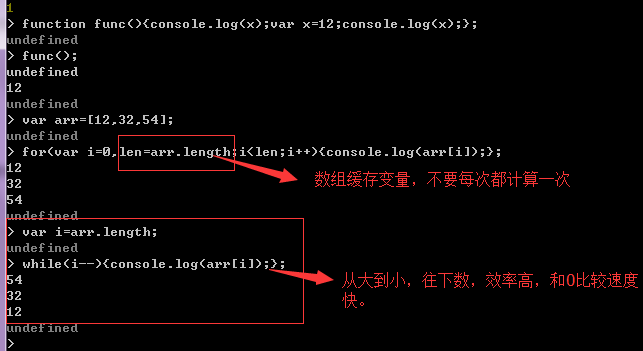
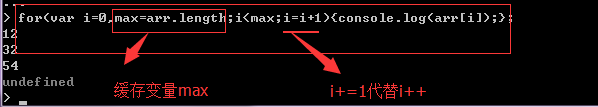
关于循环高效遍历





对象的遍历for---in



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。

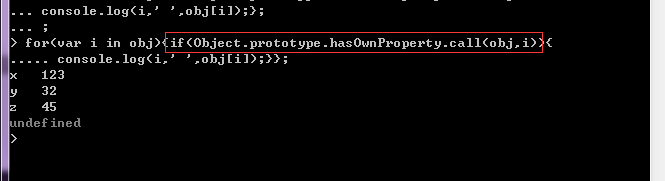
// 1.  
// for-in 循环  
for (var i in man) {  
 if (man.hasOwnProperty(i)) { // 过滤  
 console.log(i, ":", man[i]);  
 }  
}  
/\* 控制台显示结果  
hands : 2  
legs : 2  
heads : 1  
\*/  
// 2.  
// 反面例子:  
// for-in loop without checking hasOwnProperty()  
for (var i in man) {  
 console.log(i, ":", man[i]);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



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。

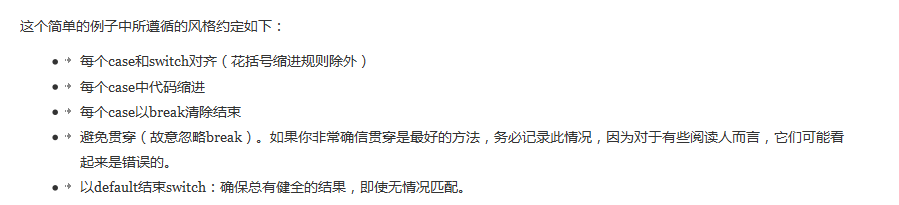
因此，不增加内置原型是最好的。你可以指定一个规则，仅当下面的条件均满足时例外：

* 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...  
 };  
}







# 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。

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）。

(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。

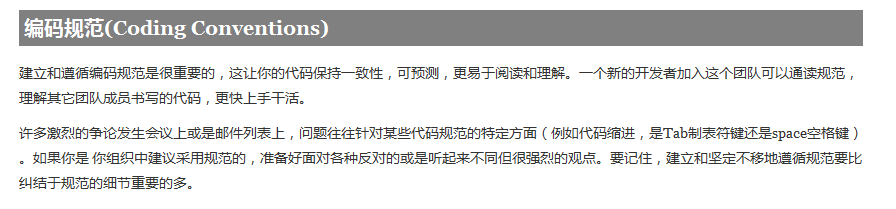
var month = "06",  
 year = "09";  
month = parseInt(month, 10);  
year = parseInt(year, 10);

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。







为了长远打算，最好总是使用花括号，即时值一行代码：

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：

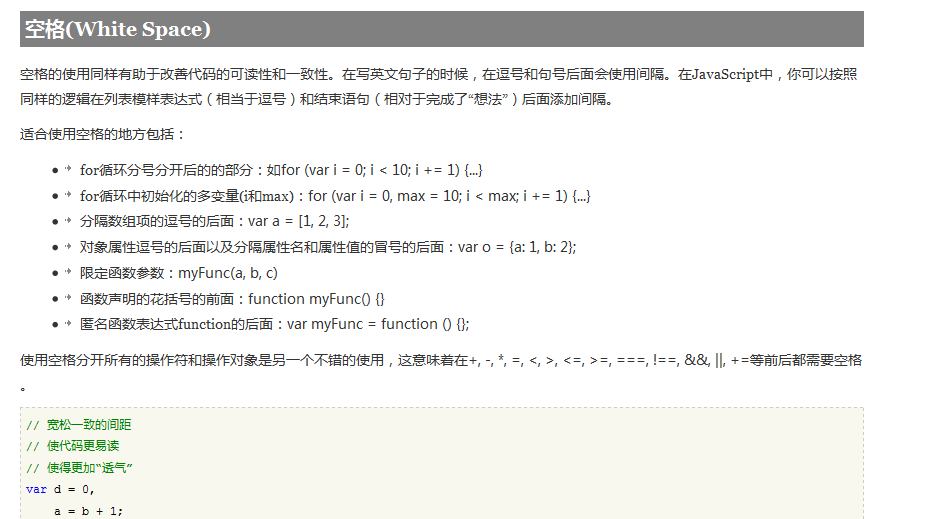
// 警告： 意外的返回值  
function func() {  
 return  
 // 下面代码不执行  
 {  
 name : "Batman"  
 }  
}

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"  
 };  
}



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(})



# 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 () {  
 // ...  
 }  
};

