

Javascript程序设计 (上)

2016.12.1

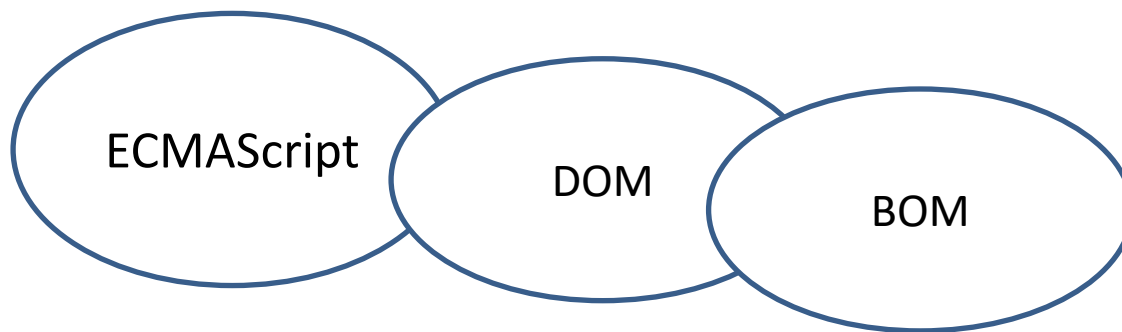
isszym sysu.edu.cn

概述

- 1995 年，Netscape Navigator 2.0 增加了一个由网景(Netscape)公司与太阳(Sun)公司开发的称为 LiveScript 的脚本语言，当时的目的是在浏览器和服务端使用它。在 Netscape Navigator 2.0 即将正式发布前，Netscape 将其更名为 JavaScript，目的是为了利用 Java 这个因特网时髦词汇。
- 因为 JavaScript 1.0 如此成功，Netscape 公司在 Netscape Navigator 3.0 中发布了 1.1 版。此时微软发布了 IE 3.0 并搭载了一个 JavaScript 的克隆版，叫做 Jscript。这样命名是为了避免与 Netscape 产生潜在的许可纠纷。当时产生了 3 种不同的 JavaScript 版本：Netscape Navigator 3.0 中的 **JavaScript**、IE 中的 **JScript** 以及 CEnv 中的 **ScriptEase**。
- 与 C 和其他编程语言不同的是，JavaScript 并没有一个标准来统一其语法或特性，而这 3 种不同的版本恰恰突出了这个问题。随着业界担心的增加，这个语言的标准化的显然已经势在必行。

[参考](#)

- 1997 年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第 39 技术委员会（[TC39](#)）被ECMA委派来“标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语义”。由来自 Netscape、Sun、微软、Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39 锤炼出了 ECMA-262，该标准定义了名为 **ECMAScript** 的全新脚本语言。
- 在接下来的几年里，国际标准化组织及国际电工委员会（ISO/IEC）也采纳 ECMAScript 作为标准（ISO/IEC-16262）。从此，Web 浏览器就开始努力（虽然有着不同的程度的成功和失败）将 ECMAScript 作为 JavaScript 实现的基础。
- 一个完整的 **JavaScript** 实现由**ECMAScript**、**DOM**和**BOM**组成的：
 - ECMAScript 描述了该语言的基本语法。
 - DOM (Document Object Model)描述了与文档对象交互的属性和方法。
 - BOM(Browser Object Model) 描述了与浏览器对象进行交互的属性和方法。



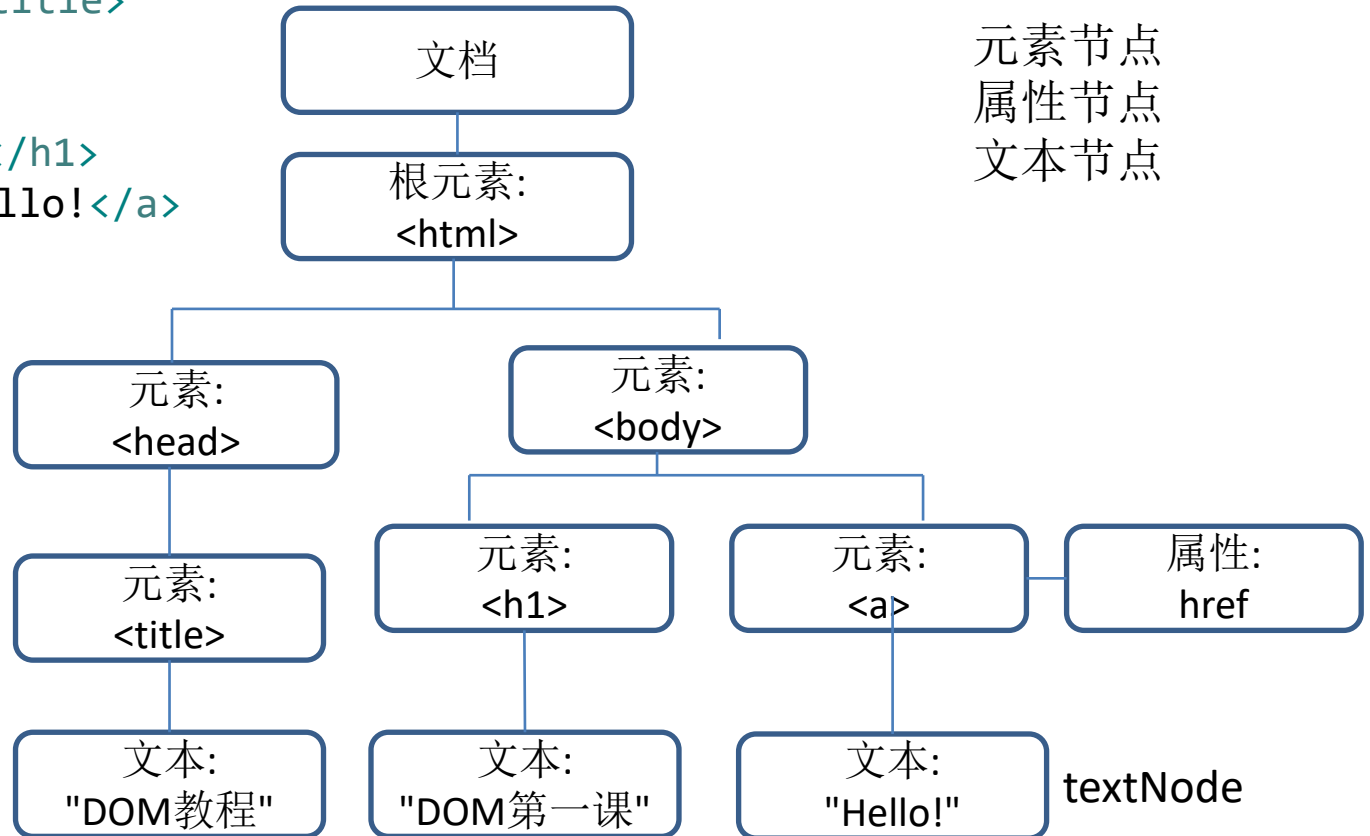
[参考1](#) [参考2](#)

文档树

(DOM节点树)

```
<html>
<head>
<title>DOM 教程</title>
</head>
<body>
  <h1>DOM 第一课</h1>
  <a href="#">Hello!</a>
</body>
</html>
```

元素节点
属性节点
文本节点



可以用JavaScript
语言来修改节点
的内容

window对象、document对象、location对象

```
<html>
<head>
<title>DOM 教程</title>
<script type="text/javascript">
    function replace(){
        var x=document.getElementById("a1");
        x.innerHTML="DOM 第二课";
    }
</script>
</head>
<body>
    <h1 id="a1">DOM 第一课</h1>
    <p onclick="replace()">Hello!</p>
</body>
</html>
```

- * document代表当前网页对象：
document.write("abc")
- * Javascript的全局变量和函数
都定义在window对象中
- * Javascript是解释执行的

点击后会替换元素h1的内容

<script>可以定义多个，而且放在html的任何地方。它是作为一个整体，自上往下执行，上面定义的变量，下面依然可以可以使用。引用外部文件：

```
<script type="text/javascript" charset="utf-8" src="js/ex.js">
</script>
```

把元素<script>的内容放在外部文件里

如果替换replace函数为如下内容(ajax)，更可以不刷新整个页面而从网站获取一个页面来替换掉<h1>的内容。

```
function replace(){
    var url="http://172.18.187.11:8080/lab/js/haha.html";
    var ctrlID="a1";
    var param=null;
    var xmlhttp = new XMLHttpRequest();

    xmlhttp.onreadystatechange = function () {
        if (xmlhttp.readyState == 4)    //读取服务器响应结束
        {
            if (xmlhttp.status >= 200 && xmlhttp.status < 300
                || xmlhttp.status >= 304) {
                //alert(xmlhttp.responseText);
                var obj = document.getElementById(ctrlID)
                obj.innerHTML = xmlhttp.responseText; //可以加上textarea
            }
            else {
                alert("Request was unsuccessful:" + xmlhttp.status);
            }
        }
    }
    xmlhttp.open("get", url, true);
    xmlhttp.send(param);
}
```

HaHa,
这是Ajax的内容

JavaScript变量

- 定义

- ✓ **JavaScript**变量可以不经定义直接使用，这些变量是全局变量。不过为了确定生存期和作用域，最好对变量进行定义，这些变量为局部变量。
- ✓ **JavaScript**变量分为基本类型和引用类型。基本类型的变量直接保存值，而引用类型的变量则保存指向实际内容的指针。
- ✓ 变量名以字母、下划线(_)和美元符号(\$)开头，其它部分还可以加上数字。变量名区分大小写。

```
<script type="text/javascript">
```

```
    var x;
```

```
    var y=3;
```

```
    z="hello";
```

```
    var a=3.5, b="123456";
```

```
    b=5;
```

```
</script>
```

```
// 变量定义(未指明类型)
```

```
// 定义并初始化(数字型)
```

```
// 未定义变量(全局变量)
```

```
// 一次定义多个变量
```

```
// 改变值和类型(不推荐)
```

● 基本类型

数字型(number): 整数和小数。070(8进制), 0xFF, 100, 0.345
布尔型(boolean): true 或 false。true转化为整数1, false为0
字符串(string): 用单引号或双引号括起的单个或连续字符。
null: 空类型（只有这一个值, 未初始化对象的取值）
undefined: 未定义（只有这一个值, 未初始化变量的取值）

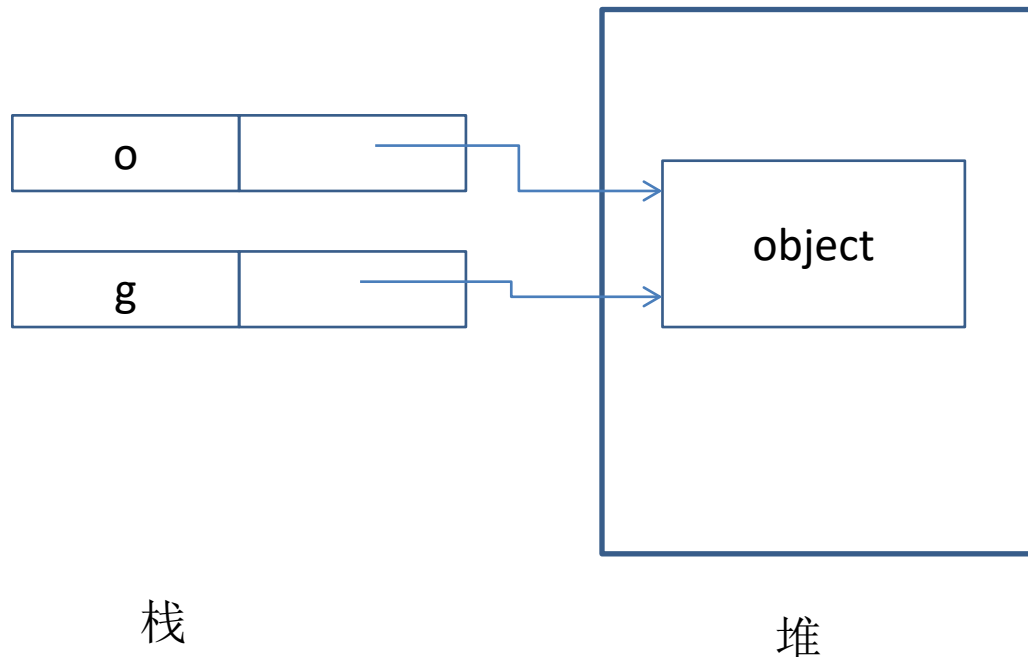
```
var i=10;           //typeof i ="number"  
var s="abcde";      //typeof s ="string"  
var b=false;        //typeof b ="boolean"  
var o=null;         //typeof o ="object" (bug, 结果应该是null)  
var t;              //typeof t ="undefined"  
                    //typeof x ="undefined"
```

* 数值类型、布尔类型、字符串类型有对应的类: Number、Boolean、String

● 引用类型

JavaScript中的对象采用引用类型，是属性的无序列表。每个对象属性都是一个键值对。

```
var o=new Object(); // typeof o = "object"  
var g=o;           // 复制指针  
var o=null;        // o不再指向任何对象。在不使用该变量时最好解除引用  
                  // 没有被引用的对象的空间可以被垃圾收集器所释放
```



• 变量的特殊取值和类型转换

| | |
|------------|--|
| undefined | 未初始化变量的取值 |
| null | 未初始化对象的取值 |
| NaN | not a number。除以0、非数值字符串等的取值 |
| false | 0、空串、null、undefined、NaN |
| true | 其它 |
| isNaN() | 判断参数是否为数值 |
| isFinite() | 数值在Number.Min_VALUE~Number.Max_VALUE之间 |
| N/A | Not Applicable |

```
var a=3;
var b="5";
var c1=a+b;           //35
var c2=b+a;           //53
var d=a+parseInt(b);  //8, 把字符串转换为整数.
var e=a+(b-0);        //8
var f=parseFloat('a'); //NaN
var g=Number('abc');  //NaN.
b=4;                  //4. 转换了类型, 不提倡
document.write(c1+" "+c2+" "+d+" "+e+" "+f+" "+g) //写入到网页
```

运算符与表达式

- 算术表达式：用算术运算符形成的表达式，计算结果为整值

```
var x = 5, y = 6, z = 10;    // 一次定义多个变量，并赋初值
var exp = (y + 3) * z + x;    // 右边为算术表达式，exp得值95
```

- 关系表达式：用关系运算符形成的表达式，计算结果为真假值

```
var x = 300;
var r1 = x > 10;    // x是否大于10。r1得值true
var r2 = x <= 100;  // x是否小于等于100。r2得值false
```

- 逻辑表达式：用逻辑运算符形成的表达式，计算结果为真假值。

```
var y = 99;
var r3 = (y > 10) && (y < 100); // y是否大于10并且小于100。 true
```

- 位表达式：按位运算的表达式，先转换整数(32位)再运算。

```
var z = 16;
var r4 = z | 0x1E0F; // 按位或    结果：0x1E1F (7711)
var r5 = r4 << 4;    // 算术左移4位 结果：0xE1F0
```

- 表达式计算

```
var x = 20, y = 100.2;
document.write(eval("x+y")); // 120.2. eval中可以写入一个表达式
```

算术运算符: $a+b$ $a-b$ $a*b$ a/b (商) $a\%b$ (余数) $\text{Math.floor}(i/j)$ (整除, i 和 j 为整数)

关系运算符: $a>b$ $a<b$ $a\geq b$ $a\leq b$ $a==b$ (等于) $a===b$ (恒等) $a!=b$ (不等于)

逻辑运算符: $a\&\&b$ (短路与) $a||b$ (短路或) $!a$ (非)

位运算: $\sim a$ (按位非) $a\&b$ (按位与) $a|b$ (按位或) a^b (按位异或)

移位运算: $b<<1$ (左移1位) $a>>2$ (带符号右移2位) $b>>>3$ (无符号右移3位)

三目运算: $x<3?10:7$ (如果 x 小于3, 则取值10, 否则, 取值7)

单目运算: $++x$ (x 先加1, 再参与运算) $--x$ $x++$ $x--$ $-x$ (变符号)

赋值运算: $x+=a$ ($x=x+a$) $x-=a$ $x*=a$ $x/=a$ $x\%=a$ $x\&=a$
 $x|=a$ $x\&=a$ $x|=a$ $x^=a$ $x>>=a$ $x>>>=a$ $x<=<=a$

运算优先级:

高 $[]()$ \rightarrow 单目 $\rightarrow * / \% \rightarrow + - \rightarrow << >> >>>$
 $\rightarrow < > <= >= \rightarrow == != \rightarrow \& \rightarrow ^$
 $\rightarrow | \rightarrow \&\& \rightarrow || \rightarrow$ 三目运算 \rightarrow 复杂赋值 低

* $==$ 会自动转换类型再判断值是否相等, $===$ 不会自动转换类型 (只要有`null`, `undefined`, `NaN`, 返回`false`)

* 移位操作为32位的算术移位。0x80000001>>4 得到 0xF8000000 (十进制 -134217728)

基本语句

- 赋值语句

```
var x;           //变量定义
x = 20;          //赋值语句，左边为变量，右边为表达式
var y = 100.2;
x = y;
z = y = 30       //单行语句的结束处可以不加分号；
alert(x+" "+y+" "+z); //100.2 30 30

// console.log(x+" "+y+" "+z)可以在浏览器调试用的控制台中显示
```

- 注释语句

```
//      注释一行
/* ..... */  注释若干行
```

• 分支控制语句

```
var age = 8;
var state;
if(age<1)
    state="婴儿";
else if(age>=1 && age<10)
    state="儿童";
else
    state="少年或青年或中年或老年";    //state="儿童"
```

*条件嵌套: else属于最靠近它的if

```
var cnt = 10;
var x;
switch(cnt){
    case 1:  x=5.0;break;
    case 12: x=30.0;break;
    default: x=100.0;
}           //x=100.0
```

- 循环控制语句

```
var sum=0;
for(var i=0;i<=100;i++){
    sum+=i;
}                                     //sum=5050
```

```
sum=0;
cnt = 0;
var scores=[100.0, 90.2, 80.0, 78.0,93.5];
for(var score in scores){ // score为下标(元素为数值)或字符串(元素为对象)
    sum+=scores[score];
    cnt++;
}
avg = sum/cnt; // avg=88.34
```

```
sum=0;
var k=0;
while(k<=10){
    sum=sum+k;
    k++;
} //55
```

Java:

```
for(double score:scores){
    sum=sum+score;
    cnt++;
}
```

```

sum=0;
k=0;
do{
    sum=sum+k;
    k++;
}while(k<=20);  // 210

```

```

/* 求距阵之和
 * 1 2 3 ... 10
 * 1 2 3 ... 10
 * .....
 * 1 2 3 ... 10
 */

```

```

sum=0;
for(var i=1;i<=10;i++){
    for(var j=1;j<=10;j++){
        sum=sum+j;
    }
}  // sum=550


```



```

sum=0;
Label1:
for(var i=1;i<=10;i++){
    for(var j=1;j<=10;j++){
        if(j==i){
            continue; //跳到for结束处继续执行
        }
        sum=sum+j; //除去对角线的矩阵之和
    }
} //sum=495


```


continue跳到这里
continue Label1跳到这里

```

sum=0;
Label2:
for(var i=1;i<=10;i++){
    for(var j=1;j<=10;j++){
        if(j==i){
            break; //跳出for循环继续执行
        }
        sum=sum+j; //下三角加对角线矩阵之和
    }
} //sum=165

```


break跳到这里
break Label2跳到这里

字符串

字符串是 JavaScript 的一种原始类型。"abc"就是一个字符串类型的常量。**JavaScript 的字符串是不可变的(immutable)**，String 类定义的方法返回的是全新的字符串，而原始字符串没有改动。**==可以直接判断内容是否相等。**

下面是常用的字符串函数：

```
//concat将两个或多个字符的文本组合起来
var a = "hello";
var b = ",world";
var c = a.concat(b); //"hello,world".与c=a+b结果相同
// indexOf返回字符串中一个子串第一处
// 出现的索引（从0开始）。如果没有
// 匹配项，则返回 -1 。lastIndexOf从后往前搜索
var index1 = a.indexOf("l");//2
var index2 = a.indexOf("l",3);//3
//charAt返回指定位置的字符。
var get_char = a.charAt(0); //"h"    charCodeAt返回该字符的unicode编码
var len = a.length;           //5
```

```
//var a = "hello";
//var b = ",world";
//match检查一个字符串匹配一个正则表达式内容，如果不匹配则返回 null。
var re = new RegExp(/^he/);
var is_alpha1 = a.match(re); // "he"
var is_alpha2 = b.match(re); // null

//substring返回字符串的一个子串，传入参数是起始位置和结束位置。
var sub_string1 = a.substring(1); // "ello"
var sub_string2 = a.substring(1,4); // "ell"

//substr返回字符串的一个子串，传入参数是起始位置和长度
var sub_string3 = a.substr(1); // "ello"
var sub_string4 = a.substr(1,4); // "ello"

//replace把匹配正则表达式的字符串替换为新配的字符串。
var result1 = a.replace(re,"Hello"); // "Hellollo"
var result2 = b.replace(re,"Hello"); // ",world"

//search查找正则表达式，如果成功，返回匹配的索引值，否则返回 -1
var index1 = a.search(re); // 0
var index2 = b.search(re); // -1
```

```
//var a = "hello";
//var b = ",world";
//split将一个字符串做成一个字符串数组。join把数组变为字符串。
var arr1 = a.split(""); // [h,e,l,l,o],可以指定间隔符,例如,""
var s2=arr1.join(","); //返回Hello

//length返回字符串的长度,即其包含的字符的个数。
len = a.length; // 5

//toLowerCase和将整个字符串转成小写字母和大写字母。
var lower_string = a.toLowerCase();//"hello"
var upper_string = a.toUpperCase();// "HELLO"

//parseInt把字符串转化为数值。数值转化为字符串:""+number
int1=parseInt("1234blue"); // 1234
int2=parseInt("0xA"); // 10
int3=parseInt("22.5") // 22
int4=parseInt("blue") // NaN

//返回链接字符串
"Free Web Tutorials!".link("http://www.w3school.com.cn");
```

* 字符串方法参见附录 [参考1](#) [参考2](#)

函数

- 定义

函数是可以反复执行(调用)的一段程序。它可以带入参数并返回执行结果。
JavaScript的函数都用作对象方法。

```
function sum(num1,num2){  
    return num1+num2;  
}
```

也可以这样定义：

```
var sum = function(num1,num2){  
    return num1+num2;  
}
```

字面量定义

Javascript中定义的每一个函数都表示为一个对象，是Function类的一个实例。
函数名为引用类型，指向该函数对象。

* 第一种函数定义是window对象的方法。函数还可以作为创建新对象的构造函数。

- JavaScript的函数就是对象

```
function sum(num1,num2){  
    return num1+num2;  
}  
alert(sum(10,10));           // 返回 20  
var asum = sum;             // 函数为对象，可以直接赋值  
alert(asum(10,10));         // 返回20  
sum = null;                 // 清除对象sum  
alert(asum(10,10));         // 返回20。asum依然可用  
asum=function(num1,num2){  
    return num1-num2;  
}  
alert(asum(10,10));         //返回0 。 没有重载，直接覆盖
```

用构造函数的方法定义函数：

```
var sayHi = new Function("sName", "sMessage",    // sName和sMessage为参数名  
                          "alert(\"Hello \" + sName + sMessage);");  
var sayHello = sayHi;  
sayHello("Zhang","come here!");
```

↑
函数体

- 要求先定义再引用

情形1：调用正确（函数定义是全局的，可以在定义前引用）

```
alert(sum(10,10));           // 20
function sum(num1,num2){
    return num1+num2;
}
```

情形2：调用错误(sum作为函数对象，必须先定义再引用)

```
alert(sum(10,10));           // 出错
sum= function(num1,num2){
    return num1+num2;
}
```

- 递归定义

Javascript的函数可以递归定义，函数对象可以作为参数

```
// 函数可以递归定义
alert(sum(inc,10));
function inc(num1){
    if(num1<=1)
        return 1;
    return num1 * inc(num1-1);
}
```

```
// 函数对象可以作为参数
function sum(inc1,num2){
    return inc1(num2)+num2;
}
```


- 可变参数

无论函数有无参数，arguments都作为函数的可变参数。

```
fucnton add(){                                     // 可变参数
    var c=0;
    for (var i=0; i<arguments.length;i++){
        var c=parseInt(arguments[i]) +c ;
    }
    alert(Array.isArray(arguments)); //false. arguments并非数组
    return c;
}
document.write("<p> no param=" + add() + "</p>");
document.write("<p> four param=" + add(1,2,3,4) + "</p>");
```

对象

- 创建对象

```
var person = new Object();      // 等同 var person ={};
person.name = "Nicholas";
person.age = 26;
person.print = function(){alert(person.name)};
```

或（字面量定义方法）

```
var person = { name: "Nicholas",
               age: 26,
               print: function(){alert(person.name)};
};
```

或

```
var person = { "name": "Nicholas",
               "age": 26,
               5: true
};
```

* Javascript没有类的概念（**ECMAScript 6**引入了类的定义）。

* **Object**是Javascript的基本类，其它对象都是它的实例。属性值也可以是对象。

- 对象访问

```
alert(person["name"]);  
alert(person.name);
```

// 属性名可以是字符串表达式

```
display({name:"Nicholas", age:26});  
function display(person){  
    if(typeof person.name=="string") alert("1");  
    if(typeof person.age=="number") alert("2");  
}
```

//可以直接把对象作为参数

```
delete person.name;
```

// 删除属性

● 属性枚举

```
var person = {name:"Nicholas", age:26};
alert("age" in person);           //true (age为自有属性)
alert("toString" in person);      //true (toString为原生属性)
alert(person.propertyIsEnumerable("age")); //true (可枚举属性)
alert(person.propertyIsEnumerable("toString")); //false
for(var prop in person){          //列举所有可枚举的属性
    document.write("name:"+prop + " &nbsp;value:"+person[prop]+"<br>");
}
var props = Object.keys(person);  //包含所有可枚举的属性名
for(var i=0,len=props.length;i<len;i++){
    document.write("name:"+props[i]
        + " &nbsp;value:"+person[props[i]]+"<br>");
}
```

结果: name:name value:Nicholas
 name:age value:26
 name:name value:Nicholas
 name:age value:26

* `toString`是一个自`Object`继承来的原生属性。原生属性默认是不可枚举的，自定义属性默认是可枚举的。

* ECMAScript 5 加入了修改属性的枚举特征，详细请见附录。

- 用工厂模式创建对象

```
function createPerson(name,age,job) {  
    var o = new Object();  
    o.name = name;  
    o.age = age;  
    o.job = job;  
    o.sayName=function(){alert(this.name);};  
    return o;  
}  
var person1=createPerson("Nicholas",29,"Software Engineer");  
var person2=createPerson("Greg",27,"Doctor");  
alert(person1.name);  
alert(person1.sayName===person2.sayName);           // false
```

* 对象方法最后"return this"可以实现链式调用: car.start().run()。

- 用构造器创建对象

```
function Person(name,age,job) {  
    this.name=name;           // this代表当前对象  
    this.age=age;  
    this.job=job;  
    this.sayName=function(){alert(this.name);};  
}  
  
var person1 = new Person("Nicholas",29,"Software Engineer");  
var person2 = new Person("Greg",27,"Doctor");  
alert(person1.name);           // Nicholas  
alert(person1.constructor === Person);    // true  
alert(person2.constructor === Person);    // true  
alert(person1 instanceof Person);         // true  
alert(person2 instanceof Person);         // true  
alert(person2 instanceof Object);         // true  
alert(person1.sayName===person2.sayName); // false  
delete person1                    // 删除对象
```

总结：定义一个函数就定义了一个对象，函数名指向该对象。函数还可以用于创建对象，此时该函数是创建新对象的构造器。

下面的Person是window对象的一个方法，直接执行它会让其中属性和方法成为window的属性和方法：

```
function Person(name,age,job){  
    this.name = name;  
    this.age = age;  
    this.job = job;  
    this.sayName = function(){  
        alert(this.name+" "+this.age+ " " + this.job);  
    }  
};
```

```
var p1=Person;  
p1("David",23,"Engineer"); //执行Person("David",23,"Engineer")效果相同  
alert(name);  
sayName(); //同this.sayName()或window.sayName()
```

无论采用工厂模式还是构造函数来创建对象，它们都有一个缺点，就是这些对象的方法不是共享同一个方法的代码，而是独立创建的，这很浪费空间。解决这个问题的方法之一是定义一个全局函数SayName()，然后让this.sayName=SayName()，但是这种做法破坏了对象的封装性。

```
function Person(name,age,job) {  
    this.name=name;           // this代表当前对象  
    this.age=age;  
    this.job=job;  
    this.sayName=SayName;  
}  
function SayName(){  
    alert(this.name);  
}
```


对原始封装类型引入新属性:

```
var a = "hello";           //原始类型（没有关联任何方法）
var s1 = a.substring(2,4); //11
a.next = "world";
alert(a.next);             // undefined
```

为什么会出现上面的情况？因为原始类型并没有关联任何方法，上面的语句实际上要引入String类型的临时变量才可以执行：

```
var a = "hello";
var temp = new String(a);    // temp为String的引用类型
var s1 = temp.sustring(2,4); //11
temp = null;

var temp = new String(a);    // temp为String的引用类型
temp.next = "world";
alert(temp.next);           // undefined
temp = null;
```

} a.substring(2,4)

} a.next="world"

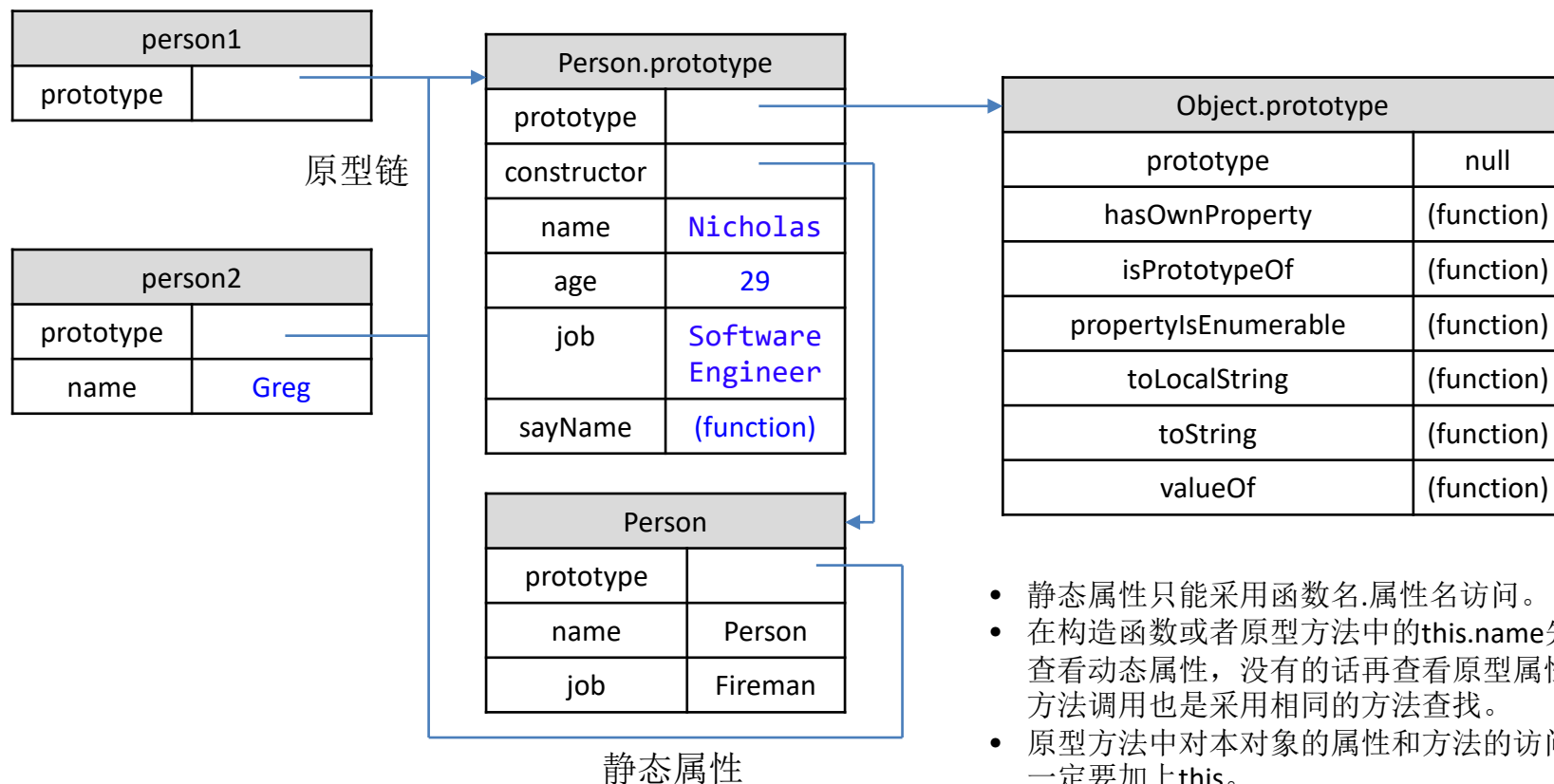
原型

每个函数（以及每个对象）都会有一个原型(**prototype**)属性。通过函数创建的对象实例会继承其原型对象上定义的属性和方法。如果对象实例中定义了同名属性和方法，它们将覆盖(**override**)原型中的属性和方法。

```
function Person() {}  
Person.prototype.name="Nicholas";           // 在函数原型上定义新属性  
Person.prototype.age=29;  
Person.prototype.job="Software Engineer";  
Person.prototype.sayName= function() {alert(this.name);};  
Person.job="Fireman";                        // 在函数对象上定义新属性  
  
var person1 = new Person();  
person1.sayName();                           // Nicholas--来自原型  
var person2 = new Person();  
person2.name = "Greg";  
person2.sayName();                           // Greg--来自实例  
alert(person1.sayName===person2.sayName); //true  
alert(Person.job);                           //Fireman  
alert(person1.job);                           //Software Engineer  
alert(Person.name);                           //Person
```

```
function Person() {}
Person.prototype.name="Nicholas";
Person.prototype.age=29; // 原型属性
Person.prototype.job="Software Engineer";
Person.prototype.sayName
    = function(){alert(this.name)};
Person.job="Fireman";
var person1 = new Person();
person1.sayName(); // Nicholas--来自原型
```

```
var person2 = new Person();
person2.name = "Greg"; // 动态属性
person2.sayName(); // Greg--来自实例
alert(person1.sayName===person2.sayName);
//true
alert(Person.job); //Fireman 静态属性
alert(person1.job); //Software Engineer
alert(Person.name); //Person
```



- 静态属性只能采用函数名.属性名访问。
- 在构造函数或者原型方法中的this.name先查看动态属性，没有的话再查看原型属性。方法调用也是采用相同的方法查找。
- 原型方法中对本对象的属性和方法的访问一定要加上this。

```

function Person() {}
Person.prototype.name="Nicholas";
Person.prototype.age=29;
Person.prototype.job="Software Engineer";
Person.prototype.sayName= function(){alert(this.name);};
Person.job="Fireman";
var person1 = new Person();
var person2 = new Person();
person2.name = "Greg";
alert(person1 instanceof Person); //true
alert(person1 instanceof Object); //true
alert(person2.hasOwnProperty("name")); //true. 自有属性(不是原型的属性)
alert(person1.hasOwnProperty("name")); //false
alert("name" in person1); //true. 是它的属性（可以来自原型）
alert("name" in person2); //true
alert(Person.prototype instanceof person1); //false
alert(Person.prototype instanceof person1); //true
alert(Person.prototype.hasOwnProperty("name")); //true
alert(person1.constructor===Person); //true
var obj
for(obj in person1){ // 取出所有属性和方法名(包括继承来的)
    alert(obj);      // toString, 显示: name age job sayName
}
alert(Person.prototype.isEnumerable("job")); //true.可枚举的
alert(person1.prototype.isEnumerable("job")); //false.只有自定义属性才可枚举

```

Javascript的对象模型没有继承性，但是可以通过原型链来实现继承关系。

```
function Person() {}  
function Parent(age) {this.age=age;}  
Parent.prototype.name="Nicholas";  
Parent.prototype.age=29;  
Parent.prototype.job="Software Engineer";  
Parent.prototype.sayName=function(){  
    alert(this.name);  
};
```

```
Person.prototype = new Parent(16);  
Person.prototype.constructor = Person;  
var person1 = new Person();  
person1.sayName();  
var person2 = new Person();  
person2.name = "Greg";  
person2.sayName();  
var person3 = new Person();  
person3.sayName();  
alert(person1.sayName===person2.sayName);  
alert(person1.constructor===Person);  
alert(person1.age);  
alert(person2.age);
```

```
// 继承（构造函数变为Parent）  
// 重新设置构造函数
```

```
// Nicholas--来自原型
```

```
// Greg--来自实例
```

```
// Nicholas--来自原型
```

```
// true
```

```
// true
```

```
// 16
```

```
// 16
```

[参考1](#)

[参考2](#)

闭包

创建一个Javascript对象将保存当时的上下文环境（可以访问的变量和函数），也就是保存创建时闭包(closure)。当Javascript执行一个对象的方法时使用的环境就是其创建时闭包。这是Javascript的一个特点。

```
var sMessage = "hello world!";
function sayHelloWorld() {
    alert(sMessage);
}
var s1 = new sayHelloWorld();
sMessage = "welcome!";
var s2 = new sayHelloWorld();
s1();    // 执行s1的构造函数    ?    //hello world!
s2();    //welcome!
```

* 局部变量必须先定义再使用，而且它们无论在函数哪个位置定义(例如，for语句的内部)，其作用域都是整个函数范围。

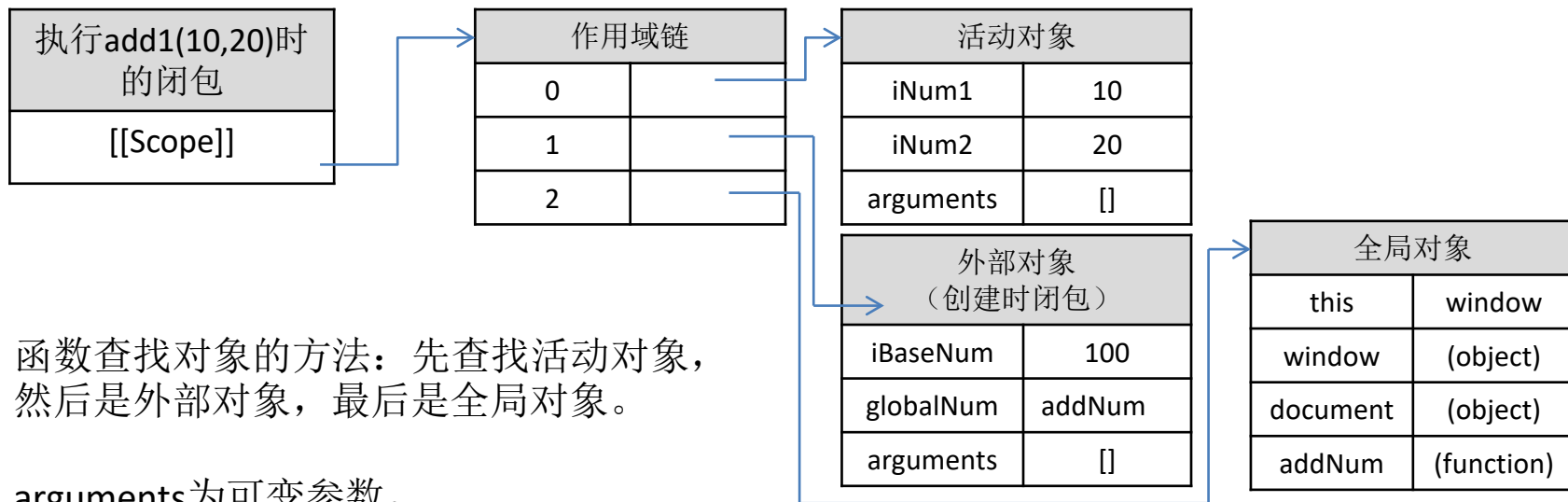
```

var globalNum = 300;
function addNum(iBaseNum) {
    return function(iNum1, iNum2) {
        return iNum1 + iNum2 + iBaseNum + globalNum;
    }
}

var add1 = addNum(100);           // 保存执行addNum时的环境
//globalNum = 600;               // 取消注释后结果有什么变化？为什么？
var add2 = addNum(200);         // 保存执行addNum时的环境
alert(add1(10,20));              // 430                                定义函数addNum时的闭包
alert(add2(10,20));              // 530                                包含globalNum。

```

创建add1函数的闭包就是当时addNum函数的对象，实际执行add1时的闭包包含活动对象(局部数据)、外部对象（创建时闭包）、全局对象。



如果定义了一个函数，希望被多个对象使用，就需要采用call方法改变调用函数的环境：

```
function Person(name){
    this.name=name;
};
p1={
    name: "David",
    sayName: function(age,job){
        alert(this.name + " " + age + " " + job);
    }
}
p1.sayName(21,"Fireman");
p2=new Person("John");
p1.sayName.call(p2, 31, "Engineer");
```

apply的功能和call一样，只是只能采用一个数组参数。apply方法在函数的参数个数可变时非常有用。

```
p1.sayName.apply(p2, [31, "Engineer"]);
```

bind也可以用来改变环境，只是bind返回一个在新环境下的函数，而不是直接执行它。

```
var newSayName=p1.sayName.bind(p2);
newSayName(31, "Engineer");
```


this是环境中的一个变量，代表当前对象。

```
alert(this === window);  
function Person(){  
    alert(this === window);  
}  
Person();  
var person1 = {  
    name:"John",  
    sayName:function() {  
        alert(this === person1);  
        alert(this.name);  
    }  
}  
person1.sayName();  
alert(person1.name);
```

输出:

true
true
true
John
John

如果在一个方法内部定义一个函数，**this**代表的是什么呢？

```
alert(this === window);  
function Person(){  
    alert(this === window);  
}  
var person1= {  
    name: "John",  
    sayName: function() {  
        var that = this;  
        alert(this === person1);  
        alert(that.name);  
        function hi(){  
            alert(this === persion1);  
            alert(this === window);  
            alert(that === persion1);  
        }  
        hi();  
    }  
}  
person1.sayName();
```

结果？

采用闭包的方法建立对象:

```
var book=(function(){           // 匿名函数, (function(){}))是函数表达式
    function Book(title,author){
        this.title =title;
        this.author=author;
        this.sayTitle=function(){alert(this.title)};
    };
    return new Book("The Million Pound Note", "Mark Twain");
})();                           // (function(){}())也可以
alert(typeof book);             // object
book.sayTitle();                // The Million Pound Note
var book2= new Book("title","author"); // 出错, 因为Book不可见
```

数组

- 定义

```
var a = new Array();           // 等同于 var a=[];
var b = new Array(2);          // 两个元素
var c = new Array("tom",3,"jerry");
var d = ["tom",3,"jerry"];      // 等同c;

alert(a.length);               //0
alert(b.length);               //2
alert(c.length);               //3
alert(Array.isArray(a))        //true
```

● 引用

```
var colors=["red","green","blue"];
alert(colors.toString());           //red,blue,green
alert(colors.valueOf());            //red,blue,green
alert(colors);                      //red,blue,green
alert(colors.join(";"));            //red;blue;green
alert(colors.push("yellow","brown")); //5. red,blue,green,yellow,brown
alert(colors.pop());                //brown
alert(colors.shift());              //red
alert(colors.unshift("brown"));     //4.brown,blue,green,yellow
alert(colors.sort());               //blue,brown,green,yellow
alert(colors.reverse());            //yellow,green,brown,blue
```

```
var mycars = new Array()
mycars[0] = "Saab"
mycars[1] = "Volvo"
mycars[2] = "BMW"
for (var x in mycars){              //x的作用域为当前函数体。不能用于对象
    document.write(mycars[x] + "<br />")
}
```

Javascript[数组](#)的常用方法:

| | |
|----------|-------------------------|
| concat | 连接两个或更多的数组，并返回结果。 |
| join | 把数组所有元素放入一个字符串，可以指定分隔符。 |
| pop | 删除并返回数组最后一个元素。 |
| push | 向数组末尾添加一个或更多元素，并返回新长度。 |
| shift | 删除并返回数组第一个元素。 |
| unshift | 向数组的开头添加一个或更多元素，并返回新长度 |
| reverse | 颠倒数组中元素的顺序。 |
| slice | 从某个已有的数组返回选定元素 |
| sort | 对数组元素进行排序 |
| splice | 删除元素，并向数组添加新元素 |
| toString | 把数组转换为字符串 |

• 遍历

```
function prn(){
    for(var i=0;i<arguments.length;i++){
        document.write(arguments[i]+"<br>");
    }
}

var colors = ["red","green","blue","yellow","brown","gray","purple"];
alert("OK");
colors.forEach(function(value,index,fullArray){ // 遍历所有数组元素
    prn(value+ " is " + (index+1) + "th color of "+fullArray.length + " colors");
});
var allTwoOrMoreChars=colors.every(function(value,index,fullArray){
    return value.length>2; // 每个数组元素都返回true，结果才为true
});
prn(allTwoOrMoreChars); //true
var someSixOrMoreChars=colors.some(function(value,index,fullArray){
    return value.length>=7; // 某个数组元素返回true，结果就为true
});
prn(someSixOrMoreChars); //true
var oneCharColors=colors.map(function(value,index,fullArray){
    return value.charAt(0); // 结果为由遍历每个数组元素时的返回值构成的新数组
});
prn(oneCharColors.join(",")); //r,g,b,y,b,g,p
var filterColors=colors.filter(function(value,index,fullArray){
    return value.indexOf("e")>=0; //结果为由返回值为true的数组元素构成的新数组
});
prn(filterColors.join(",")); //red,green,blue,yellow,purple
```

• Json的数据格式

```
var people={
  "programmers": [{
    "firstName": "Brett",
    "lastName": "McLaughlin",
    "email": "aaaa"
  }, {
    "firstName": "Jason",
    "lastName": "Hunter",
    "email": "bbbb"
  }, {
    "firstName": "Elliotte",
    "lastName": "Harold",
    "email": "cccc"
  }],
  "musicians": [{
    "firstName": "Eric",
    "lastName": "Clapton",
    "instrument": "guitar"
  }, {
    "firstName": "Sergei",
    "lastName": "Rachmaninoff",
    "instrument": "piano"
  }],
  "authors": [{
    "firstName": "Isaac",
    "lastName": "Asimov",
    "genre": "sciencefiction"
  }, {
    "firstName": "Tad",
    "lastName": "Williams",
    "genre": "fantasy"
  }, {
    "firstName": "Frank",
    "lastName": "Peretti",
    "genre": "christianfiction"
  }]
}

alert(people.authors[1].genre);      ?    // Value is "fantasy"
alert(people.musicians[1].lastName); // Value is "Rachmaninoff"
alert(people.programmers[2].firstName); // Value is "Elliotte"
```


利用Javascript(ECMAScript 5)内置的JSON对象可以实现JSON字符串与对象的互相转换:

```
var text = JSON.stringify(['hello', {who: 'Greg'}]);
alert(text);                                //[ "hello", { "who": "Greg" } ]
var obj=JSON.parse(text);
alert(obj[0]);                              // hello
alert(obj[1].who);                          // Greg
var text1='{ "name": "Greg", "job": "Driver", "birthdate": "1990-9-1" }';
var obj1 = JSON.parse(text1);
alert(obj1.birthdate);                      //1990-9-1
var obj2 = JSON.parse(text1,
    function (key, value) {
        return key.indexOf('date') >= 0 ?
            new Date(value) : value;});
alert(obj2.birthdate.getFullYear()+2);      //92
```

用eval也可以实现字符串转换为object:

```
eval("var obj3="+ '{ "name": "Greg", "job": "Driver", "birthdate": "1990-9-1" } ');
alert(obj3.birthdate);                      //1990-9-1
```

由于有安全性问题,一般不主张使用eval。