的方式来对多浏览器进行支持。

示例：

```
<audio controls>
    <source src="audio.mp3" type="audio/mpeg">
    <source src="audio.ogg" type="audio/ogg">
    <object width="100" height="50" data="audio.mp3">
        <embed width="100" height="50" src="audio.swf">
    </object>
</audio>

<video width="100" height="50" controls>
    <source src="video.mp4" type="video/mp4">
    <source src="video.ogg" type="video/ogg">
    <object width="100" height="50" data="video.mp4">
        <embed width="100" height="50" src="video.swf">
    </object>
</video>
```

[建议] 只在必要的时候开启音视频的自动播放。

[建议] 在 object 标签内部提供指示浏览器不支持该标签的说明。

示例：

```
<object width="100" height="50" data="something.swf">DO NOT SUPPORT
THIS TAG</object>
```

8 模板中的 HTML

[建议] 模板代码的缩进优先保证 HTML 代码的缩进规则。

示例：

```
<!-- good -->
{if $display == true}
<div>
    <ul>
    {foreach $item_list as $item}
        <li>{$item.name}<li>
    {/foreach}
    </ul>
</div>
{/if}

<!-- bad -->
{if $display == true}
    <div>
        <ul>
    {foreach $item_list as $item}
        <li>{$item.name}<li>
    {/foreach}
        </ul>
    </div>
{/if}
```

[建议] 模板代码应以保证 HTML 单个标签语法的正确性为基本原则。

示例：

```
<!-- good -->
<li class="{if $item.type_id == $current_type}focus{/if}">{
$item.type_name }</li>

<!-- bad -->
<li {if $item.type_id == $current_type} class="focus"{/if}>{
$item.type_name }</li>
```

[建议]
在循环处理模板数据构造表格时，若要求每行输出固定的个数，建议先将数据分组，之后再循环输出。

示例：

```
<!-- good -->
<table>
    {foreach $item_list as $item_group}
    <tr>
```

```
        {foreach $item_group as $item}
        <td>{ $item.name }</td>
        {/foreach}
    <tr>
    {/foreach}
</table>

<!-- bad -->
<table>
<tr>
    {foreach $item_list as $item}
    <td>{ $item.name }</td>
        {if $item@iteration is div by 5}
    </tr>
    <tr>
        {/if}
    {/foreach}
</tr>
</table>
```

CSS编码规范

1 前言

CSS作为网页样式的描述语言，在百度一直有着广泛的应用。本文档的目标是使CSS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CSS设计的，但是在使用各种CSS的预编译器(如less、sass、stylus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] CSS 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。

2.2 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。

示例：

```
.selector {
    margin: 0;
    padding: 0;
}
```

2.3 空格

[强制] 选择器 与 { 之间必须包含空格。

示例：

```
.selector {
}
```

[强制] 属性名 与之后的 : 之间不允许包含空格， : 与 属性值 之间必须包含空格。

示例：

```
margin: 0;
```

[强制] 列表型属性值 书写在单行时， , 后必须跟一个空格。

示例：

```
font-family: Arial, sans-serif;
```

2.4 行长度

[强制] 每行不得超过 120 个字符，除非单行不可分割。

解释：

常见不可分割的场景为URL超长。

[建议] 对于超长的样式，在样式值的 空格 处或 , 后换行，建议按逻辑分组。

示例：

```
/* 不同属性值按逻辑分组 */
background:
    transparent url(aVeryVeryVeryLongUrlIsPlacedHere)
    no-repeat 0 0;

/* 可重复多次的属性，每次重复一行 */
background-image:
    url(aVeryVeryVeryLongUrlIsPlacedHere)
    url(anotherVeryVeryVeryLongUrlIsPlacedHere);

/* 类似函数的属性值可以根据函数调用的缩进进行 */
background-image: -webkit-gradient(
    linear,
    left bottom,
    left top,
    color-stop(0.04, rgb(88,94,124)),
    color-stop(0.52, rgb(115,123,162))
);
```

2.5 选择器

[强制] 当一个 rule 包含多个 selector 时，每个选择器声明必须独占一行。

示例：

```
/* good */
.post,
.page,
.comment {
    line-height: 1.5;
}

/* bad */
.post, .page, .comment {
    line-height: 1.5;
}
```

[强制] >、+、~ 选择器的两边各保留一个空格。

示例：

```
/* good */
main > nav {
    padding: 10px;
}

label + input {
    margin-left: 5px;
}

input:checked ~ button {
    background-color: #69C;
}

/* bad */
main>nav {
    padding: 10px;
}

label+input {
    margin-left: 5px;
}

input:checked~button {
    background-color: #69C;
}
```

[强制] 属性选择器中的值必须用双引号包围。

解释：

不允许使用单引号，不允许不使用引号。

示例：

```
/* good */
article[character="juliet"] {
    voice-family: "Vivien Leigh", victoria, female
}

/* bad */
article[character='juliet'] {
    voice-family: "Vivien Leigh", victoria, female
}
```

2.6 属性

[强制] 属性定义必须另起一行。

示例：

```
/* good */
.selector {
    margin: 0;
    padding: 0;
}

/* bad */
.selector { margin: 0; padding: 0; }
```

[强制] 属性定义后必须以分号结尾。

示例：

```
/* good */
.selector {
    margin: 0;
}

/* bad */
.selector {
    margin: 0
}
```

3 通用

3.1 选择器

[强制] 如无必要，不得为 id、class 选择器添加类型选择器进行限定。

解释：

在性能和维护性上，都有一定的影响。

示例：

```
/* good */
#error,
.danger-message {
    font-color: #c00;
}

/* bad */
dialog#error,
p.danger-message {
    font-color: #c00;
}
```

[建议] 选择器的嵌套层级应不大于 3 级，位置靠后的限定条件应尽可能精确。

示例：

```
/* good */
#username input {}
.comment .avatar {}

/* bad */
.page .header .login #username input {}
.comment div * {}
```

3.2 属性缩写

[建议] 在可以使用缩写的情况下，尽量使用属性缩写。

示例：

```
/* good */
.post {
    font: 12px/1.5 arial, sans-serif;
}

/* bad */
.post {
    font-family: arial, sans-serif;
```

```
    font-size: 12px;
    line-height: 1.5;
}
```

[建议] 使用 border / margin / padding 等缩写时，应注意隐含值对实际数值的影响，确实需要设置多个方向的值时才使用缩写。

解释：

border / margin / padding 等缩写会同时设置多个属性的值，容易覆盖不需要覆盖的设定。如某些方向需要继承其他声明的值，则应该分开设置。

示例：

```
/* centering <article class="page"> horizontally and highlight featured ones */
article {
    margin: 5px;
    border: 1px solid #999;
}

/* good */
.page {
    margin-right: auto;
    margin-left: auto;
}

.featured {
    border-color: #69c;
}

/* bad */
.page {
    margin: 5px auto; /* introducing redundancy */
}

.featured {
    border: 1px solid #69c; /* introducing redundancy */
}
```

3.3 属性书写顺序

[建议] 同一 rule set 下的属性在书写时，应按功能进行分组，并以 Formatting Model（布局方式、位置） > Box Model（尺寸） > Typographic（文本相关） > Visual（视觉效果） 的顺序书写，以提高代码的可读性。

解释：

- Formatting Model 相关属性包括：`position` / `top` / `right` / `bottom` / `left` / `float` / `display` / `overflow` 等
- Box Model 相关属性包括：`border` / `margin` / `padding` / `width` / `height` 等
- Typographic 相关属性包括：`font` / `line-height` / `text-align` / `word-wrap` 等
- Visual 相关属性包括：`background` / `color` / `transition` / `list-style` 等

另外，如果包含 `content` 属性，应放在最前面。

示例：

```
.sidebar {
    /* formatting model: positioning schemes / offsets / z-indexes /
display / ...  */
    position: absolute;
    top: 50px;
    left: 0;
    overflow-x: hidden;

    /* box model: sizes / margins / paddings / borders / ...  */
    width: 200px;
    padding: 5px;
    border: 1px solid #ddd;

    /* typographic: font / aligns / text styles / ... */
    font-size: 14px;
    line-height: 20px;

    /* visual: colors / shadows / gradients / ... */
    background: #f5f5f5;
    color: #333;
    -webkit-transition: color 1s;
       -moz-transition: color 1s;
            transition: color 1s;
}
```

3.4 清除浮动

[建议] 当元素需要撑起高度以包含内部的浮动元素时，通过对伪类设置 `clear` 或触发 `BFC` 的方式进行 `clearfix`。尽量不使用增加空标签的方式。

解释：

触发 BFC 的方式很多，常见的有：

- float 非 none
- position 非 static
- overflow 非 visible

如希望使用更小副作用的清除浮动方法，参见 A new micro clearfix hack 一文。

另需注意，对已经触发 BFC 的元素不需要再进行 clearfix。

3.5 !important

[建议] 尽量不使用 `!important` 声明。

[建议] 当需要强制指定样式且不允许任何场景覆盖时，通过标签内联和 `!important` 定义样式。

解释：

必须注意的是，仅在设计上 确实不允许任何其它场景覆盖样式 时，才使用内联的 `!important` 样式。通常在第三方环境的应用中使用这种方案。下面的 z-index 章节是其中一个特殊场景的典型样例。

3.6 z-index

[建议] 将 `z-index` 进行分层，对文档流外绝对定位元素的视觉层级关系进行管理。

解释：

同层的多个元素，如多个由用户输入触发的 Dialog，在该层级内使用相同的 `z-index` 或递增 `z-index`。

建议每层包含100个 `z-index`
来容纳足够的元素，如果每层元素较多，可以调整这个数值。

[建议] 在可控环境下，期望显示在最上层的元素，`z-index` 指定为 `999999`。

解释：

可控环境分成两种，一种是自身产品线环境；还有一种是可能会被其他产品线引用，但是不会被外部第三方的产品引用。

不建议取值为
`2147483647`。以便于自身产品线被其他产品线引用时，当遇到层级覆盖冲突的情况，留出向上调整的空间。

[建议] 在第三方环境下，期望显示在最上层的元素，通过标签内联和 `!important`，将 `z-index` 指定为 `2147483647`。

解释：

第三方环境对于开发者来说完全不可控。在第三方环境下的元素，为了保证元素不被其页面其他样式定义覆盖，需要采用此做法。

4 值与单位

4.1 文本

[强制] 文本内容必须用双引号包围。

解释：

文本类型的内容可能在选择器、属性值等内容中。

示例：

```
/* good */
html[lang|="zh"] q:before {
    font-family: "Microsoft YaHei", sans-serif;
    content: "“";
}

html[lang|="zh"] q:after {
    font-family: "Microsoft YaHei", sans-serif;
    content: "”";
}

/* bad */
html[lang|=zh] q:before {
    font-family: 'Microsoft YaHei', sans-serif;
    content: '“';
}

html[lang|=zh] q:after {
    font-family: "Microsoft YaHei", sans-serif;
    content: "”";
}
```

4.2 数值

[强制] 当数值为 0 - 1 之间的小数时，省略整数部分的 0。

示例：

```
/* good */
panel {
    opacity: .8
}

/* bad */
panel {
    opacity: 0.8
}
```

4.3 url()

[强制] url() 函数中的路径不加引号。

示例：

```
body {
    background: url(bg.png);
}
```

[建议] url() 函数中的绝对路径可省去协议名。

示例：

```
body {
    background: url(//baidu.com/img/bg.png) no-repeat 0 0;
}
```

4.4 长度

[强制] 长度为 0 时须省略单位。 (也只有长度单位可省)

示例：

```
/* good */
body {
    padding: 0 5px;
}

/* bad */
body {
    padding: 0px 5px;
}
```

4.5 颜色

[强制] RGB颜色值必须使用十六进制记号形式 #rrggbb。不允许使用 rgb()。

解释：

带有alpha的颜色信息可以使用 rgba()。使用 rgba()
时每个逗号后必须保留一个空格。

示例：

```
/* good */
.success {
    box-shadow: 0 0 2px rgba(0, 128, 0, .3);
    border-color: #008000;
}
```

```
/* bad */
.success {
    box-shadow: 0 0 2px rgba(0,128,0,.3);
    border-color: rgb(0, 128, 0);
}
```

[强制] 颜色值可以缩写时，必须使用缩写形式。

示例：

```
/* good */
.success {
    background-color: #aca;
}

/* bad */
.success {
    background-color: #aaccaa;
}
```

[强制] 颜色值不允许使用命名色值。

示例：

```
/* good */
.success {
    color: #90ee90;
}

/* bad */
.success {
    color: lightgreen;
}
```

[建议]
颜色值中的英文字符采用小写。如不用小写也需要保证同一项目内保持大小写一致。

示例：

```
/* good */
.success {
    background-color: #aca;
    color: #90ee90;
}

/* good */
.success {
    background-color: #ACA;
    color: #90EE90;
```

```
}

/* bad */
.success {
    background-color: #ACA;
    color: #90ee90;
}
```

4.6 2D 位置

[强制] 必须同时给出水平和垂直方向的位置。

解释：

2D 位置初始值为 `0% 0%`，但在只有一个方向的值时，另一个方向的值会被解析为 center。为避免理解上的困扰，应同时给出两个方向的值。background-position属性值的定义

示例：

```
/* good */
body {
    background-position: center top; /* 50% 0% */
}

/* bad */
body {
    background-position: top; /* 50% 0% */
}
```

5 文本编排

5.1 字体族

[强制] font-family 属性中的字体族名称应使用字体的英文 Family Name，其中如有空格，须放置在引号中。

解释：

所谓英文 Family Name，为字体文件的一个元数据，常见名称如下：

字体	操作系统	Family Name
宋体 (中易宋体)	Windows	SimSun
黑体 (中易黑体)	Windows	SimHei
微软雅黑	Windows	Microsoft YaHei
微软正黑	Windows	Microsoft JhengHei

华文黑体	Mac/iOS	STHeiti
冬青黑体	Mac/iOS	Hiragino Sans GB
文泉驿正黑	Linux	WenQuanYi Zen Hei
文泉驿微米黑	Linux	WenQuanYi Micro Hei

示例：

```
h1 {
    font-family: "Microsoft YaHei";
}
```

[强制] font-family 按「西文字体在前、中文字体在后」、「效果佳(质量高/更能满足需求)的字体在前、效果一般的字体在后」的顺序编写，最后必须指定一个通用字体族(serif / sans-serif)。

解释：

更详细说明可参考本文。

示例：

```
/* Display according to platform */
.article {
    font-family: Arial, sans-serif;
}

/* Specific for most platforms */
h1 {
    font-family: "Helvetica Neue", Arial, "Hiragino Sans GB",
"WenQuanYi Micro Hei", "Microsoft YaHei", sans-serif;
}
```

[强制] font-family 不区分大小写，但在同一个项目中，同样的 Family Name 大小写必须统一。

示例：

```
/* good */
body {
    font-family: Arial, sans-serif;
}

h1 {
    font-family: Arial, "Microsoft YaHei", sans-serif;
}

/* bad */
```

```
body {
    font-family: arial, sans-serif;
}

h1 {
    font-family: Arial, "Microsoft YaHei", sans-serif;
}
```

5.2 字号

[强制] 需要在 Windows 平台显示的中文内容，其字号应不小于 12px。

解释：

由于 Windows 的字体渲染机制，小于 12px 的文字显示效果极差、难以辨认。

5.3 字体风格

[建议] 需要在 Windows 平台显示的中文内容，不要使用除 normal 外的 font-style。其他平台也应慎用。

解释：

由于中文字体没有 italic 风格的实现，所有浏览器下都会 fallback 到 obilique 实现 (自动拟合为斜体)，小字号下 (特别是 Windows
下会在小字号下使用点阵字体的情况下) 显示效果差，造成阅读困难。

5.4 字重

[强制] font-weight 属性必须使用数值方式描述。

解释：

CSS 的字重分 100 – 900
共九档，但目前受字体本身质量和浏览器的限制，实际上支持 400 和 700
两档，分别等价于关键词 normal 和 bold。

浏览器本身使用一系列启发式规则来进行匹配，在 <700 时一般匹配字体的 Regular 字重，>=700 时匹配 Bold 字重。

但已有浏览器开始支持 =600 时匹配 Semibold 字重
(见此表)，故使用数值描述增加了灵活性，也更简短。

示例：

```
/* good */
h1 {
    font-weight: 700;
}
```

```
/* bad */
h1 {
    font-weight: bold;
}
```

5.5 行高

[建议] line-height 在定义文本段落时，应使用数值。

解释：

将 line-height 设置为数值，浏览器会基于当前元素设置的 font-size 进行再次计算。在不同字号的文本段落组合中，能达到较为舒适的行间间隔效果，避免在每个设置了 font-size 都需要设置 line-height。

当 line-height 用于控制垂直居中时，还是应该设置成与容器高度一致。

示例：

```
.container {
    line-height: 1.5;
}
```

6 变换与动画

[强制] 使用 transition 时应指定 transition-property。

示例：

```
/* good */
.box {
    transition: color 1s, border-color 1s;
}

/* bad */
.box {
    transition: all 1s;
}
```

[建议] 尽可能在浏览器能高效实现的属性上添加过渡和动画。

解释：

见本文，在可能的情况下应选择这样四种变换：

- `transform: translate(npx, npx);`
- `transform: scale(n);`
- `transform: rotate(ndeg);`

- opacity: 0..1;

典型的，可以使用 translate 来代替 left 作为动画属性。

示例：

```
/* good */
.box {
    transition: transform 1s;
}
.box:hover {
    transform: translate(20px); /* move right for 20px */
}

/* bad */
.box {
    left: 0;
    transition: left 1s;
}
.box:hover {
    left: 20px; /* move right for 20px */
}
```

7 响应式

[强制] Media Query 不得单独编排，必须与相关的规则一起定义。

示例：

```
/* Good */
/* header styles */
@media (...) {
    /* header styles */
}

/* main styles */
@media (...) {
    /* main styles */
}

/* footer styles */
@media (...) {
    /* footer styles */
}


/* Bad */
/* header styles */
/* main styles */
/* footer styles */
```

```
@media (...) {
    /* header styles */
    /* main styles */
    /* footer styles */
}
```

[强制] Media Query
如果有多个逗号分隔的条件时，应将每个条件放在单独一行中。

示例：

```
@media
(-webkit-min-device-pixel-ratio: 2), /* Webkit-based browsers */
(min--moz-device-pixel-ratio: 2),    /* Older Firefox browsers (prior
to Firefox 16) */
(min-resolution: 2dppx),             /* The standard way */
(min-resolution: 192dpi) {           /* dppx fallback */
    /* Retina-specific stuff here */
}
```

[建议] 尽可能给出在高分辨率设备 (Retina) 下效果更佳的样式。

8 兼容性

8.1 属性前缀

[强制] 带私有前缀的属性由长到短排列，按冒号位置对齐。

解释：

标准属性放在最后，按冒号对齐方便阅读，也便于在编辑器内进行多行编辑。

示例：

```
.box {
    -webkit-box-sizing: border-box;
       -moz-box-sizing: border-box;
            box-sizing: border-box;
}
```

8.2 Hack

[建议] 需要添加 hack 时应尽可能考虑是否可以采用其他方式解决。

解释：

如果能通过合理的 HTML 结构或使用其他的 CSS
定义达到理想的样式，则不应该使用 hack 手段解决问题。通常 hack
会导致维护成本的增加。

[建议] 尽量使用 选择器 hack 处理兼容性，而非 属性 hack。

解释：

尽量使用符合 CSS 语法的 selector hack，可以避免一些第三方库无法识别 hack 语法的问题。

示例：

```
/* IE 7 */
*:first-child + html #header {
    margin-top: 3px;
    padding: 5px;
}

/* IE 6 */
* html #header {
    margin-top: 5px;
    padding: 4px;
}
```

[建议] 尽量使用简单的 属性 hack。

示例：

```
.box {
    _display: inline; /* fix double margin */
    float: left;
    margin-left: 20px;
}

.container {
    overflow: hidden;
    *zoom: 1; /* triggering hasLayout */
}
```

8.3 Expression

[强制] 禁止使用 Expression。

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

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。