深入理解 JavaScript 系列（1）：编写高质量 JavaScript 代码的基本要点

2011-12-28 23:00 by 汤姆大叔, 103745 阅读, 106 评论, 收藏, 编辑

才华横溢的 Stoyan Stefanov，在他写的由 O'Reilly 初版的新书《JavaScript Patterns》(JavaScript 模式)中，我想要是为我们的读者贡献其摘要，那会是件很美妙的事情。具体一点就是编写高质量 JavaScript 的一些要素，例如避免全局变量，使用单变量声明，在循环中预缓存 length(长度)，遵循代码阅读，以及更多。

此摘要也包括一些与代码不太相关的习惯，但对整体代码的创建息息相关，包括撰写 API 文档、执行同行评审以及运行 JSLint。这些习惯和最佳做法可以帮助你写出更好的，更易于理解和维护的代码，这些代码在几个月或是几年之后再回过头看看也是会觉得很自豪的。

书写可维护的代码(Writing Maintainable Code)

软件 bug 的修复是昂贵的，并且随着时间的推移，这些 bug 的成本也会增加，尤其当这些 bug 潜伏并慢慢出现在已经发布的软件中时。当你发现 bug 的时候就立即修复它是最好的，此时你代码要解决的问题在你脑中还是很清晰的。否则，你转移到其他任务，忘了那个特定的代码，一段时间后再去查看这些代码就 需要：

- 花时间学习和理解这个问题
- 化时间是了解应该解决的问题代码

还有问题，特别对于大的项目或是公司，修复 bug 的这位伙计不是写代码的那个人（且发现 bug 和修复 bug 的不是同一个人）。因此，必须降低理解代 码花费的时间，无论是一段时间前你自己写的代码还是团队中的其他成员写的代码。这关系到底线（营业收入）和开发人员的幸福，因为我们更应该去开发新的激动 人心的事物而不是花几小时几天的时间去维护遗留代码。

另一个相关软件开发生命的事实是，读代码花费的时间要比写来得多。有时候，当你专注并深入思考某个问题的时候，你可以坐下来，一个下午写大量的代码。

你的代码很能很快就工作了，但是，随着应用的成熟，还会有很多其他的事情发生，这就要求你的进行进行审查，修改，和调整。例如：

- bug 是暴露的
- 新功能被添加到应用程序
- 程序在新的环境下工作（例如，市场上出现新想浏览器）
- 代码改变用途
- 代码得完全从头重新，或移植到另一个架构上或者甚至使用另一种语言

由于这些变化，很少人力数小时写的代码最终演变成花数周来阅读这些代码。这就是为什么创建可维护的代码对应用程序的成功至关重要。

可维护的代码意味着：

- 可读的
- 一致的
- 可预测的
- 看上去就像是同一个人写的
- 已记录

最小全局变量(Minimizing Globals)

JavaScript 通过函数管理作用域。在函数内部声明的变量只在这个函数内部，函数外面不可用。另一方面，全局变量就是在任何函数外面声明的或是未声明直接简单使用的。

每个 JavaScript 环境有一个全局对象，当你在任意的函数外面使用 this 的时候可以访问到。你创建的每一个全部变量都成了这个全局对象的属 性。在浏览器中，方便起见，该全局对象有个附加属性叫做 window，此 window(通常)指向该全局对象本身。下面的代码片段显示了如何在浏览器环境 中创建和访问的全局变量：

```
myglobal = "hello"; // 不推荐写法
console.log(myglobal); // "hello"
console.log(window.myglobal); // "hello"
console.log(window["myglobal"]); // "hello"
console.log(this.myglobal); // "hello"
```

全局变量的问题

全局变量的问题在于，你的 JavaScript 应用程序和 web 页面上的所有代码都共享了这些全局变量，他们住在同一个全局命名空间，所以当程序的两个不同部分定义同名但不同作用的全局变量的时候，命名冲突在所难免。

web 页面包含不是该页面开发者所写的代码也是比较常见的，例如：

- 第三方的 JavaScript 库
- 广告方的脚本代码
- 第三方用户跟踪和分析脚本代码
- 不同类型的小组件，标志和按钮

比方说，该第三方脚本定义了一个全局变量，叫做 result；接着，在你的函数中也定义一个名为 result 的全局变量。其结果就是后面的变量覆盖前面的，第三方脚本就一下子嗝屁啦！

因此，要想和其他脚本成为好邻居的话，尽可能少的使用全局变量是很重要的。在书中后面提到的一些减少全局变量的策略，例如命名空间模式或是函数立即自动执行，但是要想让全局变量少最重要的还是始终使用 var 来声明变量。

由于 JavaScript 的两个特征，不自觉地创建出全局变量是出乎意料的容易。首先，你可以甚至不需要声明就可以使用变量；第二，JavaScript 有隐含的全局概念，意味着你不声明的任何变量都会成为一个全局对象属性。参考下面的代码：

```
function sum(x, y) {
   // 不推荐写法: 隐式全局变量
   result = x + y;
   return result;
}
```

此段代码中的 result 没有声明。代码照样运作正常，但在调用函数后你最后的结果就多一个全局命名空间，这可以是一个问题的根源。

经验法则是始终使用 var 声明变量，正如改进版的 sum()函数所演示的：

```
function sum(x, y) {
   var result = x + y;
   return result;
}
```

另一个创建隐式全局变量的反例就是使用任务链进行部分 var 声明。下面的片段中，a 是本地变量但是 b 确实全局变量，这可能不是你希望发生的：

```
// 反例，勿使用
function foo() {
   var a = b = 0;
   // ...
}
```

此现象发生的原因在于这个从右到左的赋值，首先，是赋值表达式 b = 0，此情况下 b 是未声明的。这个表达式的返回值是 0，然后这个 0 就分配给了通过 var 定义的这个局部变量 a。换句话说，就好比你输入了：

```
var a = (b = 0);
```

如果你已经准备好声明变量，使用链分配是比较好的做法，不会产生任何意料之外的全局变量，如：

```
function foo() {
   var a, b;
   // ... a = b = 0; // 两个均局部变量
}
```

然而，另外一个避免全局变量的原因是可移植性。如果你想你的代码在不同的环境下（主机下）运行，使用全局变量如履薄冰，因为你会无意中覆盖你最初环境下不存在的主机对象（所以你原以为名称可以放心大胆地使用，实际上对于有些情况并不适用）。

忘记 var 的副作用(Side Effects When Forgetting var)

隐式全局变量和明确定义的全局变量间有些小的差异，就是通过 delete 操作符让变量未定义的能力。

- 通过 var 创建的全局变量（任何函数之外的程序中创建）是不能被删除的。
- 无 var 创建的隐式全局变量（无视是否在函数中创建）是能被删除的。

这表明，在技术上，隐式全局变量并不是真正的全局变量，但它们是全局对象的属性。属性是可以通过 delete 操作符删除的，而变量是不能的：

```
// 定义三个全局变量
var global_var = 1;
global_novar = 2; // 反面教材
(function () {
   global_fromfunc = 3; // 反面教材
}());

// 试图删除
delete global_var; // false
delete global_novar; // true
delete global_fromfunc; // true

// 测试该删除
typeof global_var; // "number"
typeof global_novar; // "undefined"
typeof global_fromfunc; // "undefined"
```

在 ES5 严格模式下，未声明的变量（如在前面的代码片段中的两个反面教材）工作时会抛出一个错误。

访问全局对象(Access to the Global Object)

在浏览器中，全局对象可以通过 window 属性在代码的任何位置访问（除非你做了些比较出格的事情，像是声明了一个名为 window 的局部变量）。但是在其他环境下，这个方便的属性可能被叫做其他什么东西（甚至在程序中不可用）。如果你需要在没有硬编码的 window 标识符下访问全局对象，你可以在任何层级的函数作用域中做如下操作：

```
var global = (function () {
   return this;
}());
```

这种方法可以随时获得全局对象，因为其在函数中被当做函数调用了（不是通过 new 构造），this 总 是指向全局对象。实际上这个病不适用于 ECMAScript 5 严格模式，所以，在严格模式下时，你必须采取不同的形式。例如，你正在开发一个 JavaScript 库，你可以将你的代码包裹在一个即时函数中，然后从 全局作用域中，传递一个引用指向 this 作为你即时函数的参数。

单 var 形式（Single var Pattern）

在函数顶部使用单 var 语句是比较有用的一种形式，其好处在于：

- 提供了一个单一的地方去寻找功能所需要的所有局部变量
- 防止变量在定义之前使用的逻辑错误
- 帮助你记住声明的全局变量，因此较少了全局变量//zxx:此处我自己是有点晕乎的...
- 少代码（类型啊传值啊单线完成）

单 var 形式长得就像下面这个样子：

```
function func() {
    var a = 1,
        b = 2,
        sum = a + b,
        myobject = {},
        i,
        j;
    // function body...
}
```

您可以使用一个 var 语句声明多个变量，并以逗号分隔。像这种初始化变量同时初始化值的做法是很好的。这样子可以防止逻辑错误（所有未初始化但声明的变量的初始值是 undefined）和增加代码的可读性。在你看到代码后，你可以根据初始化的值知道这些变量大致的用途，例如是要当作对象呢还是当作整数来使。

你也可以在声明的时候做一些实际的工作，例如前面代码中的 sum = a + b 这个情况，另外一个例子就是当你使用 DOM（文档对象模型）引用时，你可以使用单一的 var 把 DOM 引用一起指定为局部变量，就如下面代码所示的：

```
function updateElement() {
    var el = document.getElementById("result"),
        style = el.style;
    // 使用 el 和 style 干点其他什么事...
}
```

预解析：var 散布的问题(Hoisting: A Problem with Scattered vars)

JavaScript 中，你可以在函数的任何位置声明多个 var 语句，并且它们就好像是在函数顶部声明一样发挥作用，这种行为称为 hoisting（悬置/置顶解析/预解析）。当你使用了一个变量，然后不久在函数中又重新声明的话，就可能产生逻辑错误。对于 JavaScript，只要你的变量是在同一个作用域中（同一函数），它都被当做是声明的，即使是它在 var 声明前使用的时候。看下面这个例子：

```
// 反例
myname = "global"; // 全局变量
function func() {
    alert(myname); // "undefined"
    var myname = "local";
    alert(myname); // "local"
}
func();
```

在这个例子中，你可能会以为第一个 alert 弹出的是"global"，第二个弹出"loacl"。这种期许是可以理解的，因为在第一个 alert 的时候，myname 未声明，此时函数肯定很自然而然地看全局变量 myname，但是，实际上并不是这么工作的。第一个 alert 会弹出"undefined"是因为 myname 被当做了函数的局部变量（尽管是之后声明的），所有的变量声明当被悬置到函数的顶部了。因此，为了避免这种混乱，最好是预先声明你想使用的全部变量。

var 置顶、预解析

上面的代码片段执行的行为可能就像下面这样：

```
myname = "global"; // global variable
function func() {
   var myname; // 等同于 -> var myname = undefined;
   alert(myname); // "undefined"
   myname = "local";
   alert(myname); // "local"}
func();
```

为了完整，我们再提一提执行层面的稍微复杂点的东西。代码处理分两个阶段，第一阶段是变量，函数声明，以及正常格式的参数创建，这是一个解析和进入上下文的阶段。第二个阶段是代码执行，函数表达式和不合格的标识符（为声明的变量）被创建。但是，出于实用的目的，我们就采用了"hoisting"这个概念，这种 ECMAScript 标准中并未定义，通常用来描述行为。

for 循环(for Loops)

在 for 循环中，你可以循环取得数组或是数组类似对象的值，譬如 arguments 和 HTMLCollection 对象。通常的循环形式如下：

```
// 次佳的循环
for (var i = 0; i < myarray.length; i++) {
   // 使用 myarray[i]做点什么
}
```

这种形式的循环的不足在于每次循环的时候数组的长度都要去获取下。这回降低你的代码，尤其当 myarray 不是数组，而是一个 HTMLCollection 对象的时候。

HTMLCollections 指的是 DOM 方法返回的对象，例如：

```
document.getElementsByName()
document.getElementsByClassName()
document.getElementsByTagName()
```

还有其他一些 HTMLCollections，这些是在 DOM 标准之前引进并且现在还在使用的。有：

```
document.images: 页面上所有的图片元素
document.links : 所有 a 标签元素
document.forms : 所有表单
document.forms[0].elements : 页面上第一个表单中的所有域
```

集合的麻烦在于它们实时查询基本文档（HTML 页面）。这意味着每次你访问任何集合的长度，你要实时查询 DOM，而 DOM 操作一般都是比较昂贵的。

这就是为什么当你循环获取值时，缓存数组(或集合)的长度是比较好的形式，正如下面代码显示的：

```
for (var i = 0, max = myarray.length; i < max; i++) {
   // 使用 myarray[i]做点什么
}
```

这样，在这个循环过程中，你只检索了一次长度值。

在所有浏览器下，循环获取内容时缓存 HTMLCollections 的长度是更快的，2 倍(Safari 3)到 190 倍(IE7)之间。//zxx:此数据貌似很老，仅供参考

注意到，当你明确想要修改循环中的集合的时候（例如，添加更多的 DOM 元素），你可能更喜欢长度更新而不是常量。

伴随着单 var 形式，你可以把变量从循环中提出来，就像下面这样：

```
function looper() {
    var i = 0,
        max,
        myarray = [];
    // ...
    for (i = 0, max = myarray.length; i < max; i++) {
        // 使用 myarray[i]做点什么
    }
}
```

这种形式具有一致性的好处，因为你坚持了单一 var 形式。不足在于当重构代码的时候，复制和粘贴整个循环有点困难。例如，你从一个函数复制了一个循环到另一个函数，你不得不去确定你能够把 i 和 max 引入新的函数（如果在这里没有用的话，很有可能你要从原函数中把它们删掉）。

最后一个需要对循环进行调整的是使用下面表达式之一来替换 i++。

```
i = i + 1
i += 1
```

JSLint 提示您这样做，原因是++和 - -促进了“过分棘手(excessive trickiness)”。//zxx:这里比较难翻译，我想本意应该是让代码变得更加的棘手
如果你直接无视它，JSLint 的 plusplus 选项会是 false（默认是 default）。

还有两种变化的形式，其又有了些微改进，因为：

使用for循环两个改进点：提高性能

- 少了一个变量(无 max)
- 向下数到 0，通常更快，因为和 0 做比较要比和数组长度或是其他不是 0 的东西作比较更有效率

```
//第一种变化的形式：

var i, myarray = [];
for (i = myarray.length; i--;) {
    // 使用 myarray[i]做点什么
}

//第二种使用 while 循环：

var myarray = [],
    i = myarray.length;
while (i--) {
```

```
    // 使用 myarray[i]做点什么
}
```

这些小的改进只体现在性能上，此外 JSLint 会对使用 i--加以抱怨。

for-in 循环(for-in Loops)

for-in 循环应该用在非数组对象的遍历上，使用 for-in 进行循环也被称为“枚举”。

从技术上将，你可以使用 for-in 循环数组（因为 JavaScript 中数组也是对象），但这是不推荐的。因为如果数组对象已被自定义的功能增强，就可能发生逻辑错误。另外，在 for-in 中，属性列表的顺序（序列）是不能保证的。所以最好数组使用正常的 for 循环，对象使用 for-in 循环。

有个很重要的 hasOwnProperty()方法，当遍历对象属性的时候可以过滤掉从原型链上下来的属性。

思考下面一段代码：

```
// 对象
var man = {
    hands: 2,
    legs: 2,
    heads: 1
};

// 在代码的某个地方
// 一个方法添加给了所有对象
if (typeof Object.prototype.clone === "undefined") {
    Object.prototype.clone = function () {};
}
```

在这个例子中，我们有一个使用对象字面量定义的名叫 man 的对象。在 man 定义完成后的某个地方，在对象原型上增加了一个很有用的名叫 clone()的方法。此原型链是实时的，这就意味着所有的对象自动可以访问新的方法。为了避免枚举 man 的时候出现 clone()方法，你需要应用 hasOwnProperty()方法过滤原型属性。如果不做过滤，会导致 clone()函数显示出来，在大多数情况下这是不希望出现的。

```
// 1.
// for-in 循环
for (var i in man) {
    if (man.hasOwnProperty(i)) { // 过滤
        console.log(i, ":", man[i]);
    }
}
/* 控制台显示结果
```

```
hands : 2
legs : 2
heads : 1
*/
// 2.
// 反面例子:
// for-in loop without checking hasOwnProperty()
for (var i in man) {
   console.log(i, ":", man[i]);
}
/*
控制台显示结果
hands : 2
legs : 2
heads : 1
clone: function()
*/
```

另外一种使用 hasOwnProperty()的形式是取消 Object.prototype 上的方法。像是：

```
for (var i in man) {
   if (Object.prototype.hasOwnProperty.call(man, i)) { // 过滤
      console.log(i, ":", man[i]);
   }
}
```

其好处在于在 man 对象重新定义 hasOwnProperty 情况下避免命名冲突。也避免了长属性查找对象的所有方法，你可以使用局部变量“缓存”它。

```
var i, hasOwn = Object.prototype.hasOwnProperty;
for (i in man) {
    if (hasOwn.call(man, i)) { // 过滤                    最佳实践
        console.log(i, ":", man[i]);
    }
}
```

严格来说，不使用 hasOwnProperty()并不是一个错误。根据任务以及你对代码的自信程度，你可以跳过它以提高些许的循环速度。但是当你对当前对象内容（和其原型链）不确定的时候，添加 hasOwnProperty()更加保险些。

格式化的变化（通不过 JSLint）会直接忽略掉花括号，把 if 语句放到同一行上。其优点在于循环语句读起来就像一个完整的想法（每个元素都有一个自己的属性”X”，使用”X”干点什么）：

```
// 警告： 通不过 JSLint 检测
var i, hasOwn = Object.prototype.hasOwnProperty;
for (i in man) if (hasOwn.call(man, i)) { // 过滤
    console.log(i, ":", man[i]);
}
```

（不）扩展内置原型((Not) Augmenting Built-in Prototypes)

扩增构造函数的 prototype 属性是个很强大的增加功能的方法，但有时候它太强大了。

增加内置的构造函数原型（如 Object(), Array(), 或 Function()）挺诱人的，但是这严重降低了可维护性，因为它让你的代码变得难以预测。使用你代码的其他开发人员很可能更期望使用内置的 JavaScript 方法来持续不断地工作，而不是你另加的方法。

另外，属性添加到原型中，可能会导致不使用 hasOwnProperty 属性时在循环中显示出来，这会造成混乱。

因此，不增加内置原型是最好的。你可以指定一个规则，仅当下面的条件均满足时例外：

- 可以预期将来的 ECMAScript 版本或是 JavaScript 实现将一直将此功能当作内置方法来实现。例如，你可以添加 ECMAScript 5 中描述的方法，一直到各个浏览器都迎头赶上。这种情况下，你只是提前定义了有用的方法。
- 如果您检查您的自定义属性或方法已不存在——也许已经在代码的其他地方实现或已经是你支持的浏览器 JavaScript 引擎部分。
- 你清楚地文档记录并和团队交流了变化。

如果这三个条件得到满足，你可以给原型进行自定义的添加，形式如下：

```
if (typeof Object.protoype.myMethod !== "function") {
   Object.protoype.myMethod = function () {
      // 实现...
   };
}
```

switch 模式(switch Pattern)

你可以通过类似下面形式的 switch 语句增强可读性和健壮性：

```
var inspect_me = 0,
    result = '';
```

```
switch (inspect_me) {
case 0:
   result = "zero";
   break;
case 1:
   result = "one";
   break;
default:
   result = "unknown";
}
```

这个简单的例子中所遵循的风格约定如下：

- 每个 case 和 switch 对齐（花括号缩进规则除外）
- 每个 case 中代码缩进
- 每个 case 以 break 清除结束
- 避免贯穿（故意忽略 break）。如果你非常确信贯穿是最好的方法，务必记录此情况，因为对于有些阅读人而言，它们可能看起来是错误的。
- 以 default 结束 switch：确保总有健全的结果，即使无情况匹配。

避免隐式类型转换(Avoiding Implied Typecasting)

JavaScript 的变量在比较的时候会隐式类型转换。这就是为什么一些诸如：false == 0 或 “” == 0 返回的结果是 true。为避免引起混乱的隐含类型转换，在你比较值和表达式类型的时候始终使用===和!==操作符。

```
var zero = 0;
if (zero === false) {
   // 不执行，因为 zero 为 0, 而不是 false
}

// 反面示例
if (zero == false) {
   // 执行了...
}
```

还有另外一种思想观点认为==就足够了===是多余的。例如，当你使用 typeof 你就知道它会返回一个字符串，所以没有使用严格相等的理由。然而，JSLint 要求严格相等，它使代码看上去更有一致性，可以降低代码阅读时的精力消耗。（“==是故意的还是一个疏漏？”）

避免(Avoiding) eval()

如果你现在的代码中使用了 eval()，记住该咒语“eval()是魔鬼”。此方法接受任意的字符串，并当作 JavaScript 代码来处理。当有 问题的代码是事先知道的（不是运行时确定的），没有理由使用 eval()。如果代码是在运行时动态生成，有一个更好的方式不使用 eval 而达到同样的目 标。例如，用方括号表示法来访问动态属性会更好更简单：

```
// 反面示例
var property = "name";
alert(eval("obj." + property));

// 更好的
var property = "name";
alert(obj[property]);
```

使用 eval()也带来了安全隐患，因为被执行的代码（例如从网络来）可能已被篡改。这是个很常见的反面教材，当处理 Ajax 请求得到的 JSON 相应的时候。在这些情况下，最好使用 JavaScript 内置方法来解析 JSON 相应，以确保安全和有效。若浏览器不支持 JSON.parse()，你可 以使用来自 JSON.org 的库。

eval替换方法以及setInterval等时间函数应避免的错误

同样重要的是要记住，给 setInterval()、setTimeout()和 Function()构造函数传递字符串，大部分情况下，与使用 eval()是类似的，因此要避免。在幕后，JavaScript 仍需要评估和执行你给程序传递的字符串：

```
// 反面示例
setTimeout("myFunc()", 1000);
setTimeout("myFunc(1, 2, 3)", 1000);

// 更好的
setTimeout(myFunc, 1000);
setTimeout(function () {
    myFunc(1, 2, 3);
}, 1000);
```

使用新的 Function()构造就类似于 eval()，应小心接近。这可能是一个强大的构造，但往往被误用。如果你绝对必须使用 eval()，你 可以考虑使用 new Function()代替。有一个小的潜在好处，因为在新 Function()中作代码评估是在局部函数作用域中运行，所以代码中任何被评估的通过 var 定义的变量都不会自动变成全局变量。另一种方法来阻止自动全局变量是封装 eval()调用到一个即时函数中。

考虑下面这个例子，这里仅 un 作为全局变量污染了命名空间。

```
console.log(typeof un);    // "undefined"
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"

var jsstring = "var un = 1; console.log(un);";
```

```
eval(jsstring); // logs "1"

jsstring = "var deux = 2; console.log(deux);";
new Function(jsstring)(); // logs "2"

jsstring = "var trois = 3; console.log(trois);";
(function () {
   eval(jsstring);
}()); // logs "3"

console.log(typeof un); // number
console.log(typeof deux); // "undefined"
console.log(typeof trois); // "undefined"
```

另一间 eval()和 Function 构造不同的是 eval()可以干扰作用域链，而 Function()更安分守己些。不管你在哪里执行 Function()，它只看到全局作用域。所以其能很好的避免本地变量污染。在下面这个例子中，eval()可以访问和修改它外部作用域中的变量，这是 Function 做不来的（注意到使用 Function 和 new Function 是相同的）。

eval与Function的优缺点

```
(function () {
   var local = 1;
   eval("local = 3; console.log(local)"); // logs "3"
   console.log(local); // logs "3"
}());

(function () {
   var local = 1;
   Function("console.log(typeof local);")(); // logs undefined
}());
```

parseInt()下的数值转换(Number Conversions with parseInt())

使用 parseInt()你可以从字符串中获取数值，该方法接受另一个基数参数，这经常省略，但不应该。当字符串以"0"开头的时候就有可能会出问 题，例如，部分时间进入表单域，在 ECMAScript 3 中，开头为"0"的字符串被当做 8 进制处理了，但这已在 ECMAScript 5 中改变了。为了避免矛盾和意外的结果，总是指定基数参数。

```
var month = "06",
    year = "09";
```

```
month = parseInt(month, 10);
year = parseInt(year, 10);
```

此例中，如果你忽略了基数参数，如 parseInt(year)，返回的值将是 0，因为“09”被当做 8 进制（好比执行 parseInt(year, 8)），而 09 在 8 进制中不是个有效数字。

替换方法是将字符串转换成数字，包括：

```
+"08" // 结果是 8
Number("08") // 8
```

这些通常快于 parseInt()，因为 parseInt()方法，顾名思意，不是简单地解析与转换。但是，如果你想输入例如“08 hello”，parseInt()将返回数字，而其它以 NaN 告终。

编码规范(Coding Conventions)

建立和遵循编码规范是很重要的，这让你的代码保持一致性，可预测，更易于阅读和理解。一个新的开发者加入这个团队可以通读规范，理解其它团队成员书写的代码，更快上手干活。

许多激烈的争论发生会议上或是邮件列表上，问题往往针对某些代码规范的特定方面（例如代码缩进，是 Tab 制表符键还是 space 空格键）。如果你是 你组织中建议采用规范的，准备好面对各种反对的或是听起来不同但很强烈的观点。要记住，建立和坚定不移地遵循规范要比纠结于规范的细节重要的多。

缩进(Indentation)

代码没有缩进基本上就不能读了。唯一糟糕的事情就是不一致的缩进，因为它看上去像是遵循了规范，但是可能一路上伴随着混乱和惊奇。重要的是规范地使用缩进。

一些开发人员更喜欢用 tab 制表符缩进，因为任何人都可以调整他们的编辑器以自己喜欢的空格数来显示 Tab。有些人喜欢空格——通常四个，这都无所谓，只要团队每个人都遵循同一个规范就好了。这本书，例如，使用四个空格缩进，这也是 JSLint 中默认的缩进。

什么应该缩进呢？规则很简单——花括号里面的东西。这就意味着函数体，循环 (do, while, for, for-in)，if，switch，以及对象字面量中的对象属性。下面的代码就是使用缩进的示例：

```
function outer(a, b) {
    var c = 1,
        d = 2,
        inner;
    if (a > b) {
        inner = function () {
```

```
            return {
                r: c - d
            };
        };
    } else {
        inner = function () {
            return {
                r: c + d
            };
        };
    }
    return inner;
}
```

花括号{}(Curly Braces)

花括号（亦称大括号，下同）应总被使用，即使在它们为可选的时候。技术上将，在 in 或是 for 中如果语句仅一条，花括号是不需要的，但是你还是应该总是使用它们，这会让代码更有持续性和易于更新。

想象下你有一个只有一条语句的 for 循环，你可以忽略花括号，而没有解析的错误。

```
// 糟糕的实例
for (var i = 0; i < 10; i += 1)
   alert(i);
```

但是，如果，后来，主体循环部分又增加了行代码？

```
// 糟糕的实例
for (var i = 0; i < 10; i += 1)
   alert(i);
   alert(i + " is " + (i % 2 ? "odd" : "even"));
```

第二个 alert 已经在循环之外，缩进可能欺骗了你。为了长远打算，最好总是使用花括号，即时值一行代码：

```
// 好的实例
for (var i = 0; i < 10; i += 1) {
   alert(i);
}
```

if 条件类似：

```
// 坏
if (true)
   alert(1);
else
   alert(2);

// 好
if (true) {
   alert(1);
} else {
   alert(2);
}
```

左花括号的位置(Opening Brace Location)

开发人员对于左大括号的位置有着不同的偏好——在同一行或是下一行。

```
if (true) {
   alert("It's TRUE!");
}

//或

if (true)
{
   alert("It's TRUE!");
}
```

这个实例中，仁者见仁智者见智，但也有个案，括号位置不同会有不同的行为表现。这是因为分号插入机制(semicolon insertion mechanism)——JavaScript 是不挑剔的，当你选择不使用分号结束一行代码时 JavaScript 会自己帮你补上。这种行为可能会导致麻 烦，如当你返回对象字面量，而左括号却在下一行的时候：

```
// 警告： 意外的返回值
function func() {
   return
  // 下面代码不执行
   {
      name : "Batman"
```

```
    }
}
```

如果你希望函数返回一个含有 name 属性的对象，你会惊讶。由于隐含分号，函数返回 undefined。前面的代码等价于：

```
// 警告： 意外的返回值
function func() {
   return undefined;
  // 下面代码不执行
   {
      name : "Batman"
   }
}
```

总之，总是使用花括号，并始终把在与之前的语句放在同一行：

```
function func() {
   return {
      name : "Batman"
   };
}
```

关于分号注：就像使用花括号，你应该总是使用分号，即使他们可由 JavaScript 解析器隐式创建。这不仅促进更科学和更严格的代码，而且有助于解决存有疑惑的地方，就如前面的例子显示。

空格(White Space)

空格的使用同样有助于改善代码的可读性和一致性。在写英文句子的时候，在逗号和句号后面会使用间隔。在 JavaScript 中，你可以按照同样的逻辑在列表模样表达式（相当于逗号）和结束语句（相对于完成了“想法”）后面添加间隔。

适合使用空格的地方包括：

- for 循环分号分开后的的部分：如 `for (var i = 0; i < 10; i += 1) {...}`
- for 循环中初始化的多变量(i 和 max)：`for (var i = 0, max = 10; i < max; i += 1) {...}`
- 分隔数组项的逗号的后面：`var a = [1, 2, 3];`
- 对象属性逗号的后面以及分隔属性名和属性值的冒号的后面：`var o = {a: 1, b: 2};`
- 限定函数参数：`myFunc(a, b, c)`

- 函数声明的花括号的前面：function myFunc() {}
- 匿名函数表达式 function 的后面：var myFunc = function () {};

使用空格分开所有的操作符和操作对象是另一个不错的使用，这意味着在+, -, *, =, <, >, <=, >=, ===, !==, &&, ||, +=等前后都需要空格。

```
// 宽松一致的间距
// 使代码更易读
// 使得更加“透气”
var d = 0,
    a = b + 1;
if (a && b && c) {
    d = a % c;
    a += d;
}

// 反面例子
// 缺失或间距不一
// 使代码变得疑惑
var d = 0,
    a = b + 1;
if (a&&b&&c) {
    d=a % c;
    a+= d;
}
```

最后需要注意的一个空格——花括号间距。最好使用空格：

- 函数、if-else 语句、循环、对象字面量的左花括号的前面({)
- else 或 while 之间的右花括号(})

空格使用的一点不足就是增加了文件的大小，但是压缩无此问题。

有一个经常被忽略的代码可读性方面是垂直空格的使用。你可以使用空行来分隔代码单元，就像是文学作品中使用段落分隔一样。

命名规范(Naming Conventions)

另一种方法让你的代码更具可预测性和可维护性是采用命名规范。这就意味着你需要用同一种形式给你的变量和函数命名。

下面是建议的一些命名规范，你可以原样采用，也可以根据自己的喜好作调整。同样，遵循规范要比规范是什么更重要。

以大写字母写构造函数(Capitalizing Constructors)

JavaScript 并没有类，但有 new 调用的构造函数：

```
var adam = new Person();
```

因为构造函数仍仅仅是函数，仅看函数名就可以帮助告诉你这应该是一个构造函数还是一个正常的函数。

命名构造函数时首字母大写具有暗示作用，使用小写命名的函数和方法不应该使用 new 调用：

```
function MyConstructor() {...}
function myFunction() {...}
```

分隔单词(Separating Words)

当你的变量或是函数名有多个单词的时候，最好单词的分离遵循统一的规范，有一个常见的做法被称作“驼峰(Camel)命名法”，就是单词小写，每个单词的首字母大写。

对于构造函数，可以使用大驼峰式命名法(upper camel case)，如 MyConstructor()。对于函数和方法名称，你可以使用小驼峰式命名法(lower camel case)，像是 myFunction()， calculateArea()和 getFirstName()。

要是变量不是函数呢？开发者通常使用小驼峰式命名法，但还有另外一种做法就是所有单词小写以下划线连接：例如，first_name, favorite_bands, 和 old_company_name，这种标记法帮你直观地区分函数和其他标识——原型和对象。

ECMAScript 的属性和方法均使用 Camel 标记法，尽管多字的属性名称是罕见的（正则表达式对象的 lastIndex 和 ignoreCase 属性）。

其它命名形式(Other Naming Patterns)

有时，开发人员使用命名规范来弥补或替代语言特性。

例如，JavaScript 中没有定义常量的方法（尽管有些内置的像 Number, MAX_VALUE），所以开发者都采用全部单词大写的规范来命名这个程序生命周期中都不会改变的变量，如：

```
// 珍贵常数，只可远观
var PI = 3.14,
    MAX_WIDTH = 800;
```

还有另外一个完全大写的惯例：全局变量名字全部大写。全部大写命名全局变量可以加强减小全局变量数量的实践，同时让它们易于区分。

另外一种使用规范来模拟功能的是私有成员。虽然可以在 JavaScript 中实现真正的私有，但是开发者发现仅仅使用一个下划线前缀来表示一个私有属性或方法会更容易些。考虑下面的例子：

```
var person = {
    getName: function () {
        return this._getFirst() + ' ' + this._getLast();
    },

    _getFirst: function () {
        // ...
    },
    _getLast: function () {
        // ...
    }
};
```

在此例中，getName()就表示公共方法，部分稳定的 API。而_getFirst()和_getLast()则表明了私有。它们仍然是正常的公共方法，但是使用下划线前缀来警告 person 对象的使用者这些方法在下一个版本中时不能保证工作的，是不能直接使用的。注意，JSLint 有些不鸟下划线前缀，除非你设置了 noman 选项为:false。

下面是一些常见的 private 规范：

- 使用尾下划线表示私有，如 name_和 getElements_()
- 使用一个下划线前缀表_protected（保护）属性，两个下划线前缀表示__private（私有）属性
- Firefox 中一些内置的变量属性不属于该语言的技术部分，使用两个前下划线和两个后下划线表示，如：__proto__和__parent__。

注释(Writing Comments)

你必须注释你的代码，即使不会有其他人向你一样接触它。通常，当你深入研究一个问题，你会很清楚的知道这个代码是干嘛用的，但是，当你一周之后再回来看的时候，想必也要耗掉不少脑细胞去搞明白到底怎么工作的。

很显然，注释不能走极端：每个单独变量或是单独一行。但是，你通常应该记录所有的函数，它们的参数和返回值，或是任何不寻常的技术和方法。要想到注 释可以给你代码未来的阅读者以诸多提示；阅读者需要的是（不要读太多的东西）仅注释和函数属性名来理解你的代码。例如，当你有五六行程序执行特定的任务， 如果你提供了一行代码目的以及为什么在这里的描述的话，阅读者就可以直接跳过这段细节。没有硬性规定注释代码比，代码的某些部分（如正则表达式）可能注释 要比代码多。

最重要的习惯，然而也是最难遵守的，就是保持注释的及时更新，因为过时的注释比没有注释更加的误导人。

关于作者（About the Author ）

Stoyan Stefanov 是 Yahoo!web 开发人员，多个 O'Reilly 书籍的作者、投稿者和技术评审。他经常在会议和他的博客 www.phpied.com 上发表 web 开发主题的演讲。Stoyan 还是 smush.it 图片优化工具的创造者，YUI 贡献者，雅虎性能优化工具 YSlow 2.0 的架构设计师。

本文转自：http://www.zhangxinxu.com/wordpress/?p=1173

英文原文：http://net.tutsplus.com/tutorials/javascript-ajax/the-essentials-of-writing-high-quality-javascript/