

JS数据类型详解 11-14 , 至数组 , &globe &Math

笔记本 : javascript
创建时间 : 2017/3/25 10:38
位置 : 25°51'26 N 114°55'34 E
URL : <https://segmentfault.com/a/1190000005022170>

漫谈程序员系列：看看你离优秀有多远

<http://blog.csdn.net/foruok/article/details/40075201?spm=5176.100239.blogcont69700.4.mAWfm1>

基本概念--语法：

- 区分大小写
- 标识符：第一个字符必须是字母，下划线或者\$。其它字符可以是字母，下划线或者\$+数字
- 严格模式：要在整个脚本中启用严格模式，可以在顶部添加如下代码：“use strict”
- 关键字和保留字：关键字：break do instanceof typeof case else new var catch finally return void continue for switch void debugger function this with default if throw delete in try
- 保留字：abstract enum int short boolean export interface static byte extends long super char final native synchronized class float package throws debugger implement protected volatile double import public
- 可以用一条语句定义多个变量：var one=1,second=2,three=3;
- 5个的基本数据类型：undefined null boolean number string
- 著名的 0.1 + 0.2 === 0.30000000000000004
- isNaN()确定一个数值是不是有穷的
- isNaN
- number ("hello world") 与 parseInt ("AF",16) 函数的区别
-
- string类型的单双引号没有区别，只是前后的单双引号要一致。转义字符，text.length，num.toString(8);八进制。
- object的每个实例都有以下属性与方法：constructor hasOwnProperty isPrototypeOf propertyIsEnumerable toLocaleString toString valueOf
- ++ ~ & | ^ << >> && || +-* / %
- >< != !==
- for while if switch
- ECMAScript中所有参数传递的都是值，不可能通过引用传递参数。
- 检测类型：alert(typeof o) //object
- 判断一个变量是不是一个对象的 alert (pattern instanceof RegExp) //变量 pattern 是RegExp 吗
- 块级作用域{}里的内容：if({})
- 一旦数据不用，就将其设置为null，来释放其引用---->手动解除引用
- array push是向数组末端添加项的方法 pop是从数组末尾移除最后一项。例子：var value=color.pop(); var value=color.push("black");
- shift/push 先出/后进 unshift /pop 反方向
- arr.sort(compare);

```
var color=['red','green'];
var color2=color.concat('yellow',['black','brown']); //拼接成新数组
var color3=colors.slice(1); //在只有一个参数时，返回全部参数
var color4=colors.slice(1,1); //green
```

splice主要用途是向数组的中部插入项：

删除：插入，替换

```
var color=['red','green']; //移除后只剩green
var removed=colors.splice(0,1); //removed会返回已移除的项
color.splice(1,0,'yellow'); //0为移除项的个数，在后面全是需添加的项数
```

indexof() 与 lastIndexof() 要

arr.indexOf(4); //查找值为4，在数组中的位置。没有查到返回-1

迭代方法：

every()/map():对数组每一项运行给定函数，如果改函数对每一项都返回true，则返回true

filter(): ,返回该函数会返回true的项组成的数组
forEach(): ,这个函数没有返回值
map(): ,返回每个函数调用的结果组成的函数
使用方法 :

```
var numbers= [1,2,3,4,5,6,7];
var filterResult=numbers.filter(function(item,index,array){
return(item>2);
})
console.log(filterResult);/
```

```
▼ Array[5]
  0: 3
  1: 4
  2: 5
  3: 6
  4: 7
  length: 5
  ► __proto__: Array[0]
```

缩小的方法 : ECMAScript新增的 : reduce()和reduceRight();

这两个方法都会迭代数组的所有项，然后构建一个最终返回的值，，其中reduce方法从数组的第一项开始，逐个遍历到最后。reduceRight则从数组的最后一项开始，向前遍历到一项。包含的参数：前一个值，当前值，项的索引和数组对象

```
var value=[1,2,3,4,5];
var sum = value.reduce(function(prev,cur,index,array){
return prev+cur;//下一次直接当作第一次参数传下去
})
console.log(sum);//15---数组求和
```

```
▼ __proto__: Array[0]
  ► concat: concat()
  ► constructor: Array()
  ► copyWithin: copyWithin()
  ► entries: entries()
  ► every: every()
  ► fill: fill()
  ► filter: filter()
  ► find: find()
  ► findIndex: findIndex()
  ► forEach: forEach()
  ► includes: includes()
  ► indexOf: indexOf()
  ► join: join()
  ► keys: keys()
  ► lastIndexOf: lastIndexOf()
  length: 0
  ► map: map()
  ► pop: pop()
  ► push: push()
  ► reduce: reduce()
  ► reduceRight: reduceRight()
  ► reverse: reverse()
  ► shift: shift()
  ► slice: slice()
  ► some: some()
  ► sort: sort()
  ► splice: splice()
  ► toLocaleString: toLocaleString()
  ► toString: toString()
  ► unshift: unshift()
  ► Symbol(Symbol.iterator): values()
  ► Symbol(Symbol.unscopables): Object
  ► __proto__: Object
```

Number类型

```
num.toFixed(2) === 10.00
num.toExponential(1); 1.0e+1; // 这样太小的值这样用，有点不好了
toPrecision(): 返回合适的格式
String类型
var str = "string";
str.charAt(1); // 字符
如果不想得到字符，想得到字符编码：可以使用 charCodeAt(1);
在ES5后面，就有了 str[1] 方式的取值，
var str = str1.concat("world"); 获取新字符串。当大部分情况，还是使用 "+" 来，特别是多个字符串时。
slice, substr, substring 都是基于子串创建的数组，接收两个参数，第一个是开始的位置，第二个结束的位置，substring 第二个参数是指字长
indexOf lastIndexOf 搜索子字符串，开头 末尾都可以接收，两个参数，表示从那个字符串开始搜索
trim()
toLowerCase() toLocaleLowerCase() | toUpperCase() toLocaleUpperCase() 针对地区
localeCompare() 方法：比较两个字符串，yellow--brick(1)---yellow(0)---zoo(-1).
alert(String.fromCharCode(104, 101, 108, 111)); // hello 字符串
Global类型
```

```
// Global 函数
var url = "https://www.axie.cc/#test 我";
console.log(encodeURIComponent(url));
console.log(encodeURI(url));
```

<https://www.axie.cc/#test%E6%88%91> // 替换所有非字母数字字符
<https://www.axie.cc/#test%E6%88%91>

Math类型

```
var max/min = Math.max/min(3, 4, 3);
```

技巧：

```
var values = [3, 4, 3];
```

```
var max = Math.max.apply(Math, values);
```

Math.ceil 向上取整 floor round 四舍五入

公式：值 = Math.floor(Math.random() * 可能值的总数 + 第一个可能的值)

```
console.log(Math.random()); // 0.2200039572771555
// 范围函数 可以运用到数组中，随机。
function selectRange(min, max){
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
console.log(selectRange(2, 9));
```

=====以下为网上的资料=====

Javascript 面向对象特性

很少有人对JavaScript的面向对象特性进行系统的分析。我希望接下来的文字让你了解到这个语言最少为人知的一面。

1. JavaScript中的类型

虽然JavaScript是一个基于对象的语言，但对象(Object)在JavaScript中不是第一型的。JS是以函数(Function)为第一型的语言。这样说，不但是因为JS中的函数具有高级语言中的函数的各种特性，而且也因为在JS中，Object也是由函数来实现的。——关于这一点，可以在后文中“构造与析构”部分看到更进一步的说明。

JS中是弱类型的，他的内置类型简单而且清晰：

```
undefined : 未定义
number : 数字
boolean : 布尔值
string : 字符串
function : 函数
object : 对象
```

1). undefined类型

=====

在IE5及以下版本中，除了直接赋值和typeof()之外，其它任何对undefined的操作都将导致异常。如果需要知道一个变量是否是undefined，只能采用typeof()的方法：

```
<script>
var v;
if (typeof(v) == 'undefined') {
// ...
}
</script>
```

但是在IE5.5及以上版本中，undefined是一个已实现的系统保留字。因此可以用undefined来比较和运算。检测一个值是否是undefined的更简单方法可以是：

```
<script>
var v;
if (v === undefined) {
// ...
}
</script>
```

因此为了使得核心代码能(部分地)兼容IE5及早期版本，Romo核心单元中有一行代码用来“声明”一个undefined值：

```
//-----
// code from Qomolangma, in JSEnhance.js
//-----
var undefined = void null;
```

这一行代码还有一点是需要说明的，就是void语句的应用。void表明“执行其后的语句，且忽略返回值”。因此在void之后可以出现能被执行的任何“单个”语句。而执行的结果就是undefined。当然，如果你愿意，你也可以用下面的代码之一“定义undefined”。

```
//-----
// 1. 较复杂的方法，利用一个匿名的空函数执行的返回
//-----
var undefined = function(){}();
```



```
//-----
// 2. 代码更简洁，但不易懂的方法
//-----
var undefined = void 0;
```

void也能像函数一样使用，因此void(0)也是合法的。有些时候，一些复杂的语句可能不能使用void的关键字形式，而必须要使用void的函数形式。例如：

```
//-----
// 必须使用void()形式的复杂表达式
//-----
void(i=1); // 或如下语句:
void(i=1, i++);
```

2). number类型

=====

JavaScript中总是处理浮点数，因此它没有象Delphi中的MaxInt这样的常量，反而是有这样两个常值定义：

Number.MAX_VALUE : 返回 JScript 能表达的最大的数。约等于 1.79E+308。
Number.MIN_VALUE : 返回 JScript 最接近0的数。约等于 2.22E-308。

因为没有整型的缘故，因此在一些关于CSS和DOM属性的运算中，如果你期望取值为整数2，你可能会得到字符串“2.0”——或者类似于此的一些情况。这种情况下，你可能需要用到全局对象(Gobal)的parseInt()方法。

全局对象(Gobal)中还有两个属性与number类型的运算有关：

Nan : 算术表达式的运算结果不是数字，则返回NaN值。

Infinity : 比MAX_VALUE更大的数。

如果一个值是NaN，那么他可以通过全局对象(Gobal)的isNaN()方法来检测。然而两个NaN值之间不是互等的。如下例：

```
//-----  
// NaN的运算与检测  
//-----  
var  
v1 = 10 * 'a';  
v2 = 10 * 'a';  
document.writeln(isNaN(v1));  
document.writeln(isNaN(v2));  
document.writeln(v1 == v2);
```

全局对象(Gobal)的Infinity表示比最大的数 (Number.MAX_VALUE) 更大的值。在JS中，它在数学运算时的价值与正无穷是一样的。——在一些实用技巧中，它也可以用来做一个数组序列的边界检测。

Infinity在Number对象中被定义为POSITIVE_INFINITY。此外，负无穷也在Number中被定义：

Number.POSITIVE_INFINITY : 比最大正数 (Number.MAX_VALUE) 更大的值。正无穷。
Number.NEGATIVE_INFINITY : 比最小负数 (-Number.MAX_VALUE) 更小的值。负无穷。

与NaN不同的是，两个Infinity(或-Infinity)之间是互等的。如下例：

```
//-----  
// Infinity的运算与检测  
//-----  
var  
v1 = Number.MAX_VALUE * 2;  
v2 = Number.MAX_VALUE * 3;  
document.writeln(v1);  
document.writeln(v2);  
document.writeln(v1 == v2);
```

在Global中其它与number类型相关的方法有：

isFinite() : 如果值是NaN/正无穷/负无穷，返回false，否则返回true。

parseFloat() : 从字符串(的前缀部分)取一个浮点数。不成功则返回NaN。

3). boolean类型

=====

(略)

4). string类型

=====

JavaScript中的String类型原本没有什么特殊的，但是JavaScript为了适应“浏览器实现的超文本环境”，因此它具有一些奇怪的方法。例如：

link() : 把一个有HREF属性的超链接标签<A>放在String对象中的文本两端。

big() : 把一对<big>标签放在String对象中的文本两端。

以下方法与此类同：

anchor()
blink()
bold()
fixed()
fontcolor()
fontsize()
italics()
small()
strike()
sub()
sup()

除此之外，string的主要复杂性来自于在JavaScript中无所不在的toString()

方法。这也是JavaScript为浏览器环境而提供的一个很重要的方法。例如我们声明一个对象，但是要用document.writeln()来输出它，在IE中会显示什么呢？

下例说明这个问题：

```
//-----
// toString()的应用
//-----
var
s = new Object();

s.v1 = 'hi,';
s.v2 = 'test!';
document.writeln(s);
document.writeln(s.toString());

s.toString = function() {
return s.v1 + s.v2;
}
document.writeln(s);
```

在这个例子中，我们看到，当一个对象没有重新声明(覆盖)自己toString()方法的时候，那么它作为字符串型态使用时(例如被writeln)，就会调用Java Script环境缺省的toString()。反过来，你也可以重新定义JavaScript理解这个对象的方法。

很多JavaScript框架，在实现“模板”机制的时候，就利用了这个特性。例如他们用这样定义一个FontElement对象：

```
//-----
// 利用toString()实现模板机制的简单原理
//-----
function FontElement(innerHTML) {
this.face = '宋体';
this.color = 'red';
// more...

var ctx = innerHTML;
this.toString = function() {
return '<Font FACE=' + this.face + '" COLOR=' + this.color + '>' +
+ ctx
+ '</FONT>';
}
}

var obj = new FontElement('这是一个测试。');

// 留意下面这行代码的写法
document.writeln(obj);
```

5). function类型

=====

javascript函数具有很多特性，除了面向对象的部分之外(这在后面讲述)，它自己的一些独特特性应用也很广泛。

首先javascript中的每个函数，在调用过程中可以执有一个arguments对象。这个对象是由脚本解释环境创建的，你没有别的方法来自己创建一个arguments对象。

arguments可以看成一个数组：它有length属性，并可以通过arguments[n]的方式来访问每一个参数。然而它最重要的，却是可以通过 callee 属性来得到正在执行的函数对象的引用。

接下的问题变得很有趣：Function对象有一个 caller 属性，指向正在调用当前

函数的父函数对象的引用。

——我们已经看到，我们可以在JavaScript里面，通过callee/caller来遍历执行期的调用栈。由于arguments事实上也是Function的一个属性，因此我们事实上也能遍历执行期调用栈上的每一个函数的参数。下面的代码是一个简单的示例：

```
-----  
// 调用栈的遍历  
-----  
function foo1(v1, v2) {  
    foo2(v1 * 100);  
}  
  
function foo2(v1) {  
    foo3(v1 * 200);  
}  
  
function foo3(v1) {  
    var foo = arguments.callee;  
    while (foo && (foo != window)) {  
        document.writeln('调用参数 : <br>', '-----<br>');  
  
        var args = foo.arguments, argn = args.length;  
        for (var i=0; i<argn; i++) {  
            document.writeln('args[' + i + ']: ' + args[i], '<br>');  
        }  
        document.writeln('<br>');  
  
        // 上一级  
        foo = foo.caller;  
    }  
}  
  
// 运行测试  
foo1(1, 2);
```

2. JavaScript面向对象的支持

在前面的例子中其实已经讲到了object类型的“类型声明”与“实例创建”。在JavaScript中，我们需要通过一个函数来声明自己的object类型：

```
-----  
// JavaScript中对象的类型声明的形式代码  
// (以后的文档中，“对象名”通常用MyObject来替代)  
-----  
function 对象名(参数表) {  
    this.属性 = 初始值;  
  
    this.方法 = function(方法参数表) {  
        // 方法实现代码  
    }  
}
```

然后，我们可以通过这样的代码来创建这个对象类型的一个实例：

```
-----  
// 创建实例的形式代码  
// (以后的文档中，“实例变量名”通常用obj来替代)  
-----  
var 实例变量名 = new 对象名(参数表);
```

接下来我们来看“对象”在JavaScript中的一些具体实现和奇怪特性。

1). 函数在JavaScript的面向对象机制中的五重身份

“对象名”——如MyObject()——这个函数充当了以下语言角色：

- (1) 普通函数
- (2) 类型声明
- (3) 类型的实现
- (4) 类引用
- (5) 对象的构造函数

一些程序员(例如Delphi程序员)习惯于类型声明与实现分开。例如在delphi中，Interface节用于声明类型或者变量，而implementation节用于书写类型的实现代码，或者一些用于执行的函数、代码流程。

但在JavaScript中，类型的声明与实现是混在一起的。一个对象的类型(类)通过函数来声明，this.xxxx表明了该对象可具有的属性或者方法。

这个函数的同时也是“类引用”。在JavaScript，如果你需要识别一个对象的具体型别，你需要执有一个“类引用”。——当然，也就是这个函数的名字。instanceof运算符就用于识别实例的类型，我们来看一下它的应用：

```
//-----  
// JavaScript中对象的类型识别  
// 语法: 对象实例 instanceof 类引用  
//-----  
function MyObject() {  
    this.data = 'test data';  
}  
  
// 这里MyObject()作为构造函数使用  
var obj = new MyObject();  
var arr = new Array();  
  
// 这里MyObject作为类引用使用  
document.writeln(obj instanceof MyObject);  
document.writeln(arr instanceof MyObject);  
  
=====
```

(未完待续)

=====

接下来的内容：

2. JavaScript面向对象的支持

- 2). 反射机制在JavaScript中的实现
- 3). this与with关键字的使用
- 4). 使用in关键字的运算
- 5). 使用instanceof关键字的运算
- 6). 其它与面向对象相关的关键字

3. 构造与析构

4. 实例和实例引用

5. 原型问题

6. 函数的上下文环境

7. 对象的类型检查问题

2). 反射机制在JavaScript中的实现

JavaScript中通过for..in语法来实现了反射机制。但是JavaScript中并不明确区分“属性”与“方法”，以及“事件”。因此，对属性的类型考查在JS中是个问题。下面的代码简单示例for..in的使用与属性识别：

```
//-----
// JavaScript中for..in的使用和属性识别
//-----
var _r_event = _r_event = /^[Oo]n.*/;
var colorSetting = {
method: 'red',
event: 'blue',
property: ''
}

var obj2 = {
a_method : function() {},
a_property: 1,
onclick: undefined
}

function propertyKind(obj, p) {
return (_r_event.test(p) && (obj[p]==undefined || typeof(obj[p])=='function')) ? 'event'
: (typeof(obj[p])=='function') ? 'method'
: 'property';
}

var objectArr = ['window', 'obj2'];

for (var i=0; i<objectArr.length; i++) {
document.writeln('<p>for ', objectArr[i], '<hr>');

var obj = eval(objectArr[i]);
for (var p in obj) {
var kind = propertyKind(obj, p);
document.writeln('obj.', p, ' is a ', kind.fontcolor(colorSetting[kind]), ': ', obj[p], '<br>');
}

document.writeln('</p>');
}
```

一个常常被开发者忽略的事实是：JavaScript本身是没有事件(Event)系统的。通常我们在JavaScript用到的onclick等事件，其实是IE的DOM模型提供的。从更内核的角度上讲：IE通过COM的接口属性公布了一组事件接口给DOM。

有两个原因，使得在JS中不能很好的识别“一个属性是不是事件”：

- COM接口中本身只有方法，属性与事件，都是通过一组get/set方法来公布的。
- JavaScript中，本身并没有独立的“事件”机制。

因此我们看到event的识别方法，是检测属性名是否是以'on'字符串开头(以'On'开头的是Qomo的约定)。接下来，由于DOM对象中的事件是可以不指定处理函数的，这种情况下事件句柄为null值(Qomo采用相同的约定)；在另外的一些情况下，用户可能象obj2这样，定义一个值为undefined的事件。因此“事件”的判定条件被处理成一个复杂的表达式：

("属性以on/On开头" && ("值为null/undefined" || "类型为function"))

另外，从上面的这段代码的运行结果来看。对DOM对象使用for..in，是不能列举出对象方法来的。

最后说明一点。事实上，在很多语言的实现中，“事件”都不是“面向对象”的语言特性，而是由具体的编程模型来提供的。例如Delphi中的事件驱动机制，是由Win32操作系统中的窗口消息机制来提供，或者由用户代码在Component/Class中主动调用事件处理函数来实现。

“事件”是一个“如何驱动编程模型”的机制 / 问题，而不是语言本身的问题。然而以PME(property/method/event)为框架的OOP概念，已经深入人心，所以当编程语言或系统表现出这些特性来的时候，就已经没人关心“event究竟是谁实现”的了。

3). this与with关键字的使用

在JavaScript的对象系统中，this关键字用在两种地方：

- 在构造器函数中，指代新创建的对象实例
- 在对象的方法被调用时，指代调用该方法的对象实例

如果一个函数被作为普通函数(而不是对象方法)调用，那么在函数中的this关键字将指向window对象。与此相同的，如果this关键字不在任何函数中，那么他也指向window对象。

由于在JavaScript中不明确区分函数与方法。因此有些代码看起来很奇怪：

```
//-----  
// 函数的几种可能调用形式  
//-----  
function foo() {  
    // 下面的this指代调用该方法的对象实例  
    if (this==window) {  
        document.write('call a function.', '<BR>');  
    }  
    else {  
        document.write('call a method, by object: ', this.name, '<BR>');  
    }  
}  
  
function MyObject(name) {  
    // 下面的this指代new关键字新创建实例  
    this.name = name;  
    this.foo = foo;  
}  
  
var obj1 = new MyObject('obj1');  
var obj2 = new MyObject('obj2');  
  
// 测试1: 作为函数调用  
foo();  
  
// 测试2: 作为对象方法的调用  
obj1.foo();  
obj2.foo();  
  
// 测试3: 将函数作为“指定对象的”方法调用  
foo.call(obj1);  
foo.apply(obj2);
```

在上面的代码里，obj1/obj2对foo()的调用是很普通的调用方法。——也就是在构造器上，将一个函数指定为对象的方法。

而测试3中的call()与apply()就比较特殊。

在这个测试中，foo()仍然作为普通函数来调用，只是JavaScript的语言特性允许在call()/apply()时，传入一个对象实例来指定foo()的上下文环境中所出现的this关键字的引用。——需要注意的是，此时的foo()仍旧是一个普通函数调用，而不是对象方法调用。

与this“指示调用该方法的对象实例”有些类同的，with()语法也用于限定“在一段代码片段中默认使用对象实例”。——如果不使用with()语法，那

么这段代码将受到更外层with()语句的影响；如果没有更外层的with()，那么这段代码的“默认使用的对象实例”将是window。

然而需要注意的是this与with关键字不是互为影响的。如下面的代码：

```
//-----  
// 测试: this与with关键字不是互为影响的  
//-----  
function test() {  
    with (obj2) {  
        this.value = 8;  
    }  
}  
var obj2 = new Object();  
obj2.value = 10;  
  
test();  
document.writeln('obj2.value: ', obj2.value, '<br>');  
document.writeln('window.value: ', window.value, '<br>');
```

你不能指望这样的代码在调用结束后，会使obj2.value属性置值为8。这几行代码的结果是：window对象多了一个value属性，并且值为8。

with(obj){...}这个语法，只能限定对obj的既有属性的读取，而不能主动的声明它。一旦with()里的对象没有指定的属性，或者with()限定了一个不是对象的数据，那么结果会产生一个异常。

4). 使用in关键字的运算

除了用for..in来反射对象的成员信息之外，JavaScript中也允许直接用in关键字去检测对象是否有指定名字的属性。

in关键字经常被提及的原因并不是它检测属性是否存在的能力，因此在早期的代码中，很多可喜欢用“if (!obj.propName) {}”这样的方式来检测propName是否是有效的属性。——很多时候，检测有效性比检测“是否存有该属性”更有实用性。因此这种情况下，in只是一个可选的、官方的方案。

in关键字的重要应用是高速字符串检索。尤其是在只需要判定“字符串是否存在”的情况下。例如10万个字符串，如果存储在数组中，那么检索效率将会极差。

```
//-----  
// 使用对象来检索  
//-----  
function arrayToObject(arr) {  
    for (var obj=new Object(), i=0, imax=arr.length; i<imax; i++) {  
        obj[arr[i]]=null;  
    }  
    return obj;  
}  
  
var  
arr = ['abc', 'def', 'ghi']; // more and more...  
obj = arrayToObject(arr);  
  
function valueInArray(v) {  
    for (var i=0, imax=arr.length; i<imax; i++) {  
        if (arr[i]==v) return true;  
    }  
  
    return false;  
}
```

```
function valueInObject(v) {
  return v in obj;
}
```

这种使用关键字in的方法，也存在一些限制。例如只能查找字符串，而数组元素可以是任意值。另外，arrayToObject()也存在一些开销，这使得它不适合于频繁变动的查找集。最后，(我想你可能已经注意到了)使用对象来查找的时候并不能准确定位到查找数据，而数组中可以指向结果的下标。

八、JavaScript面向对象的支持

~~~~~

(续)

### 2. JavaScript面向对象的支持

-----

(续)

#### 5). 使用instanceof关键字的运算

在JavaScript中提供了instanceof关键字来检测实例的类型。这在前面讨论它的“五重身份”时已经讲过。但instanceof的问题是，它总是列举整个原型链以检测类型(关于原型继承的原理在“构造与析构”小节讲述)，如：

```
//-----
// instanceof使用中的问题
//-----
function MyObject() {
// ...
}

function MyObject2() {
// ...
}
MyObject2.prototype = new MyObject();

obj1 = new MyObject();
obj2 = new MyObject2();

document.writeln(obj1 instanceof MyObject, '<BR>');
document.writeln(obj2 instanceof MyObject, '<BR>');
```

我们看到，obj1与obj2都是MyObject的实例，但他们是不同的构造函数产生的。——注意，这在面向对象理论中正确的：因为obj2是MyObject的子类实例，因此它具有与obj1相同的特性。在应用中这是obj2的多态性的体现之一。

但是，即便如此，我们也必须面临这样的问题：如何知道obj2与obj1是否是相同类型的实例呢？——也就是说，连构造器都相同？

instanceof关键字不提供这样的机制。一个提供实现这种检测的能力的，是Object.constructor属性。——但请先记住，它的使用远比你想象的要难。

好的，问题先到这里。constructor属性已经涉及到“构造与析构”的问题，这个我们后面再讲。“原型继承”、“构造与析构”是JavaScript的OOP中的主要问题、核心问题，以及“致命问题”。

#### 6). null与undefined

-----  
在JavaScript中，null与undefined曾一度使我迷惑。下面的文字，有利于你更清晰的认知它(或者让你更迷惑)：

- null是关键字；undefined是Global对象的一个属性。
- null是对象(空对象，没有任何属性和方法)；undefined是undefined类型的值。试试下面的代码：

```
document.writeln(typeof null);
document.writeln(typeof undefined);
- 对象模型中，所有的对象都是Object或其子类的实例，但null对象例外：
document.writeln(null instanceof Object);
- null “等值(==)” 于undefined，但不“全等值(==)” 于undefined：
document.writeln(null == undefined);
document.writeln(null === undefined);
- 运算时null与undefined都可以被类型转换为false，但不等值于false：
document.writeln(!null, !undefined);
document.writeln(null==false);
document.writeln(undefined==false);
```

## 八、JavaScript面向对象的支持

~~~~~

(续)

3. 构造、析构与原型问题

我们已经知道一个对象是需要通过构造器函数来产生的。我们先记住几点：

- 构造器是一个普通的函数
- 原型是一个对象实例
- 构造器有原型属性，对象实例没有
- (如果正常地实现继承模型，)对象实例的constructor属性指向构造器
- 从三、四条推出：obj.constructor.prototype指向该对象的原型

好，我们接下来分析一个例子，来说明JavaScript的“继承原型”声明，以及构造过程。

```
//-----
// 理解原型、构造、继承的示例
//-----
function MyObject() {
  this.v1 = 'abc';
}

function MyObject2() {
  this.v2 = 'def';
}
MyObject2.prototype = new MyObject();

var obj1 = new MyObject();
var obj2 = new MyObject2();
```

1). new()关键字的形式化代码

我们先来看“obj1 = new MyObject()”这行代码中的这个new关键字。

new关键字用于产生一个新的实例（说到这里补充一下，我习惯于把保留字叫关键字。另外，在JavaScript中new关键字同时也是一个运算符），这个实例的缺省属性中，(至少)会执有构造器函数的原型属性(prototype)的一个引用(在ECMA Javascript规范中，对象的这个属性名定义为__proto__)。

每一个函数，无论它是否用作构造器，都会有一个独一无二的原型对象(prototype)。对于JavaScript“内置对象的构造器”来说，它指向内部的一个原型。缺省时JavaScript构造出一个“空的初始对象实例(不是null)”并使原型引用指向它。然而如果你给函数的这个prototype赋一个新的对象，那么新的对象实例将执有它的一个引用。

接下来，构造过程将调用MyObject()来完成初始化。——注意，这里只是“初始化”。

为了清楚地解释这个过程，我用代码形式化地描述一下这个过程：

```

// new()关键字的形式化代码
//-----
function new(aFunction) {
// 基本对象实例
var _this = {};

// 原型引用
var _proto= aFunction.prototype;

/* if compat ECMA Script
_this.__proto__ = _proto;
*/
// 为存取原型中的属性添加(内部的)getter
_this._js_GetAttributes= function(name) {
if (_existAttribute.call(this, name))
return this[name]
else if (_js_LookupProperty.call(_proto, name))
retrun OBJ_GET_ATTRIBUTES.call(_proto, name)
else
return undefined;
}

// 为存取原型中的属性添加(内部的)setter
_this._js_SetAttributes = function(name, value) {
if (_existAttribute.call(this, name))
this[name] = value
else if (OBJ_GET_ATTRIBUTES.call(_proto, name) !== value) {
this[name] = value // 创建当前实例的新成员
}
}

// 调用构造函数完成初始化, (如果有,)传入args
aFunction.call(_this);

// 返回对象
return _this;
}

```

所以我们看到以下两点：

- 构造函数(aFunction)本身只是对传入的this实例做“初始化”处理，而不是构造一个对象实例。
- 构造的过程实际发生在new()关键字/运算符的内部。

而且，构造函数(aFunction)本身并不需要操作prototype，也不需要回传this。

2). 由用户代码维护的原型(prototype)链

接下来我们更深入的讨论原型链与构造过程的问题。这就是：

- 原型链是用户代码创建的，new()关键字并不协助维护原型链

以Delphi代码为例，我们在声明继承关系的时候，可以用这样的代码：

```

//-----
// delphi中使用的“类”类型声明
//-----
type
TAnimal = class(TObject); // 动物
TMammal = class(TAnimal); // 哺乳动物
TCanine = class(TMammal); // 犬科的哺乳动物
TDog = class(TCanine); // 狗

```

这时，Delphi的编译器会通过编译技术来维护一个继承关系链表。我们可以通过类似以下的代码来查询这个链表：

```
//-----
// delphi中使用继关系链表的关键代码
//-----
function isAnimal(obj: TObject): boolean;
begin
Result := obj is TAnimal;
end;

var
dog := TDog;

// ...
dog := TDog.Create();
writeln(isAnimal(dog));
```

可以看到，在Delphi的用户代码中，不需要直接维护继承关系的链表。这是因为Delphi是强类型语言，在处理用class()关键字声明类型时，delphi的编译器已经为用户构造了这个继承关系链。——注意，这个过程是声明，而不是执行代码。

而在JavaScript中，如果需要获知对象“是否是某个基类的子类对象”，那么你需要手工的来维护(与delphi这个例子类似的)一个链表。当然，这个链表不叫类型继承树，而叫“(对象的)原型链表”。——在JS中，没有“类”类型。

参考前面的JS和Delphi代码，一个类同的例子是这样：

```
//-----
// JS中“原型链表”的关键代码
//-----
// 1. 构造器
function Animal() {};
function Mammal() {};
function Canine() {};
function Dog() {};

// 2. 原型链表
Mammal.prototype = new Animal();
Canine.prototype = new Mammal();
Dog.prototype = new Canine();

// 3. 示例函数
function isAnimal(obj) {
return obj instanceof Animal;
}

var
dog = new Dog();
document.writeln(isAnimal(dog));
```

可以看到，在JS的用户代码中，“原型链表”的构建方法是一行代码：“当前类的构造器函数”.prototype = “直接父类的实例”

这与Delphi一类的语言不同：维护原型链的实质是在执行代码，而非声明。

那么，“是执行而非声明”到底有什么意义呢？

JavaScript是会有编译过程的。这个过程主要处理的是“语法检错”、“语句声明”和“条件编译指令”。而这里的“语句声明”，主要处理的就是函数声明。——这也是我说“函数是第一类的，而对象不是”的一个原因。

如下例：

```
//-----  
// 函数声明与执行语句的关系(firefox 兼容)  
//-----  
// 1. 输出1234  
testFoo(1234);  
  
// 2. 尝试输出obj1  
// 3. 尝试输出obj2  
testFoo(obj1);  
try {  
    testFoo(obj2);  
}  
catch(e) {  
    document.writeln('Exception: ', e.description, '<BR>');  
}  
  
// 声明testFoo()  
function testFoo(v) {  
    document.writeln(v, '<BR>');  
}  
  
// 声明object  
var obj1 = {};  
obj2 = {  
    toString: function() {return 'hi, object.'}  
}  
  
// 4. 输出obj1  
// 5. 输出obj2  
testFoo(obj1);  
testFoo(obj2);
```

这个示例代码在JS环境中执行的结果是：

```
1234  
undefined  
Exception: 'obj2' 未定义  
[object Object]  
hi, obj
```

问题是，`testFoo()`是在它被声明之前被执行的；而同样用“直接声明”的形式定义的`object`变量，却不能在声明之前引用。——例子中，第二、三个输入是不正确的。

函数可以在声明之前引用，而其它类型的数值必须在声明之后才能被使用。这说明“声明”与“执行期引用”在JavaScript中是两个过程。

另外我们也可以发现，使用`"var"`来声明的时候，编译器会先确认有该变量存在，但变量的值会是`"undefined"`。——因此`"testFoo(obj1)"`不会发生异常。但是，只有等到关于`obj1`的赋值语句被执行过，才会有正常的输出。请对照第二、三与第四、五行输出的差异。

由于JavaScript对原型链的维护是“执行”而不是“声明”，这说明“原型链是由用户代码来维护的，而不是编译器维护的。

由这个推论，我们来看下面这个例子：

```
//-----  
// 示例：错误的原型链  
//-----  
// 1. 构造器
```

```

function Animal() {} // 动物
function Mammal() {} // 哺乳动物
function Canine() {} // 犬科的哺乳动物

// 2. 构造原型链
var instance = new Mammal();
Mammal.prototype = new Animal();
Canine.prototype = instance;

// 3. 测试输出
var obj = new Canine();
document.writeln(obj instanceof Animal);

```

这个输出结果，使我们看到一个错误的原型链导致的结果“犬科的哺乳动物‘不是一种动物’”。

根源在于“2. 构造原型链”下面的几行代码是解释执行的，而不是像var和function那样是“声明”并在编译期被理解的。解决问题的方法是修改那三行代码，使得它的“执行过程”符合逻辑：

```

//-----
// 上例的修正代码(部分)
//-----
// 2. 构造原型链
Mammal.prototype = new Animal();
var instance = new Mammal();
Canine.prototype = instance;

```

3). 原型实例是如何被构造过程使用的

仍以Delphi为例。构造过程中，delphi中会首先创建一个指定实例大小的“空的对象”，然后逐一给属性赋值，以及调用构造过程中的方法、触发事件等。

JavaScript中的new()关键字中隐含的构造过程，与Delphi的构造过程并不完全一致。但在构造器函数中发生的行为却与上述的类似：

```

//-----
// JS中的构造过程(形式代码)
//-----
function MyObject2() {
  this.prop = 3;
  this.method = a_method_function;

  if (you_want) {
    this.method();
    this.fire_OnCreate();
  }
}
MyObject2.prototype = new MyObject(); // MyObject()的声明略

var obj = new MyObject2();

```

如果以单个类为参考对象的，这个构造过程中JavaScript可以拥有与Delphi一样丰富的行为。然而，由于Delphi中的构造过程是“动态的”，因此事实上Delphi还会调用父类(MyObject)的构造过程，以及触发父类的OnCreate()事件。

JavaScript没有这样的特性。父类的构造过程仅仅发生在为原型prototype属性赋值的那一行代码上。其后，无论有多少个new MyObject2()发生，MyObject()这个构造器都不会被使用。——这也意味着：
 - 构造过程中，原型对象是一次性生成的；新对象只持有这个原型实例的引用（并用“写复制”的机制来存取其属性），而并不再调用原型的构造器。

由于不再调用父类的构造器，因此Delphi中的一些特性无法在JavaScript中实现。这主要影响到构造阶段的一些事件和行为。——无法把一些“对象构造过程中”的代码写到父类的构造器中。因为无论子类构造多少次，这次对象的构造过程根本不会激活父类构造器中的代码。

JavaScript中属性的存取是动态的，因为对象存取父类属性依赖于原型链表，构造过程却是静态的，并不访问父类的构造器；而在Delphi等一些编译型语言中，(不使用读写器的)属性的存取是静态的，而对象的构造过程则动态地调用父类的构造函数。所以再一次请大家看清楚new()关键字的形式代码中的这一行：

```
//-----  
// new()关键字的形式化代码  
//-----  
function new(aFunction) {  
    // 原型引用  
    var _proto = aFunction.prototype;  
  
    // ...  
}
```

这个过程中，JavaScript做的是“get a prototype_Ref”，而Delphi等其它语言做的是“Inherited Create()”。

八、JavaScript面向对象的支持

~~~~~

(续)

### 4). 需要用户维护的另一个属性：constructor

-----

回顾前面的内容，我们提到过：

- (如果正常地实现继承模型，)对象实例的constructor属性指向构造器
- obj.constructor.prototype指向该对象的原型
- 通过Object.constructor属性，可以检测obj2与obj1是否是相同类型的实例

与原型链要通过用户代码来维护prototype属性一样，实例的构造器属性constructor也需要用户代码维护。

对于JavaScript的内置对象来说，constructor属性指向内置的构造器函数。如：

```
//-----  
// 内置对象实例的constructor属性  
//-----  
var _object_types = {  
    'function' : Function,  
    'boolean' : Boolean,  
    'regexp' : RegExp,  
    // 'math' : Math,  
    // 'debug' : Debug,  
    // 'image' : Image;  
    // 'undef' : undefined,  
    // 'dom' : undefined,  
    // 'activex' : undefined,  
    'vbaray' : VBAArray,  
    'array' : Array,  
    'string' : String,  
    'date' : Date,  
    'error' : Error,  
    'enumerator' : Enumerator,  
    'number' : Number,  
    'object' : Object  
}  
  
function objectTypes(obj) {
```

```

if (typeof obj !== 'object') return typeof obj;
if (obj === null) return 'null';

for (var i in _object_types) {
if (obj.constructor===_object_types[i]) return i;
}
return 'unknow';
}

// 测试数据和相关代码
function MyObject() {
}
function MyObject2() {
}
MyObject2.prototype = new MyObject();

window.execScript(""+
'Function CreateVBArray()' +
'Dim a(2, 2)' +
'CreateVBArray = a' +
'End Function', 'VBScript');

document.writeln('<div id=dom style="display:none">dom<', '/div>');

// 测试代码
var ax = new ActiveXObject("Microsoft.XMLHTTP");
var dom = document.getElementById('dom');
var vba = new VBArray(CreateVBArray());
var obj = new MyObject();
var obj2 = new MyObject2();

document.writeln(objectTypes(vba), '<br>');
document.writeln(objectTypes(ax), '<br>');
document.writeln(objectTypes(obj), '<br>');
document.writeln(objectTypes(obj2), '<br>');
document.writeln(objectTypes(dom), '<br>');

```

在这个例子中，我们发现constructor属性被实现得并不完整。对于DOM对象、ActiveX对象来说这个属性都没有正确的返回。

确切的说，DOM（包括Image）对象与ActiveX对象都不是标准JavaScript的对象体系中的，因此它们也可能会具有自己的constructor属性，并有着与JavaScript不同的解释。因此，JavaScript中不维护它们的constructor属性，是具有一定的合理性的。

另外的一些单体对象（而非构造器），也不具有constructor属性，例如“Math”和“Debug”、“Global”和“RegExp对象”。他们是JavaScript内部构造的，不应该公开构造的细节。

我们也发现实例obj的constructor指向function MyObject()。这说明JavaScript维护了对象的constructor属性。——这与一些人想象的不一样。

然而再接下来，我们发现MyObject2()的实例obj2的constructor仍然指向function MyObject()。尽管这很说不通，然而现实的确如此。——这到底是为什么呢？

事实上，仅下面的代码：

```

-----
function MyObject2() {
}

obj2 = new MyObject2();
document.writeln(MyObject2.prototype.constructor === MyObject2);
-----
```

构造的obj2.constructor将正确的指向function MyObject2()。事实上，我们也会注意到这种情况下，MyObject2的原型属性的constructor也正确的指向该函数。然而，由于JavaScript要求指定prototype对象来构造原型链：

```
-----  
function MyObject2() {  
}  
MyObject2.prototype = new MyObject();  
  
obj2 = new MyObject2();  
-----
```

这时，再访问obj2，将会得到新的原型(也就是MyObject2.prototype)的constructor属性。因此，一切很明了：原型的属性影响到构造过程对对象的constructor的初始设定。

作为一种补充的解决问题的手段，JavaScript开发规范中说“need to remember to reset the constructor property”，要求用户自行设定该属性。

所以你会看到更规范的JavaScript代码要求这样书写：

```
//-----  
// 维护constructor属性的规范代码  
//-----  
function MyObject2() {  
}  
MyObject2.prototype = new MyObject();  
MyObject2.prototype.constructor = MyObject2;  
  
obj2 = new MyObject2();  
-----
```

更外一种解决问题的方法，是在function MyObject()中去重置该值。当然，这样会使得执行效率稍低一点点：

```
//-----  
// 维护constructor属性的第二种方式  
//-----  
function MyObject2() {  
    this.constructor = arguments.callee;  
    // or, this.constructor = MyObject2;  
  
    // ...  
}  
MyObject2.prototype = new MyObject();  
  
obj2 = new MyObject2();  
-----
```

## 5). 析构问题

JavaScript中没有析构函数，但却有“对象析构”的问题。也就是说，尽管我们不知道一个对象什么时候会被析构，也不能截获它的析构过程并处理一些事务。然而，在一些不多见的时候，我们会遇到“要求一个对象立即析构”的问题。

问题大多数的时候出现在对ActiveX Object的处理上。因为我们可能在JavaScript里创建了一个ActiveX Object，在做完一些处理之后，我们又需要再创建一个。而如果原来的对象供应者(Server)不允许创建多个实例，那么我们就需要在JavaScript中确保先前的实例是已经被释放过了。接下来，即使Server允许创建多个实例，而在多个实例间允许共享数据(例如OS的授权，或者资源、文件的锁)，那么我们在新实例中的操作就可能会出问题。

可能还是有人不明白我们在说什么，那么我就举一个例子：如果创建一个Excel对象，打开文件A，然后我们save它，然后关闭这个实例。然后我们再创建Excel对象并打开同一文件。——注意这时JavaScript可能还没有来得及析构前一个对象。——这时我们再想Save这个文件，就发现失败了。下面的代码示例这种情况：

```
-----  
// JavaScript中的析构问题(ActiveX Object示例)  
-----
```

```

//-----
<script>
var strSaveLocation = 'file:///E:/1.xls'

function createXLS() {
var excel = new ActiveXObject("Excel.Application");
var wk = excel.Workbooks.Add();
wk.SaveAs(strSaveLocation);
wk.Saved = true;

excel.Quit();
}

function writeXLS() {
var excel = new ActiveXObject("Excel.Application");
var wk = excel.Workbooks.Open(strSaveLocation);
var sheet = wk.Worksheets(1);
sheet.Cells(1, 1).Value = '测试字符串';
wk.SaveAs(strSaveLocation);
wk.Saved = true;

excel.Quit();
}
</script>
<body>
<button onclick="createXLS()">创建</button>
<button onclick="writeXLS()">重写</button>
</body>

```

在这个例子中，在本地文件操作时并不会出现异常。——最多只是有一些内存垃圾而已。然而，如果strSaveLocation是一个远程的URL，这时本地将会保存一个文件存取权限的凭证，而且同时只能一个(远程的)实例来开启该excel文档并存储。于是如果反复点击“重写”按钮，就会出现异常。

——注意，这是在SPS中操作共享文件时的一个实例的简化代码。因此，它并非“学术的”无聊讨论，而且工程中的实际问题。

解决这个问题的方法很复杂。它涉及到两个问题：

- 本地凭证的释放
- ActiveX Object实例的释放

下面我们先从JavaScript中对象的“失效”问题说起。简单的说：

- 一个对象在其生存的上下文环境之外，即会失效。
- 一个全局的对象在没有被执用(引用)的情况下，即会失效。

例如：

```

//-----
// JavaScript对象何时失效
//-----
function testObject() {
var _obj1 = new Object();
}

function testObject2() {
var _obj2 = new Object();
return _obj2;
}

// 示例1
testObject();

```

```
// 示例2  
testObject2()  
  
// 示例3  
var obj3 = testObject2();  
obj3 = null;  
  
// 示例4  
var obj4 = testObject2();  
var arr = [obj4];  
obj3 = null;  
arr = [];
```

在这四个示例中：

- “示例1” 在函数testObject()中构造了\_obj1，但是在函数退出时，它就已经离开了函数的上下文环境，因此\_obj1失效了；
- “示例2” 中，testObject2()中也构造了一个对象\_obj2并传出，因此对象有了“函数外”的上下文环境(和生存周期)，然而由于函数的返回值没有被其它变量“持有”，因此\_obj2也立即失效了；
- “示例3” 中，testObject2()构造的\_obj2被外部的变量obj3持用了，这时，直到“obj3=null”这行代码生效时，\_obj2才会因为引用关系消失而失效。
- 与示例3相同的原因，“示例4”中的\_obj2会在“arr=[]”这行代码之后才会失效。

但是，对象的“失效”并不等会“释放”。在JavaScript运行环境的内部，没有任何方式来确切地告诉用户“对象什么时候会释放”。这依赖于JavaScript的内存回收机制。——这种策略与.NET中的回收机制是类同的。

在前面的Excel操作示例代码中，对象的所有者，也就是"EXCEL.EXE"这个进程只能在“ActiveX Object实例的释放”之后才会发生。而文件的锁，以及操作系统的权限凭证是与进程相关的。因此如果对象仅是“失效”而不是“释放”，那么其它进程处理文件和引用操作系统的权限凭据时就会出问题。

——有些人说这是JavaScript或者COM机制的BUG。其实不是，这是O S、I E和JavaScript之间的一种复杂关系所导致的，而非独立的问题。

Microsoft公开了解决这种问题的策略：主动调用内存回收过程。

在(微软的)JScript中提供了一个CollectGarbage()过程(通常简称GC过程)，GC过程用于清理当前IE中的“失效的对象实例”，也就是调用对象的析构过程。

在上例中调用GC过程的代码是：

```
-----  
// 处理ActiveX Object时，GC过程的标准调用方式  
-----  
function writeXLS() {  
  //(...)  
  
  excel.Quit();  
  excel = null;  
  setTimeout(CollectGarbage, 1);  
}
```

第一行代码调用excel.Quit()方法来使得excel进程中止并退出，这时由于JavaScript环境执有excel对象实例，因此excel进程并不实际中止。

第二行代码使excel为null，以清除对象引用，从而使对象“失效”。然而由于对象仍旧在函数上下文环境中，因此如果直接调用GC过程，对象仍然不会被清理。

第三行代码使用setTimeout()来调用CollectGarbage函数，时间间隔设为'1'，只

是使得GC过程发生在writeXLS()函数执行完之后。这样excel对象就满足了“能被GC清理”的两个条件：没有引用和离开上下文环境。

GC过程的使用，在使用了ActiveX Object的JS环境中很有效。一些潜在的ActiveXObject包括XML、VML、OWC(Office Web Component)、flash，甚至包括在JS中的VBArrary。从这一点来看，ajax架构由于采用了XMLHTTP，并且同时要满足“不切换页面”的特性，因此在适当的时候主动调用GC过程，会得到更好的效率用UI体验。

事实上，即使使用GC过程，前面提到的excel问题仍然不会被完全解决。因为IE还缓存了权限凭据。使页的权限凭据被更新的唯一方法，只能是“切换到新的页面”，因此事实上在前面提到的那个SPS项目中，我采用的方法并不是GC，而是下面这一段代码：

```
//-----
// 处理ActiveX Object时采用的页面切换代码
//-----
function writeXLS() {
//(略...)

excel.Quit();
excel = null;

// 下面代码用于解决IE call Excel的一个BUG, MSDN中提供的方法:
// setTimeout(CollectGarbage, 1);
// 由于不能清除(或同步)网页的受信任状态, 所以将导致SaveAs()等方法在
// 下次调用时无效.
location.reload();
}
```

最后之最后，关于GC的一个补充说明：在IE窗体被最小化时，IE将会主动调用一次CollectGarbage()函数。这使得IE窗口在最小化之后，内存占用会有明显改善。

## 八、JavaScript面向对象的支持

~~~~~

(续)

4. 实例和实例引用

在.NET Framework对CTS(Common Type System)约定“一切都是对象”，并分为“值类型”和“引用类型”两种。其中“值类型”的对象在转换成“引用类型”数据的过程中，需要进行一个“装箱”和“拆箱”的过程。

在JavaScript也有同样的问题。我们看到的typeof关键字，返回以下六种数据类型：“number”、“string”、“boolean”、“object”、“function”和“undefined”。

我们也发现JavaScript的对象系统中，有String、Number、Function、Boolean这四种对象构造器。那么，我们的问题是：如果有一个数字A，typeof(A)的结果，到底会是'number'呢，还是一个构造器指向function Number()的对象呢？

```
//-----
// 关于JavaScript的类型的测试代码
//-----
function getInfo(V) {
return (typeof V == 'object') ? 'Object, construct by '+V.constructor
:'Value, type of '+typeof V);
}

var A1 = 100;
var A2 = new Number(100);

document.writeln('A1 is ', getInfo(A1), '<BR>');
document.writeln('A2 is ', getInfo(A2), '<BR>');
document.writeln([A1.constructor === A2.constructor, A2.constructor === Number]);
```

测试代码的执行结果如下：

```
-----  
A1 is Value, type of number  
A2 is Object, construct by function Number() { [native code] }  
true,true  
-----
```

我们注意到，A1和A2的构造器都指向Number。这意味着通过constructor属性来识别对象，(有时)比typeof更加有效。因为“值类型数据” A1作为一个对象来看待时，与A2有完全相同的特性。

——除了与实例引用有关的问题。

参考JScript手册，我们对其它基础类型和构造器做相同考察，可以发现：

- 基础类型中的undefined、number、boolean和string，是“值类型”变量
- 基础类型中的array、function和object，是“引用类型”变量
- 使用new()方法构造出对象，是“引用类型”变量

下面的代码说明“值类型”与“引用类型”之间的区别：

```
-----  
//  
// 关于JavaScript类型系统中的值/引用问题  
//  
var str1 = 'abcdefg', str2 = 'abcdefg';  
var obj1 = new String('abcdefg'), obj2 = new String('abcdefg');  
  
document.writeln([str1==str2, str1==str2], '<br>');  
document.writeln([obj1==obj2, obj1==obj2]);
```

测试代码的执行结果如下：

```
-----  
true, true  
false, false  
-----
```

我们看到，无论是等值运算(==)，还是全等运算(==)，对“对象”和“值”的理解都是不一样的。

更进一步的理解这种现象，我们知道：

- 运算结果为值类型，或变量为值类型时，等值(或全等)比较可以得到预想结果
- (即使包含相同的数据，)不同的对象实例之间是不等值(或全等)的
- 同一个对象的不同引用之间，是等值(==)且全等(==)的

但对于String类型，有一点补充：根据JScript的描述，两个字符串比较时，只要有一个是值类型，则按值比较。这意味着在上面的例子中，代码“str1==obj1”会得到结果true。而全等(==)运算需要检测变量类型的一致性，因此“str1==obj1”的结果返回false。

JavaScript中的函数参数总是传入值参，引用类型(的实例)是作为指针值传入的。因此函数可以随意重写入口变量，而不用担心外部变量被修改。但是，需要留意传入的引用类型的变量，因为对它方法调用和属性读写可能会影响到实例本身。
——但，也可以通过引用类型的参数来传出数据。

最后补充说明一下，值类型比较会逐字节检测对象实例中的数据，效率低但准确性高；而引用类型只检测实例指针和数据类型，因此效率高而准确性低。如果你需要检测两个引用类型是否真的包含相同的数据，可能你需要尝试把它转换成“字符串值”再来比较。

6. 函数的上下文环境

只要写过代码，你应该知道变量是有“全局变量”和“局部变量”之分的。绝大多数的JavaScript程序员也知道下面这些概念：

```
-----  
//  
// JavaScript中的全局变量与局部变量  
//  
var v1 = '全局变量-1';  
v2 = '全局变量-2';
```

```
function foo() {  
v3 = '全局变量-3';  
  
var v4 = '只有在函数内部并使用var定义的，才是局部变量';  
}
```

按照通常对语言的理解来说，不同的代码调用函数，都会拥有一套独立的局部变量。
因此下面这段代码很容易理解：

```
//-----  
// JavaScript的局部变量  
//-----  
function MyObject() {  
var o = new Object;  
  
this.getValue = function() {  
return o;  
}  
}  
  
var obj1 = new MyObject();  
var obj2 = new MyObject();  
document.writeln(obj1.getValue() == obj2.getValue());
```

结果显示false，表明不同(实例的方法)调用返回的局部变量“obj1/obj2”是不相同。

变量的局部、全局特性与OOP的封装性中的“私有(private)”、“公开(public)”具有类同性。因此绝大多数资料总是以下面的方式来说明JavaScript的面向对象系统中的“封装权限级别”问题：

```
//-----  
// JavaScript中OOP封装性  
//-----  
function MyObject() {  
// 1. 私有成员和方法  
var private_prop = 0;  
var private_method_1 = function() {  
// ...  
return 1  
}  
function private_method_2() {  
// ...  
return 1  
}  
  
// 2. 特权方法  
this.privileged_method = function () {  
private_prop++;  
return private_prop + private_method_1() + private_method_2();  
}  
  
// 3. 公开成员和方法  
this.public_prop_1 = " ";  
this.public_method_1 = function () {  
// ...  
}  
}  
  
// 4. 公开成员和方法(2)  
MyObject.prototype.public_prop_1 = " ";  
MyObject.prototype.public_method_1 = function () {  
// ...  
}
```

```
var obj1 = new MyObject();
var obj2 = new MyObject();

document.writeln(obj1.privileged_method(), '<br>');
document.writeln(obj2.privileged_method());
```

在这里，“私有(private)” 表明只有在(构造)函数内部可访问，而“特权(privileged)” 是特指一种存取“私有域”的“公开(public)” 方法。“公开(public)” 表明在(构造)函数外可以调用和存取。

除了上述的封装权限之外，一些文档还介绍了其它两种相关的概念：

- 原型属性：Classname.prototype.propertyName = someValue
- (类)静态属性：Classname.propertyName = someValue

然而，从面向对象的角度上来讲，上面这些概念都很难自圆其说：JavaScript究竟是为何、以及如何划分出这些封装权限和概念来的呢？

——因为我们必须注意到下面这个例子所带来的问题：

```
//-----
// JavaScript中的局部变量
//-----
function MyFoo() {
    var i;

    MyFoo.setValue = function (v) {
        i = v;
    }
    MyFoo.getValue = function () {
        return i;
    }
}
MyFoo();

var obj1 = new Object();
var obj2 = new Object();

// 测试一
MyFoo.setValue.call(obj1, 'obj1');
document.writeln(MyFoo.getValue.call(obj1), '<BR>');

// 测试二
MyFoo.setValue.call(obj2, 'obj2');
document.writeln(MyFoo.getValue.call(obj2));
document.writeln(MyFoo.getValue.call(obj1));
document.writeln(MyFoo.getValue());
```

在这个测试代码中，obj1/obj2都是Object()实例。我们使用function.call()的方式来调用setValue/getValue，使得在MyFoo()调用的过程中替换this为obj1/obj2实例。

然而我们发现“测试二”完成之后，obj2、obj1以及function MyFoo()所持有的局部变量都返回了“obj2”。——这表明三个函数使用了同一个局部变量。

由此可见，JavaScript在处理局部变量时，对“普通函数”与“构造器”是分别对待的。这种处理策略在一些JavaScript相关的资料中被解释作“面向对象中的私有域”问题。而事实上，我更愿意从源代码一级来告诉你真相：这是对象的上下文环境的问题。——只不过从表面看去，“上下文环境”的问题被转嫁到对象的封装性问题上了。

(在阅读下面的文字之前，)先做一个概念性的说明：

- 在普通函数中，上下文环境被window对象所持有
- 在“构造器和对象方法”中，上下文环境被对象实例所持有

在JavaScript的实现代码中，每次创建一个对象，解释器将为对象创建一个上下文环境链，用于存放对象在进入“构造器和对象方法”时对function()内部数据的一个备份。JavaScript保证这个对象在以后再进入“构造器和对象方法”内部时，总是持有该上下文环境，和一个与之相关的this对象。由于对象可能有多个方法，且每个方法可能又存在多层嵌套函数，因此这事实上构成了一个上下文环境的树型链表结构。而在构造器和对象方法之外，JavaScript不提供任何访问(该构造器和对象方法的)上下文环境的方法。

简而言之：

- 上下文环境与对象实例调用“构造器和对象方法”时相关，而与(普通)函数无关
- 上下文环境记录一个对象在“构造函数和对象方法”内部的私有数据
- 上下文环境采用链式结构，以记录多层的嵌套函数中的上下文

由于上下文环境只与构造函数及其内部的嵌套函数有关，重新阅读前面的代码：

```
//-----
// JavaScript中的局部变量
//-----
function MyFoo() {
var i;

MyFoo.setValue = function (v) {
i = v;
}
MyFoo.getValue = function () {
return i;
}
}
MyFoo();

var obj1 = new Object();
MyFoo.setValue.call(obj1, 'obj1');
```

我们发现setValue()的确可以访问到位于MyFoo()函数内部的“局部变量i”，但是由于setValue()方法的执行者是MyFoo对象(记住函数也是对象)，因此MyFoo对象拥有MyFoo()函数的唯一一份“上下文环境”。

接下来MyFoo.setValue.call()调用虽然为setValue()传入了新的this对象，但实际上拥有“上下文环境”的仍旧是MyFoo对象。因此我们看到无论创建多少个obj1/obj2，最终操作的都是同一个私有变量i。

全局函数/变量的“上下文环境”持有者为window，因此下面的代码说明了“为什么全局变量能被任意的对象和函数访问”：

```
//-----
// 全局函数的上下文
//-----
/*
function Window() {
*/
var global_i = 0;
var global_j = 1;

function foo_0() {
}

function foo_1() {
}
/*
}

window = new Window();
*/
```

因此我们可以看到foo_0()与foo_1()能同时访问global_i和global_j。接下来的推论是，上下文环境决定了变量的“全局”与“私有”。而不是反过来通过变量的私有与全局来讨论上下文环境问题。

更进一步的推论是：JavaScript中的全局变量与函数，本质上是window对象的私有变量与方法。而这个上下文环境块，位于所有(window对象内部的)对象实例的上下文环境链表的顶端，因此都可能访问到。

用“上下文环境”的理论，你可以顺利地解释在本小节中，有关变量的“全局 / 局部”作用域的问题，以及有关对象方法的封装权限问题。事实上，在实现JavaScript的C源代码中，这个“上下文环境”被叫做“JSContext”，并作为函数 / 方法的第一个参数传入。——如果你有兴趣，你可以从源代码中证实本小节所述的理论。

另外，《JavaScript权威指南》这本书中第4.7节也讲述了这个问题，但被叫做“变量的作用域”。然而重要的是，这本书把问题讲反了。——作者试图用“全局、局部的作用域”，来解释产生这种现象的“上下文环境”的问题。因此这个小节显得凌乱而且难以自圆其说。

不过在4.6.3小节，作者也提到了执行环境(execution context)的问题，这就与我们这里说的“上下文环境”是一致的了。然而更麻烦的是，作者又将读者引错了方法，试图用函数的上下文环境去解释DOM和ScriptEngine中的问题。

但这本书在“上下文环境链表”的查询方式上的讲述，是正确的而合理的。只是把这个叫成“作用域”有点不对，或者不妥。