# 前端编码规范

## 一般规范

以下章节列举了一些可应用在 HTML, JavaScript 和 CSS/SCSS 上的通用规则。

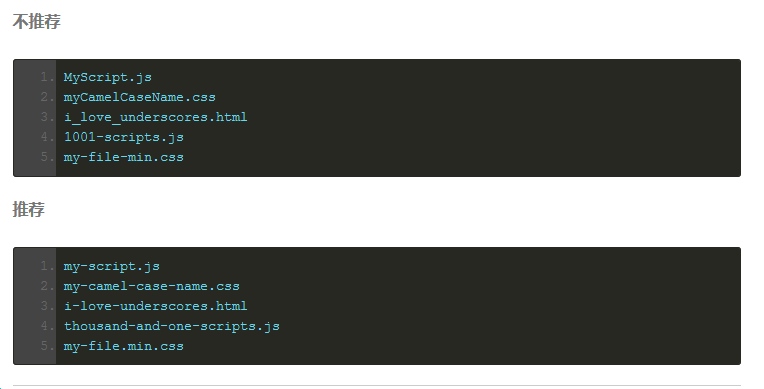
### 文件/资源命名`

在 web 项目中，所有的文件名应该都遵循同一命名约定。以可读性而言，减号（-）是用来分隔文件名的不二之选。同时它也是常见的 URL 分隔符（i.e. //example.com/blog/my-blog-entry or //s.example.com/images/big-black-background.jpg），所以理所当然的，减号应该也是用来分隔资源名称的好选择。

请确保文件命名总是以字母开头而不是数字。而以特殊字符开头命名的文件，一般都有特殊的含义与用处（比如 compass[1] 中的下划线就是用来标记跳过直接编译的文件用的）。

资源的字母名称必须全为小写，这是因为在某些对大小写字母敏感的操作系统中，当文件通过工具压缩混淆后，或者人为修改过后，大小写不同而导致引用文件不同的错误，很难被发现。

还有一些情况下，需要对文件增加前后缀或特定的扩展名（比如 .min.js, .min.css），抑或一串前缀（比如 3fa89b.main.min.css）。这种情况下，建议使用点分隔符来区分这些在文件名中带有清晰意义的元数据。



### 协议

不要指定引入资源所带的具体协议。

当引入图片或其他媒体文件，还有样式和脚本时，URLs 所指向的具体路径，不要指定协议部分（http:, https:），除非这两者协议都不可用。

不指定协议使得 URL 从绝对的获取路径转变为相对的，在请求资源协议无法确定时非常好用，而且还能为文件大小节省几个字节。



### 文本缩进

一次缩进2个空格或者一个tab。

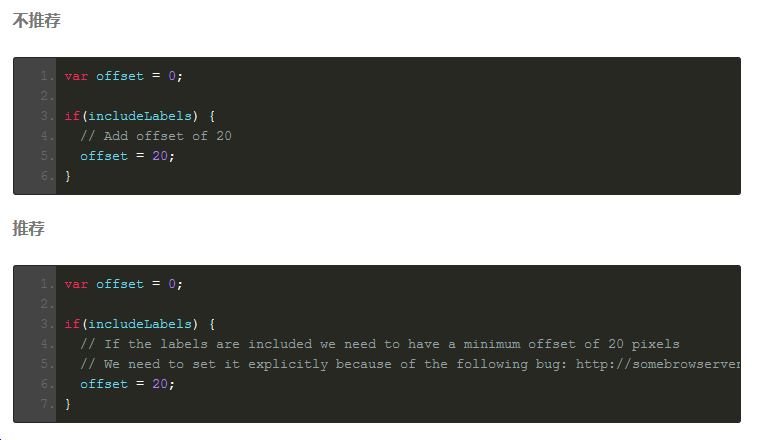


### 注释

注释是**你自己**与你的小伙伴们了解代码写法和目的的唯一途径。特别是在写一些看似琐碎的无关紧要的代码时，由于记忆点不深刻，注释就变得尤为重要了。

编写自解释代码只是一个**传说**，没有任何代码是可以完全自解释的。而代码注释，则是永远也不嫌多。

当你写注释时一定要注意：不要写你的代码都干了些什么，而要写你的代码为什么要这么写，背后的考量是什么。当然也可以加入所思考问题或是解决方案的链接地址。

一些注释工具可以帮助你写出更好的注释。[JSDoc](http://usejsdoc.org/) 或 [YUIDoc](http://yui.github.io/yuidoc/) 就是用来写 JavaScript 注释用的。你甚至可以使用工具来为这些注释生成文档，这也是激励开发者们写注释的一个好方法，因为一旦有了这样方便的生成文档的工具，他们通常会开始花更多时间在注释细节上。

### 代码检查

对于比较宽松自由的编程语言来说，严格遵循编码规范和格式化风格指南就显得极为重要。遵循规范固然很好，但是有自动化流程来确保其执行情况，岂不更佳。Trust is good, control is better.

对于 JavaScript，建议使用 [JSLint](http://www.jslint.com/) 或 [JSHint](http://www.jshint.com/)。

[1]: Compass 是一个基于 Sass 开源的 CSS 框架，而 Sass 是一个非常流行的 CSS 预编译器。

## HTML 规范

### 文档类型

推荐使用 HTML5 的文档类型申明： <!DOCTYPE html>.

（建议使用 text/html 格式的 HTML。避免使用 XHTML。XHTML 以及它的属性，比如 application/xhtml+xml 在浏览器中的应用支持与优化空间都十分有限）。

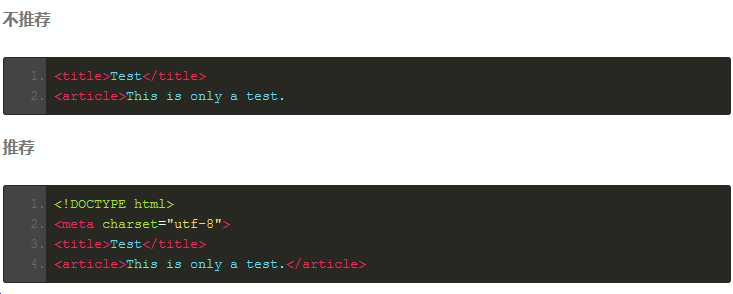
HTML 中最好不要将无内容元素[1] 的标签闭合，例如：使用 <br> 而非 <br />.

### HTML 验证

一般情况下，建议使用能通过标准规范验证的 HTML 代码，除非在性能优化和控制文件大小上不得不做出让步。

使用诸如 W3C HTML validator 这样的工具来进行检测。

规范化的 HTML 是显现技术要求与局限的显著质量基线，它促进了 HTML 被更好地运用。



### 省略可选标签

HTML5 规范中规定了 HTML 标签是可以省略的。但从可读性来说，在开发的源文件中最好不要这样做，因为省略标签可能会导致一些问题。

省略一些可选的标签确实使得页面大小减少，这很有用，尤其是对于一些大型网站来说。为了达到这一目的，我们可以在开发后期对页面进行压缩处理，在这个环节中这些可选的标签完全就可以省略掉了。

### 脚本加载

出于性能考虑，脚本异步加载很关键。一段脚本放置在 <head> 内，比如 <script src="main.js"></script>，其加载会一直阻塞 DOM 解析，直至它完全地加载和执行完毕。这会造成页面显示的延迟。特别是一些重量级的脚本，对用户体验来说那真是一个巨大的影响。

异步加载脚本可缓解这种性能影响。如果只需兼容 IE10+，可将 HTML5 的 async 属性加至脚本中，它可防止阻塞 DOM 的解析，甚至你可以将脚本引用写在 <head> 里也没有影响。

如需兼容老旧的浏览器，实践表明可使用用来动态注入脚本的脚本加载器。你可以考虑 [yepnope](http://yepnopejs.com/) 或 [labjs](http://labjs.com/)。注入脚本的一个问题是：[一直要等到 CSS 对象文档已就绪，它们才开始加载](https://www.igvita.com/2014/05/20/script-injected-async-scripts-considered-harmful/)（短暂地在 CSS 加载完毕之后），这就对需要及时触发的 JS 造成了一定的延迟，这多多少少也影响了用户体验吧。

终上所述，兼容老旧浏览器(IE9-)时，应该遵循以下最佳实践。

脚本引用写在 body 结束标签之前，并带上 async 属性。这虽然在老旧浏览器中不会异步加载脚本，但它只阻塞了 body 结束标签之前的 DOM 解析，这就大大降低了其阻塞影响。而在现代浏览器中，脚本将在 DOM 解析器发现 body 尾部的 script 标签才进行加载，此时加载属于异步加载，不会阻塞 CSSOM（但其执行仍发生在 CSSOM 之后）。



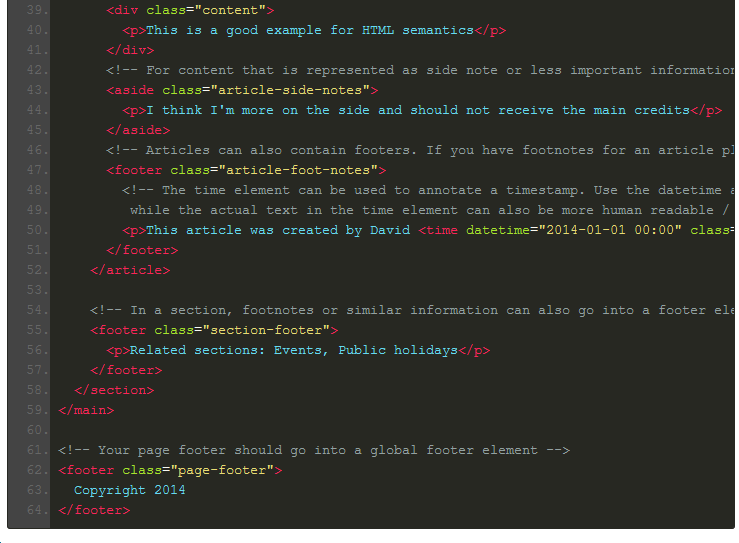
### 语义化

根据元素（有时被错误地称作“标签”）其被创造出来时的初始意义来使用它。打个比方，用 heading 元素来定义头部标题，p 元素来定义文字段落，用 a 元素来定义链接锚点，等等。

有根据有目的地使用 HTML 元素，对于可访问性、代码重用、代码效率来说意义重大。

以下示例列出了一些的语义化 HTML 主要情况：



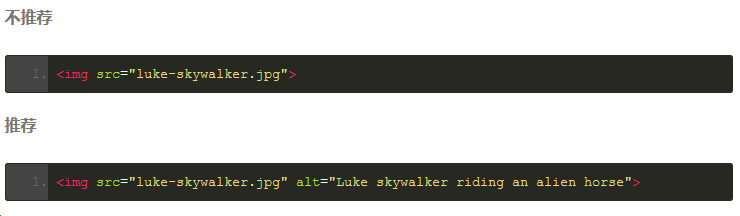


### 多媒体回溯

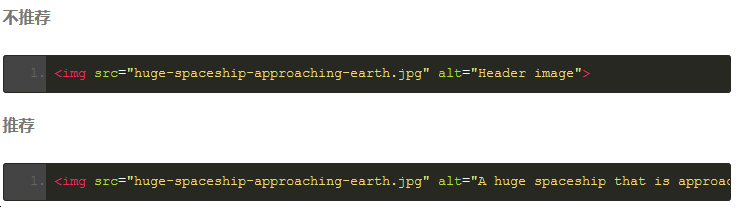
对页面上的媒体而言，像图片、视频、canvas 动画等，要确保其有可替代的接入接口。图片文件我们可采用有意义的备选文本（alt），视频和音频文件我们可以为其加上说明文字或字幕。

提供可替代内容对可用性来说十分重要。试想，一位盲人用户如何能知晓一张图片是什么，要是没有 @alt 的话。

（图片的 alt 属性是可不填写内容的，纯装饰性的图片就可用这么做：alt=""）。



尽量用 alt 标签去描述图片，设想你需要对于那些只能通过语音或者看不见图片的用户表达图片到底是什么。



### 关注点分离

理解 web 中如何和为何区分不同的关注点，这很重要。这里的关注点主要指的是：信息（HTML 结构）、外观（CSS）和行为（JavaScript）。为了使它们成为可维护的干净整洁的代码，我们要尽可能的将它们分离开来。

严格地保证结构、表现、行为三者分离，并尽量使三者之间没有太多的交互和联系。

就是说，尽量在文档和模板中只包含结构性的 HTML；而将所有表现代码，移入样式表中；将所有动作行为，移入脚本之中。

在此之外，为使得它们之间的联系尽可能的小，在文档和模板中也尽量少地引入样式和脚本文件。

清晰的分层意味着：

* 不使用超过一到两张样式表（i.e. main.css, vendor.css）
* 不使用超过一到两个脚本（学会用合并脚本）
* 不使用行内样式（<style>.no-good {}</style>）
* 不在元素上使用 style 属性（<hr style="border-top: 5px solid black">）
* 不使用行内脚本（<script>alert('no good')</script>）
* 不使用表象元素（i.e. <b>, <u>, <center>, <font>, <b>）
* 不使用表象 class 名（i.e. red, left, center）



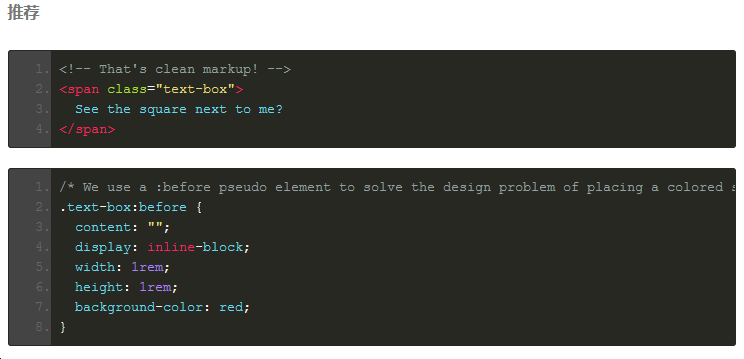
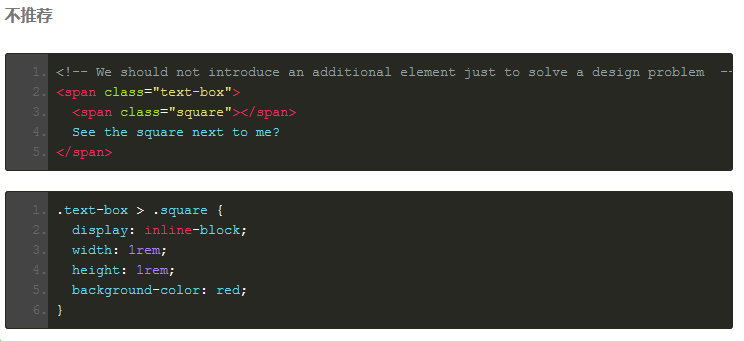
### HTML 内容至上

不要让非内容信息污染了你的 HTML。现在貌似有一种倾向：通过 HTML 来解决设计问题，这是显然是不对的。HTML 就应该只关注内容。

HTML 标签的目的，就是为了不断地展示内容信息。

* 不要引入一些特定的 HTML 结构来解决一些视觉设计问题
* 不要将 img 元素当做专门用来做视觉设计的元素

以下例子展示了误将 HTML 用来解决设计问题的这两种情况：



图片和 SVG 图形能被引入到 HTML 中的唯一理由是它们呈现出了与内容相关的一些信息。



### Type 属性

省略样式表与脚本上的 type 属性。鉴于 HTML5 中以上两者默认的 type 值就是 text/css 和 text/javascript，所以 type 属性一般是可以忽略掉的。甚至在老旧版本的浏览器中这么做也是安全可靠的。



### 可用性

如果 HTML5 语义化标签使用得当，许多可用性问题已经引刃而解。ARIA 规则在一些语义化的元素上可为其添上默认的可用性角色属性，使用得当的话已使网站的可用性大部分成立。假如你使用 nav, aside, main, footer 等元素，ARIA 规则会在其上应用一些关联的默认值。  
更多细节可参考 [ARIA specification](http://rawgit.com/w3c/aria-in-html/master/index.html#recommendations-table)

另外一些角色属性则能够用来呈现更多可用性情景（i.e. role="tab"）。

### Tab Index 在可用性上的运用

检查文档中的 tab 切换顺序并传值给元素上的 tabindex，这可以依据元素的重要性来重新排列其 tab 切换顺序。你可以设置 tabindex="-1" 在任何元素上来禁用其 tab 切换。

当你在一个默认不可聚焦的元素上增加了功能，你应该总是为其加上 tabindex 属性使其变为可聚焦状态，而且这也会激活其 CSS 的伪类 :focus。选择合适的 tabindex 值，或是直接使用 tabindex="0" 将元素们组织成同一 tab 顺序水平，并强制干预其自然阅读顺序。

### 微格式在 SEO 和可用性上的运用

如果 SEO 和可用性环境条件允许的话，建议考虑采用微格式。微格式是通过在元素标签上申明一系列特定数据来达成特定语义的方法。

谷歌、微软和雅虎对如何使用这些额外的数据一定程度上的达成一致，如果正确的使用，这将给搜索引擎优化带来巨大的好处。

你可以访问 [schema.org](http://schema.org/) 获得更多内容细节。

看一个电影网站的简单例子：



### ID 和锚点

通常一个比较好的做法是将页面内所有的头部标题元素都加上 ID. 这样做，页面 URL 的 hash 中带上对应的 ID 名称，即形成描点，方便跳转至对应元素所处位置。

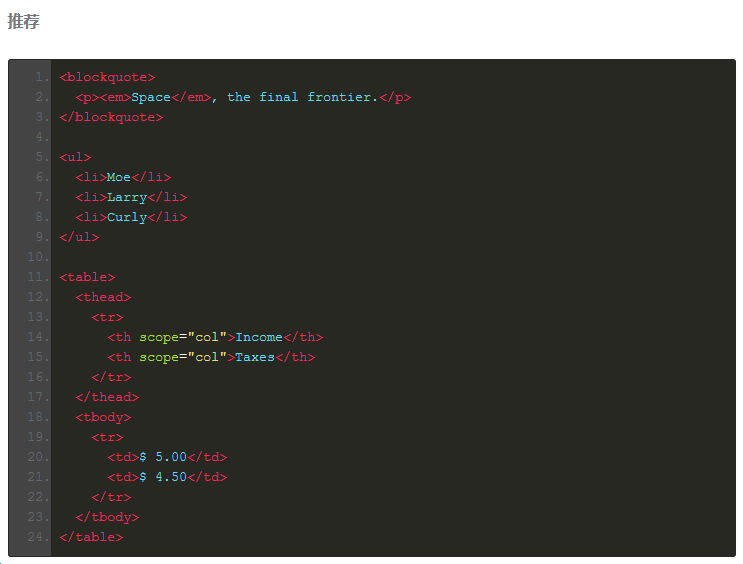
打个比方，当你在浏览器中输入 URL http://your-site.com/about#best-practices，浏览器将定位至以下 H3 上。

1. <h3 id="best-practices">Best practices</h3>

### 格式化规则

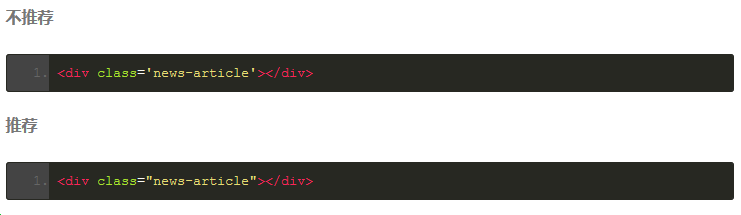
在每一个块状元素，列表元素和表格元素后，加上一新空白行，并对其子孙元素进行缩进。内联元素写在一行内，块状元素还有列表和表格要另起一行。

（如果由于换行的空格引发了不可预计的问题，那将所有元素并入一行也是可以接受的，格式警告总好过错误警告）。



### HTML 引号

使用双引号(“”) 而不是单引号(”) 。



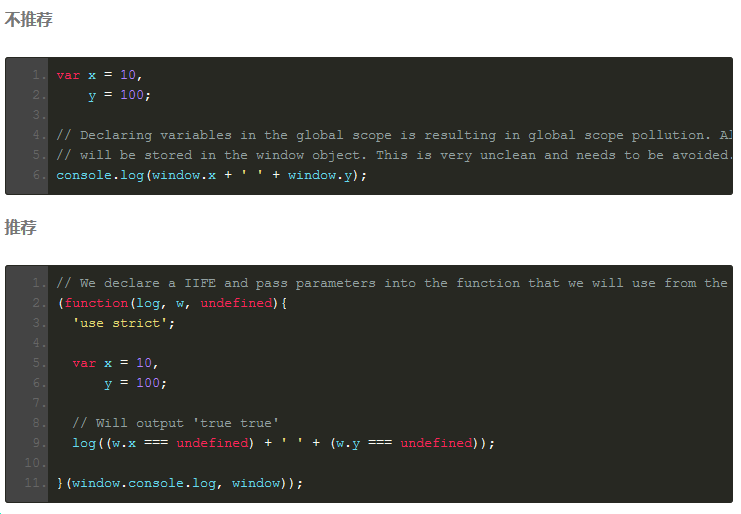
[1]: 此处的空白元素指的是以下元素：area, base, br, col, command, embed, hr, img, input, keygen, link, meta, param, source, track, wbr

## JavaScript 规范

### 全局命名空间污染与 IIFE

总是将代码包裹成一个 IIFE(Immediately-Invoked Function Expression)，用以创建独立隔绝的定义域。这一举措可防止全局命名空间被污染。

IIFE 还可确保你的代码不会轻易被其它全局命名空间里的代码所修改（i.e. 第三方库，window 引用，被覆盖的未定义的关键字等等）。



### IIFE（立即执行的函数表达式）

无论何时，想要创建一个新的封闭的定义域，那就用 IIFE。它不仅避免了干扰，也使得内存在执行完后立即释放。

所有脚本文件建议都从 IIFE 开始。

立即执行的函数表达式的执行括号应该写在外包括号内。虽然写在内还是写在外都是有效的，但写在内使得整个表达式看起来更像一个整体，因此推荐这么做。

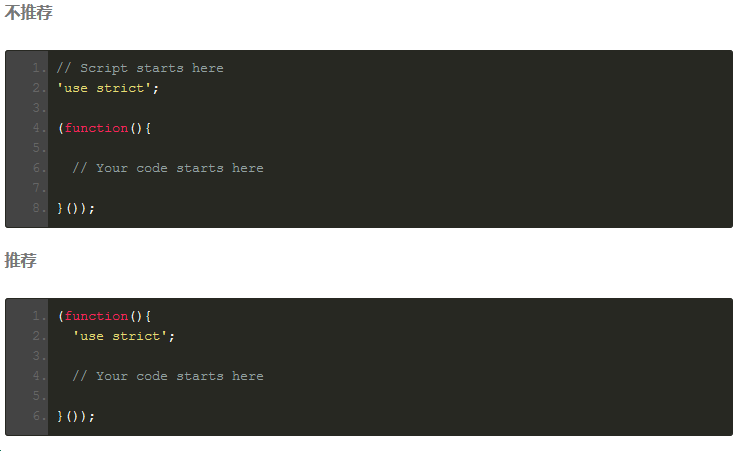
 

### 严格模式

ECMAScript 5 严格模式可在整个脚本或独个方法内被激活。它对应不同的 javascript 语境会做更加严格的错误检查。严格模式也确保了 javascript 代码更加的健壮，运行的也更加快速。

严格模式会阻止使用在未来很可能被引入的预留关键字。

你应该在你的脚本中启用严格模式，最好是在独立的 IIFE 中应用它。避免在你的脚本第一行使用它而导致你的所有脚本都启动了严格模式，这有可能会引发一些第三方类库的问题。



### 变量声明

总是使用 var 来声明变量。如不指定 var，变量将被隐式地声明为全局变量，这将对变量难以控制。如果没有声明，变量处于什么定义域就变得不清（可以是在 Document 或 Window 中，也可以很容易地进入本地定义域）。所以，请总是使用 var 来声明变量。

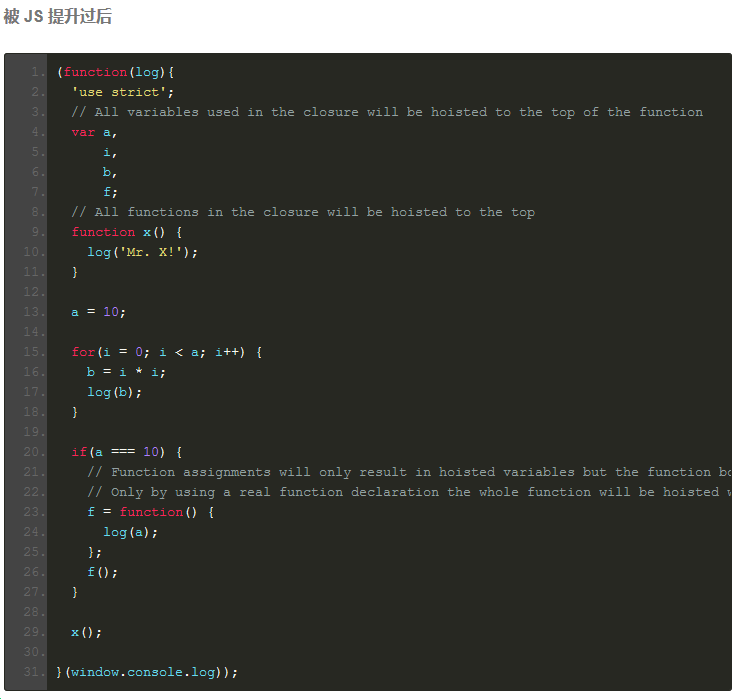
采用严格模式带来的好处是，当你手误输入错误的变量名时，它可以通过报错信息来帮助你定位错误出处。



### 理解 JavaScript 的定义域和定义域提升

在 JavaScript 中变量和方法定义会自动提升到执行之前。JavaScript 只有 function 级的定义域，而无其他很多编程语言中的块定义域，所以使得你在某一 function 内的某语句和循环体中定义了一个变量，此变量可作用于整个 function 内，而不仅仅是在此语句或循环体中，因为它们的声明被 JavaScript 自动提升了。

我们通过例子来看清楚这到底是怎么一回事：



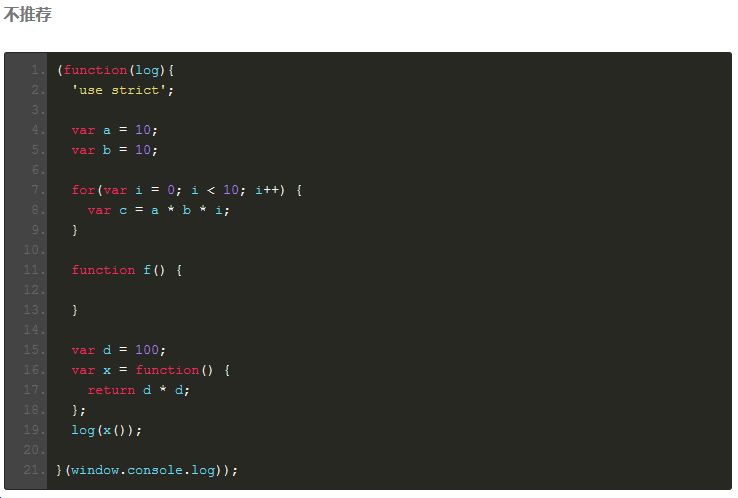
根据以上提升过程，你是否可理解以下代码？

正如你所看到的这段令人充满困惑与误解的代码导致了出人意料的结果。只有良好的声明习惯，也就是下一章节我们要提到的声明规则，才能尽可能的避免这类错误风险。

提升声明

为避免上一章节所述的变量和方法定义被自动提升造成误解，把风险降到最低，我们应该手动地显示地去声明变量与方法。也就是说，所有的变量以及方法，应当定义在 function 内的首行。

只用一个 var 关键字声明，多个变量用逗号隔开。





把赋值尽量写在变量申明中。



### 总是使用带类型判断的比较判断

总是使用 === 精确的比较操作符，避免在判断的过程中，由 JavaScript 的强制类型转换所造成的困扰。

如果你使用 === 操作符，那比较的双方必须是同一类型为前提的条件下才会有效。

如果你想了解更多关于强制类型转换的信息，你可以读一读 [Dmitry Soshnikov 的这篇文章](http://dmitrysoshnikov.com/notes/note-2-ecmascript-equality-operators/)。

在只使用 == 的情况下，JavaScript 所带来的强制类型转换使得判断结果跟踪变得复杂，下面的例子可以看出这样的结果有多怪了：



### 明智地使用真假判断

当我们在一个 if 条件语句中使用变量或表达式时，会做真假判断。if(a == true) 是不同于 if(a) 的。后者的判断比较特殊，我们称其为真假判断。这种判断会通过特殊的操作将其转换为 true 或 false，下列表达式统统返回 false：false, 0, undefined, null, NaN, ''（空字符串）.

这种真假判断在我们只求结果而不关心过程的情况下，非常的有帮助。

以下示例展示了真假判断是如何工作的：



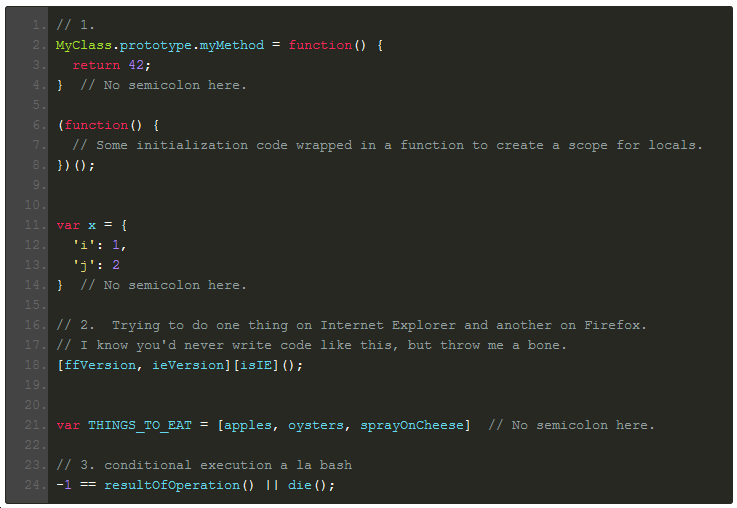
### 变量赋值时的逻辑操作

逻辑操作符 || 和 && 也可被用来返回布尔值。如果操作对象为非布尔对象，那每个表达式将会被自左向右地做真假判断。基于此操作，最终总有一个表达式被返回回来。这在变量赋值时，是可以用来简化你的代码的。



### 分号

总是使用分号，因为隐式的代码嵌套会引发难以察觉的问题。当然我们更要从根本上来杜绝这些问题[1] 。以下几个示例展示了缺少分号的危害：



**So what happens?**

1. JavaScript 错误 —— 首先返回 42 的那个 function 被第二个 function 当中参数传入调用，接着数字 42 也被“调用”而导致出错。
2. 八成你会得到 ‘no such property in undefined’ 的错误提示，因为在真实环境中的调用是这个样子：x[ffVersion, ieVersion][isIE]().
3. die 总是被调用。因为数组减 1 的结果是 NaN，它不等于任何东西（无论 resultOfOperation 是否返回 NaN）。所以最终的结果是 die() 执行完所获得值将赋给 THINGS\_TO\_EAT.

**Why?**

JavaScript 中语句要以分号结束，否则它将会继续执行下去，不管换不换行。以上的每一个示例中，函数声明或对象或数组，都变成了在一句语句体内。要知道闭合圆括号并不代表语句结束，JavaScript 不会终结语句，除非它的下一个 token 是一个中缀符[2] 或者是圆括号操作符。

这真是让人大吃一惊，所以乖乖地给语句末加上分号吧。

**澄清：分号与函数**

分号需要用在表达式的结尾，而并非函数声明的结尾。区分它们最好的例子是：

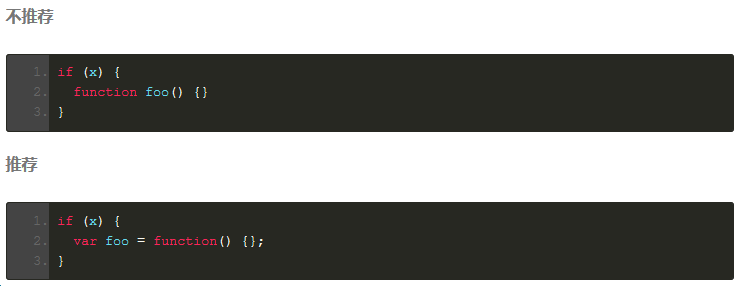


### 嵌套函数

嵌套函数是非常有用的，比如用在持续创建和隐藏辅助函数的任务中。你可以非常自由随意地使用它们。

### 语句块内的函数声明

切勿在语句块内声明函数，在 ECMAScript 5 的严格模式下，这是不合法的。函数声明应该在定义域的顶层。但在语句块内可将函数申明转化为函数表达式赋值给变量。

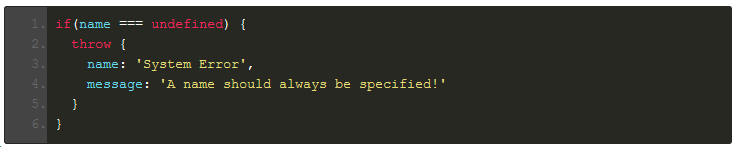


### 异常

基本上你无法避免出现异常，特别是在做大型开发时（使用应用开发框架等等）。

在没有自定义异常的情况下，从有返回值的函数中返回错误信息一定非常的棘手，更别提多不优雅了。不好的解决方案包括了传第一个引用类型来接纳错误信息，或总是返回一个对象列表，其中包含着可能的错误对象。以上方式基本上是比较简陋的异常处理方式。适时可做自定义异常处理。

在复杂的环境中，你可以考虑抛出对象而不仅仅是字符串（默认的抛出值）。



### 标准特性

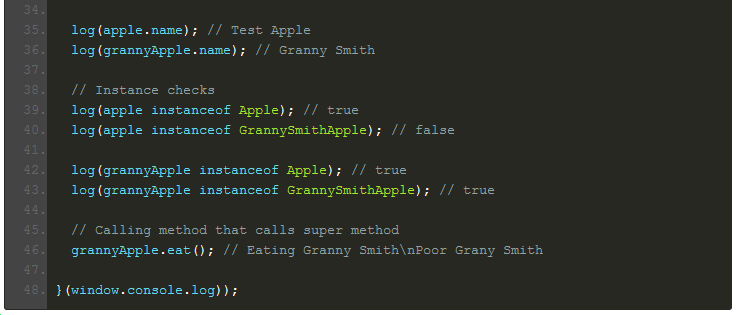
总是优先考虑使用标准特性。为了最大限度地保证扩展性与兼容性，总是首选标准的特性，而不是非标准的特性（例如：首选 string.charAt(3) 而不是 string[3]；首选 DOM 的操作方法来获得元素引用，而不是某一应用特定的快捷方法）。

### 简易的原型继承

如果你想在 JavaScript 中继承你的对象，请遵循一个简易的模式来创建此继承。如果你预计你会遇上复杂对象的继承，那可以考虑采用一个继承库，比如 [Proto.js by Axel Rauschmayer](https://github.com/rauschma/proto-js).

简易继承请用以下方式：



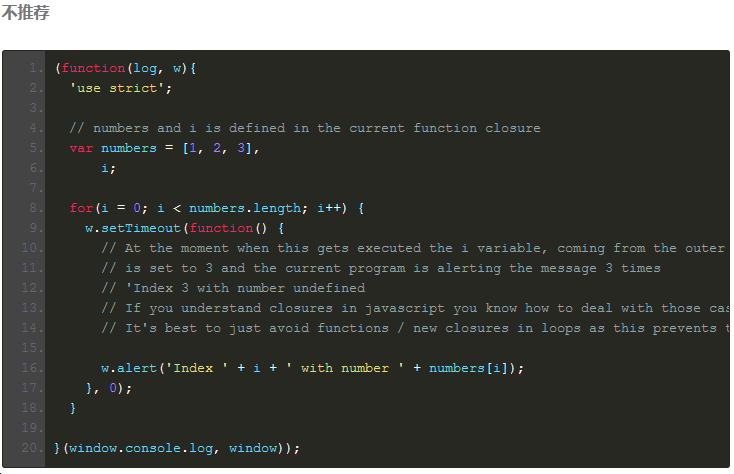


### 使用闭包

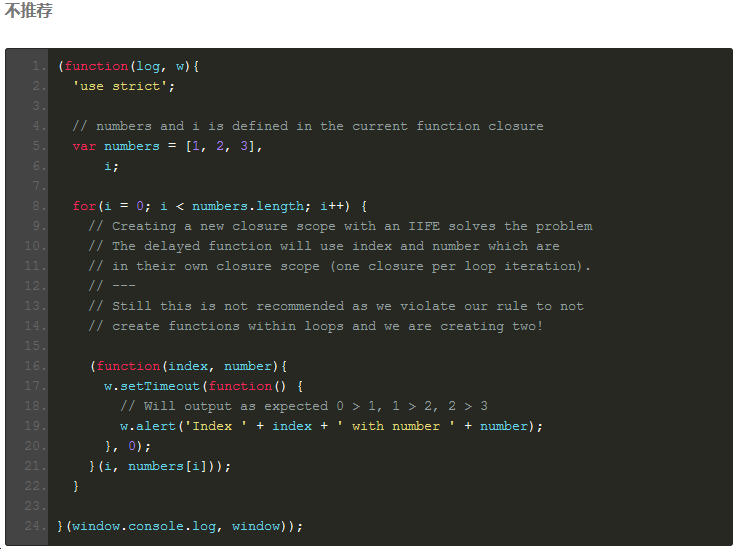
闭包的创建也许是 JS 最有用也是最易被忽略的能力了。[关于闭包如何工作的合理解释](http://jibbering.com/faq/faq_notes/closures.html)。

### 切勿在循环中创建函数

在简单的循环语句中加入函数是非常容易形成闭包而带来隐患的。下面的例子就是一个典型的陷阱：



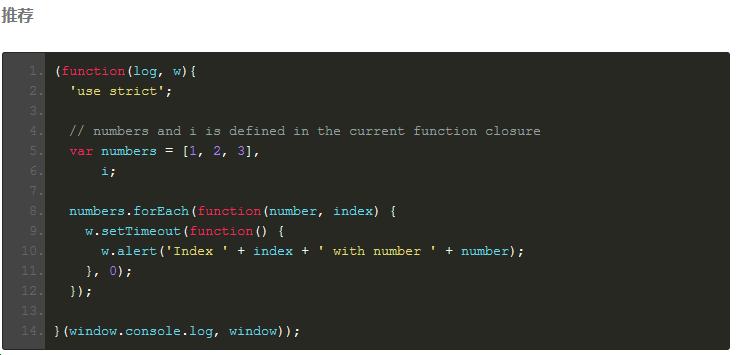
接下来的改进虽然已经解决了上述例子中的问题或 bug，但还是违反了不在循环中创建函数或闭包的原则。



接下来的改进已解决问题，而且也遵循了规范。可是，你会发现看上去似乎过于复杂繁冗了，应该会有更好的解决方案吧。



将循环语句转换为函数执行的方式问题能得到立马解决，每一次循环都会对应地创建一次闭包。函数式的风格更加值得推荐，而且看上去也更加地自然和可预料。



### eval 函数（魔鬼）

eval() 不但混淆语境还很危险，总会有比这更好、更清晰、更安全的另一种方案来写你的代码，因此尽量不要使用 evil 函数。

### this 关键字

只在对象构造器、方法和在设定的闭包中使用 this 关键字。this 的语义在此有些误导。它时而指向全局对象（大多数时），时而指向调用者的定义域（在 eval 中），时而指向 DOM 树中的某一节点（当用事件处理绑定到 HTML 属性上时），时而指向一个新创建的对象（在构造器中），还时而指向其它的一些对象（如果函数被 call() 和 apply() 执行和调用时）。

正因为它是如此容易地被搞错，请限制它的使用场景：

* 在构造函数中
* 在对象的方法中（包括由此创建出的闭包内）

### 首选函数式风格

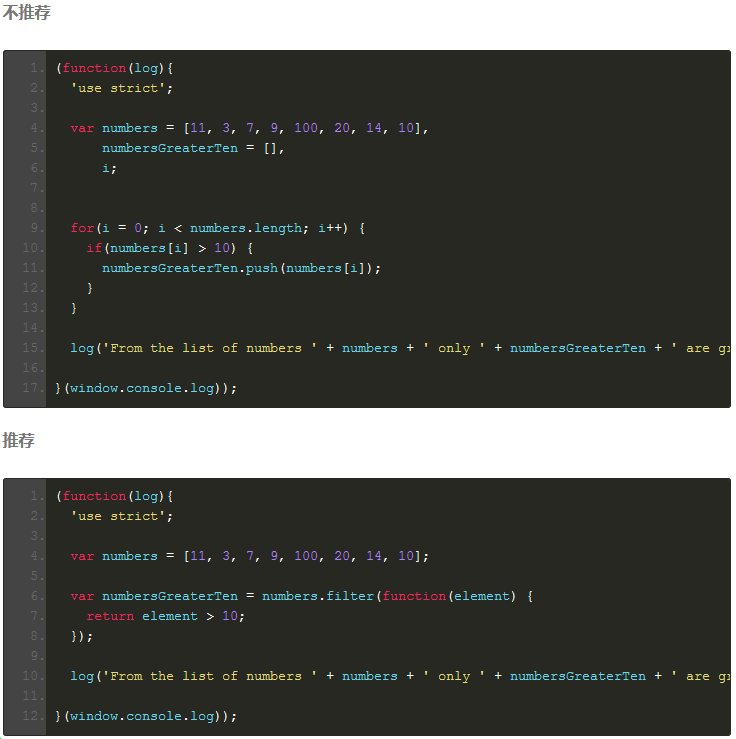
函数式编程让你可以简化代码并缩减维护成本，因为它容易复用，又适当地解耦和更少的依赖。

接下来的例子中，在一组数字求和的同一问题上，比较了两种解决方案。第一个例子是经典的程序处理，而第二个例子则是采用了函数式编程和 ECMA Script 5.1 的数组方法。

例外：往往在重代码性能轻代码维护的情况之下，要选择最优性能的解决方案而非维护性高的方案（比如用简单的循环语句代替 forEach）。



另一个例子通过某一规则对一个数组进行过滤匹配来创建一个新的数组。



### 使用 ECMA Script 5

建议使用 ECMA Script 5 中新增的语法糖和函数。这将简化你的程序，并让你的代码更加灵活和可复用。

### 数组和对象的属性迭代

用 ECMA5 的迭代方法来迭代数组。使用 Array.forEach 或者如果你要在特殊场合下中断迭代，那就用 Array.every。

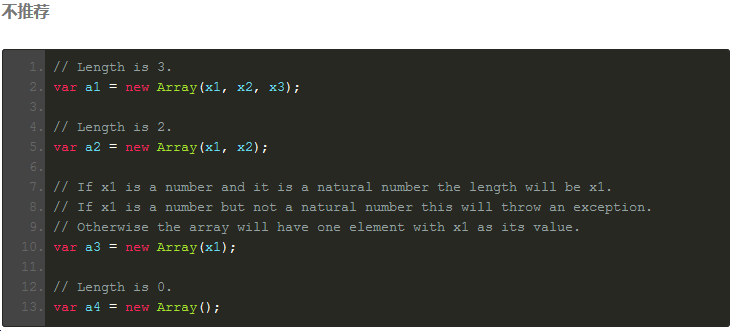


### 不要使用 switch

switch 在所有的编程语言中都是个非常错误的难以控制的语句，建议用 if else 来替换它。

### 数组和对象字面量

用数组和对象字面量来代替数组和对象构造器。数组构造器很容易让人在它的参数上犯错。

正因如此，如果将代码传参从两个变为一个，那数组很有可能发生意料不到的长度变化。为避免此类怪异状况，请总是采用更多可读的数组字面量。

对象构造器不会有类似的问题，但是为了可读性和统一性，我们应该使用对象字面量。



### 修改内建对象的原型链

修改内建的诸如 Object.prototype 和 Array.prototype 是被严厉禁止的。修改其它的内建对象比如 Function.prototype，虽危害没那么大，但始终还是会导致在开发过程中难以 debug 的问题，应当也要避免。

### 自定义 toString() 方法

你可以通过自定义 toString() 来控制对象字符串化。这很好，但你必须保证你的方法总是成功并不会有其它副作用。如果你的方法达不到这样的标准，那将会引发严重的问题。如果 toString() 调用了一个方法，这个方法做了一个断言[3] ，当断言失败，它可能会输出它所在对象的名称，当然对象也需要调用 toString()。

### 圆括号

一般在语法和语义上真正需要时才谨慎地使用圆括号。不要用在一元操作符上，例如 delete, typeof 和 void，或在关键字之后，例如 return, throw, case, new 等。

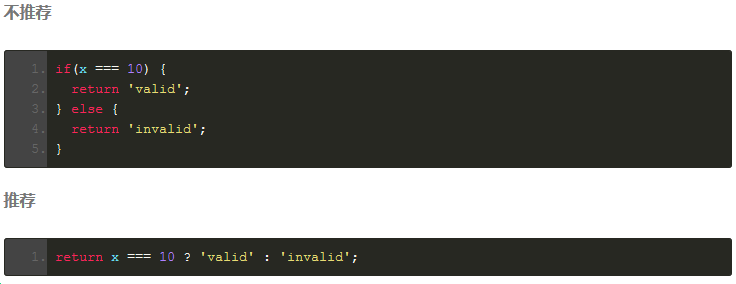
### 字符串

统一使用单引号(‘)，不使用双引号(“)。这在创建 HTML 字符串非常有好处：



### 三元条件判断（if 的快捷方法）

用三元操作符分配或返回语句。在比较简单的情况下使用，避免在复杂的情况下使用。没人愿意用 10 行三元操作符把自己的脑子绕晕。



[1]：作者指的是采用严格规范的语句写法，从根本上杜绝由分号缺失而引起的代码歧义。

[2]：中缀符，指的是像 x + y 中的 +。

[3]：断言一般指程序员在测试测序时的假设，一般是一些布尔表达式，当返回是 true 时，断言为真，代码运行会继续进行；如果条件判断为 false，代码运行停止，你的应用被终止。