**《JavaScript高级程序设计(第3版)》读书笔记**

## 第1章 JavaScript 简介

一个完整的JavaScript 实现应该由下列三个不同的部分组成：

* ECMAScript，由ECMA-262 定义，提供核心语言功能；
* 文档对象模型（DOM），提供访问和操作网页内容的方法和接口；
* 浏览器对象模型（BOM），提供与浏览器交互的方法和接口。

我们常见的Web 浏览器只是ECMAScript 实现可能的宿主环境之一。宿主环境不仅提供基本的ECMAScript 实现，同时也会提供该语言的扩展，以便语言与环境之间对接交互。而这些扩展——如DOM，则利用ECMAScript 的核心类型和语法提供更多更具体的功能，以便实现针对环境的操作。其他宿主环境包括Node（一种服务端JavaScript 平台）和Adobe Flash。

ECMAScript 就是对实现该标准规定的各个方面内容的语言的描述。

JavaScript 实现了ECMAScript，Adobe ActionScript 同样也实现了ECMAScript。

文档对象模型（DOM，[Document Object Model](http://www.w3.org/DOM/)）是针对XML 但经过扩展用于HTML 的应用程序编程接口（API，Application Programming Interface）。DOM把整个页面映射为一个多层节点结构。HTML或XML 页面中的每个组成部分都是某种类型的节点，这些节点又包含着不同类型的数据。DOM 被设计为与语言无关。

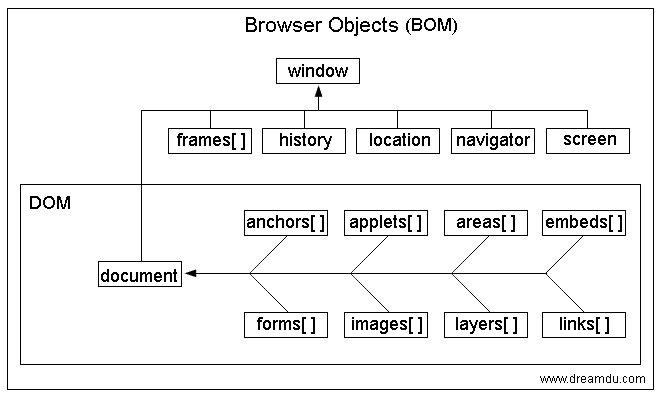
DOM 的发展经历了不同级别，每一级别在前一级别基础上添加了新功能。

级别 1 涵盖了基本功能；

级别 2 添加了名称空间支持、UI 事件模型、迭代器等功能；

级别 3 添加了可从 XML 文档文件进行加载和保存的 API，并集成了 XPath，添加了验证支持等。

BOM(Browser Object Model) 是指浏览器对象模型，是用于描述这种对象与对象之间层次关系的模型，浏览器对象模型提供了独立于内容的、可以与浏览器窗口进行互动的对象结构。BOM由多个对象组成，其中代表浏览器窗口的Window对象是BOM的顶层对象，其他对象都是该对象的子对象。



## 第2章 在HTML中使用JavaScript

所有<script>元素都会按照它们在页面中出现的先后顺序依次被解析。在不使用defer 和

async 属性的情况下，只有在解析完前面<script>元素中的代码之后，才会开始解析后面

<script>元素中的代码。

由于浏览器会先解析完不使用defer 属性的<script>元素中的代码，然后再解析后面的内容，

所以一般应该把<script>元素放在页面最后，即主要内容后面，</body>标签前面。(过时了)

使用defer 属性可以让脚本在文档完全呈现之后再执行。延迟脚本总是按照指定它们的顺序执行。

使用async 属性可以表示当前脚本不必等待其他脚本，也不必阻塞文档呈现。不能保证异步脚本按照它们在页面中出现的顺序执行。

对于标准模式，可以通过使用下面任何一种文档类型来开启：

<!-- HTML 4.01 严格型 -->

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<!-- XHTML 1.0 严格型 -->

<!DOCTYPE html PUBLIC

"-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- HTML 5 -->

<!DOCTYPE html>

## 第3章 基本概念

let 和yield 是第5 版新增的保留字；其他保留字都是第3 版定义的。为了最大程度地保

证兼容性，建议读者将第3 版定义的保留字外加let 和yield 作为编程时的参考。

给未经声明的变量赋值在严格模式下会导致抛出ReferenceError 错误。

在严格模式下，不能定义名为eval 或arguments 的变量，否则会导致语法错误。

包含undefined 值的变量与尚未定义的变量还是不一样的。看看下面这个例子：

var message; // 这个变量声明之后默认取得了undefined 值

// 下面这个变量并没有声明

// var age;

alert(message); // "undefined"

alert(age); // 产生错误

对于那些极大或极小的数值，可以用e 表示法（即科学计数法）表示的浮点数值表示。用e 表示法表示的数值等于e 前面的数值乘以10 的指数次幂。ECMAScript 中e 表示法的格式也是如此，即前面是一个数值（可以是整数也可以是浮点数），中间是一个大写或小写的字母E，后面是10 的幂中的指数，该幂值将用来与前面的数相乘。下面是一个使用e 表示法表示数值的例子：

var floatNum = 3.125e7; // 等于31250000

在这个例子中，使用e 表示法表示的变量floatNum 的形式虽然简洁，但它的实际值则是31250000。在此，e 表示法的实际含义就是“3.125 乘以107”。

也可以使用e 表示法表示极小的数值，如0.00000000000000003，这个数值可以使用更简洁的3e17表示。在默认情况下，ECMASctipt 会将那些小数点后面带有6 个零以上的浮点数值转换为以e 表示法表示的数值（例如，0.0000003 会被转换成3e7）。

浮点数值的最高精度是17 位小数，但在进行算术计算时其精确度远远不如整数。例如，0.1 加0.2的结果不是0.3，而是0.30000000000000004。这个小小的舍入误差会导致无法测试特定的浮点数值。

例如：

if (a + b == 0.3){ // 不要做这样的测试！

alert("You got 0.3.");

}

在这个例子中，我们测试的是两个数的和是不是等于0.3。如果这两个数是0.05 和0.25，或者是0.15和0.15 都不会有问题。而如前所述，如果这两个数是0.1 和0.2，那么测试将无法通过。因此，永远不要测试某个特定的浮点数值。

关于浮点数值计算会产生舍入误差的问题，有一点需要明确：这是使用基于IEEE754 数值的浮点计算的通病，ECMAScript 并非独此一家；其他使用相同数值格式的语言也存在这个问题。

alert(isNaN("blue")); //true（不能转换成数值）

字符串"blue"不能被转换成数值，因此函数返回了true。

在使用parseInt()解析像八进制字面量的字符串时，ECMAScript 3 和5 存在分歧。例如：

//ECMAScript 3 认为是56（八进制），ECMAScript 5 认为是70（十进制）

var num = parseInt("070");

在ECMAScript 3 JavaScript 引擎中，"070"被当成八进制字面量，因此转换后的值是十进制的56。

而在ECMAScript 5 JavaScript 引擎中，parseInt()已经不具有解析八进制值的能力，因此前导的零会被认为无效，从而将这个值当成"70"，结果就得到十进制的70。在ECMAScript 5 中，即使是在非严格模式下也会如此。

为了消除在使用parseInt()函数时可能导致的上述困惑，可以为这个函数提供第二个参数：转换时使用的基数（即多少进制）。不指定基数意味着让parseInt()决定如何解析输入的字符串，因此为了避免错误的解析，我们建议无论在什么情况下都明确指定基数，例如：

var num = parseInt("0xAF", 16); //175

由于parseFloat()只解析十进制值，因此它没有用第二个参数指定基数的用法。

如果字符串包含的是一个可解析为整数的数（没有小数点，或者小数点后都是零），parseFloat()会返回整数。

var num1 = parseFloat("1234blue"); //1234 （整数，不是浮点数）

**字符串的特点**

ECMAScript 中的字符串是不可变的，也就是说，字符串一旦创建，它们的值就不能改变。要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量，例如：

var lang = "Java";

lang = lang + "Script";

以上示例中的变量lang 开始时包含字符串"Java"。而第二行代码把lang 的值重新定义为"Java"与"Script"的组合，即"JavaScript"。实现这个操作的过程如下：首先创建一个能容纳10 个字符的新字符串，然后在这个字符串中填充"Java"和"Script"，最后一步是销毁原来的字符串"Java"和字符串"Script"，因为这两个字符串已经没用了。这个过程是在后台发生的，而这也是在某些旧版本的浏览器（例如版本低于1.0 的Firefox、IE6 等）中拼接字符串时速度很慢的原因所在。但这些浏览器后来的版本已经解决了这个低效率问题。

var value1 = 10;

var value2 = true;

var value3 = null;

var value4;

alert(String(value1)); // "10"

alert(String(value2)); // "true"

alert(String(value3)); // "null"

alert(String(value4)); // "undefined"

要把某个值转换为字符串，可以使用加号操作符（3.5 节讨论）把它与一个字符串（""）加在一起。

在ECMAScript 中，如果不给构造函数传递参数，则可以省略后面的那一对圆括号。

var o = new Object; // 有效，但不推荐省略圆括号

ECMAScript 对象的属性没有顺序。因此，通过for-in 循环输出的属性名的顺序是不可预测的。具体来讲，所有属性都会被返回一次，但返回的先后次序可能会因浏览器而异。

但是，如果表示要迭代的对象的变量值为null 或undefined，for-in 语句会抛出错误。

ECMAScript 5 更正了这一行为；对这种情况不再抛出错误，而只是不执行循环体。为了保证最大限度的兼容性，建议在使用for-in 循环之前，先检测确认该对象的值不是null 或undefined。

## 第4章 变量、作用域和内存问题

基本类型值在内存中占据固定大小的空间，因此被保存在栈内存中；

我们不能给基本类型的值添加属性，尽管这样做不会导致任何错误。比如：

var name = "Nicholas";

name.age = 27;

alert(name.age); //undefined

执行环境（execution context，为简单起见，有时也称为“环境”）是JavaScript 中最为重要的一个概念。执行环境定义了变量或函数有权访问的其他数据，决定了它们各自的行为。每个执行环境都有一个与之关联的变量对象（variable object），环境中定义的所有变量和函数都保存在这个对象中。虽然我们编写的代码无法访问这个对象，但解析器在处理数据时会在后台使用它。

全局执行环境是最外围的一个执行环境。根据ECMAScript 实现所在的宿主环境不同，表示执行环境的对象也不一样。在Web 浏览器中，全局执行环境被认为是window 对象（第7 章将详细讨论），因此所有全局变量和函数都是作为window 对象的属性和方法创建的。某个执行环境中的所有代码执行完毕后，该环境被销毁，保存在其中的所有变量和函数定义也随之销毁（全局执行环境直到应用程序退出——例如关闭网页或浏览器——时才会被销毁）。

每个函数都有自己的执行环境。当执行流进入一个函数时，函数的环境就会被推入一个环境栈中。而在函数执行之后，栈将其环境弹出，把控制权返回给之前的执行环境。ECMAScript 程序中的执行流正是由这个方便的机制控制着。

当代码在一个环境中执行时，会创建变量对象的一个作用域链（scope chain）。作用域链的用途，是保证对执行环境有权访问的所有变量和函数的有序访问。作用域链的前端，始终都是当前执行的代码所在环境的变量对象。如果这个环境是函数，则将其活动对象（activation object）作为变量对象。活动对象在最开始时只包含一个变量，即arguments 对象（这个对象在全局环境中是不存在的）。作用域链中的下一个变量对象来自包含（外部）环境，而再下一个变量对象则来自下一个包含环境。这样，一直延续到全局执行环境；全局执行环境的变量对象始终都是作用域链中的最后一个对象。

标识符解析是沿着作用域链一级一级地搜索标识符的过程。搜索过程始终从作用域链的前端开始，然后逐级地向后回溯，直至找到标识符为止（如果找不到标识符，通常会导致错误发生）。

JavaScript 没有块级作用域经常会导致理解上的困惑。在其他类C 的语言中，由花括号封闭的代码块都有自己的作用域（如果用ECMAScript 的话来讲，就是它们自己的执行环境），因而支持根据条件来定义变量。例如，下面的代码在JavaScript 中并不会得到想象中的结果：

if (true) {

var color = "blue";

}

alert(color); //"blue"

这里是在一个if 语句中定义了变量color。如果是在C、C++或Java 中，color 会在if 语句执行完毕后被销毁。但在JavaScript 中，if 语句中的变量声明会将变量添加到当前的执行环境（在这里是全局环境）中。在使用for 语句时尤其要牢记这一差异，例如：

for (var i=0; i < 10; i++){

doSomething(i);

}

alert(i); //10

对于有块级作用域的语言来说，for 语句初始化变量的表达式所定义的变量，只会存在于循环的环境之中。而对于JavaScript 来说，由for 语句创建的变量i 即使在for 循环执行结束后，也依旧会存在于循环外部的执行环境中。

变量查询也不是没有代价的。很明显，访问局部变量要比访问全局变量更快，因为不用向上搜索作用域链。JavaScript 引擎在优化标识符查询方面做得不错，因此这个差别在将来恐怕就可以忽略不计了。

我们知道，IE 中有一部分对象并不是原生JavaScript 对象。例如，其BOM 和DOM 中的对象就是使用C++以COM（Component Object Model，组件对象模型）对象的形式实现的，而COM 对象的垃圾收集机制采用的就是引用计数策略。因此，即使IE 的JavaScript 引擎是使用标记清除策略来实现的，但JavaScript 访问的COM 对象依然是基于引用计数策略的。换句话说，只要在IE 中涉及COM 对象，就会存在循环引用的问题。下面这个简单的例子，展示了使用COM 对象导致的循环引用问题：

var element = document.getElementById("some\_element");

var myObject = new Object();

myObject.element = element;

element.someObject = myObject;

这个例子在一个DOM 元素（element）与一个原生JavaScript 对象（myObject）之间创建了循环引用。其中，变量myObject 有一个名为element 的属性指向element 对象；而变量element 也有一个属性名叫someObject 回指myObject。由于存在这个循环引用，即使将例子中的DOM 从页面中移除，它也永远不会被回收。

为了避免类似这样的循环引用问题，最好是在不使用它们的时候手工断开原生JavaScript 对象与DOM 元素之间的连接。例如，可以使用下面的代码消除前面例子创建的循环引用：

myObject.element = null;

element.someObject = null;

将变量设置为null 意味着切断变量与它此前引用的值之间的连接。当垃圾收集器下次运行时，就会删除这些值并回收它们占用的内存。

IE9 把BOM 和DOM 对象都转换成了真正的JavaScript 对象。这样，就避免了两种垃圾收集算法并存导致的问题，也消除了常见的内存泄漏现象。