# JavaScript程序设计（上）

2017.12.8

isszym　sysu.edu.cn

w3school runoob ecma

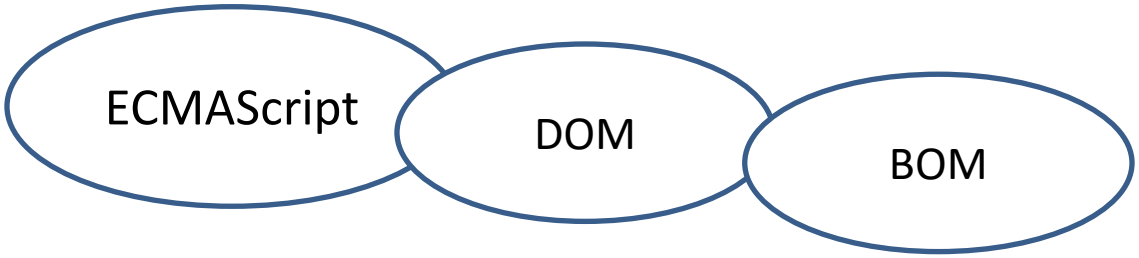
# 目录

# 概述

- 1995 年，Netscape Navigator 2.0 增加了一个由网景(Netscape)公司与 太阳(Sun)公司共同开发的称为 LiveScript 的脚本语言，当时的目的是在浏览器和服务器端使用它。在 Netscape Navigator 2.0 即将正式发布前，Netscape将其更名为 JavaScript（简称JS），目的是为了利用 Java 这个因特网时髦词汇。
- 在网页下载后执行其中的JavaScript程序可以变换显示和增加互动性。JavaScript是一种类型的**解释执行语言**，只是在执行前才进行编译源程序语句 。使用了JavaScript的HTML称为Dynamic HTML（DHTML）。
- 因为 JavaScript 1.0 如此成功，Netscape公司在 Netscape Navigator 3.0 中发布了 1.1 版。微软也很快发布了 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 CEnvi 中的 ScriptEase。
- 与 C 和其他编程语言不同的是，当时的JavaScript 并没有一个标准来统一不同版本JavaScript的语法和特性。随着业界担心的增加，这个语言的标准化变得势在必行。

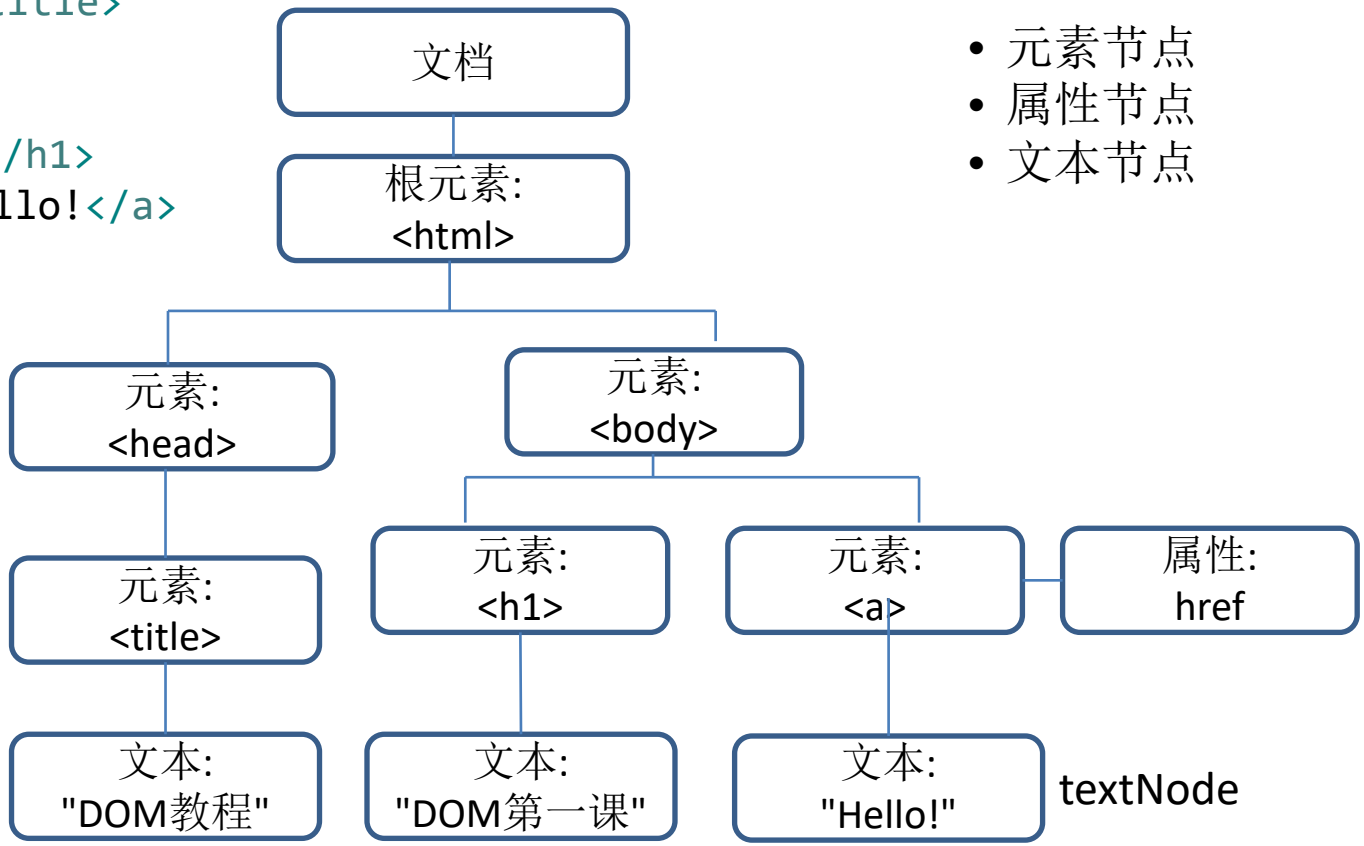
参考

- 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 标准ECMA-262，即ECMAScript。
- 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。该标准的最新版为ECMAScript 8（ES 8）。
- 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。



# 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语言来修改树的结构和节点的内容

window对象、document对象、location对象

测试

```
<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,Click here!</p>
</body>
</html>
```

- document代表当前网页对象
- document.write("abc")网网页对象中写入字符串"abc"
- Javascript的全局变量和全局函数都定义在window对象中

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);
}
```

```
<span style="font-weight:900">HaHa，
我是Ajax！ </span>
```

# JavaScript变量

- 定义
  - **不经定义直接被赋值的Javascript变量为全局变量**，都作为window对象的属性。
  - 局部变量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。
  - Javascript变量分为基本类型和引用类型。
  - 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;                          // 变量定义(可变类型)
    var y=3;                        // 定义并初始化(数值型)
    z="hello";                      // 未定义变量(全局变量)
    var a=3.5, b="123456";          // 一次定义多个变量
    b=5;                            // 改变值和类型(不推荐)
</script>
```

- 基本类型

定义为基本类型的变量直接保存值。有如下五种基本类型，其中的数值类型、布尔类型、字符串类型对应的类分别是Number、Boolean、String：

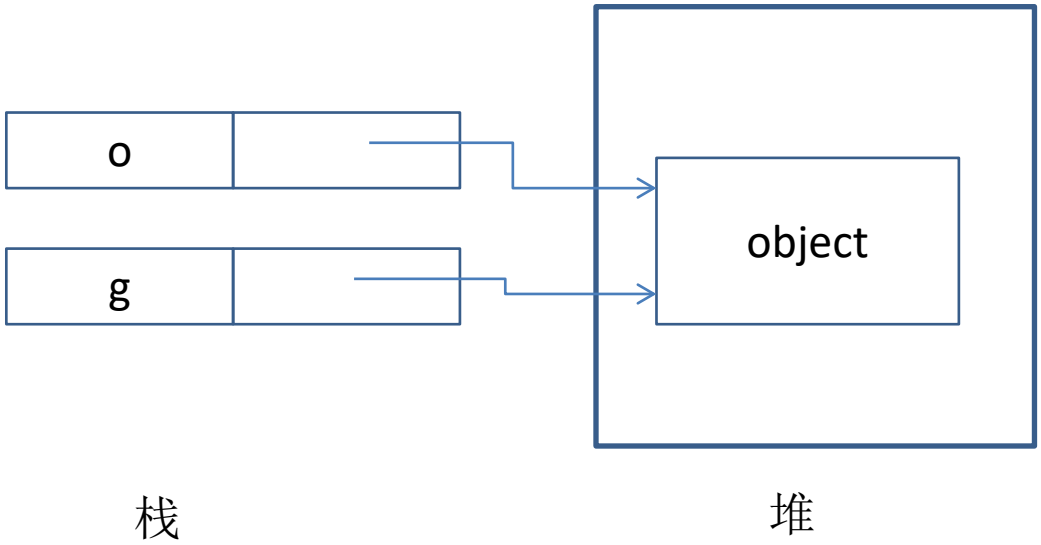
| | |
|---|---|
| number | 数值型， 取值整数和小数。070(8进制)，0xFF，100，0.345 |
| boolean | 布尔型，取值 true 或 false。true转化为整数1，false为0 |
| string | 字符串， 用单引号或双引号括起的单个或连续字符。 |
| null | 空类型，只取这个值，是未初始化对象的取值 |
| undefined | 未定义，只取这个值，是未初始化变量的取值 |

```
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"
```

- 引用类型

定义为引用类型的变量保存指向实际内容的指针。JavaScript中的对象(Object)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 (见下图)
var o=null;         // o不再指向任何对象。
var g=null;         // 在不使用该变量时最好解除引用
                    //    没有任何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的值、非数值字符串转换成的数值等 |
| **false** | 0、空串、null、undefined、NaN |
| true | 其它 |
| isNaN() | 判断参数是否为数值 |
| isFinite() | 数值是否在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)     //写入到网页中
```

# 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的字符串参数可以是一段程序
                             //或一个表达式。（由于有副作用，一般不建议使用）
```

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

关系运算符：a>b　a<b　a>=b　a<=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：　~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 []() →单目→ * / % → + - →<< >> >>>
→ < > <= >= →== != → & →^
→ | →&& →|| →三目运算 →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(op包含null、undefined和NaN时返回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(数组元素为基本类型)
        sum+=scores[score];          //或者为属性名和方法名(数组为对象)
        cnt++;
}
avg = sum/cnt; // avg=88.34

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

对比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
```

* 还有with语句，用于重复引用一个对象的简略写法。因为有副作用，一般不建议使用。

```
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
        }                  <-------- continue跳到这里
}   //sum=495             <-------- 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
        }                  <-------- break跳到这里
}   //sum=165             <-------- break Label2跳到这里
```

# 字符串

字符串是 JavaScript 的一种基本类型。**JavaScript 的字符串变量是不可变的（immutable），每次赋值该变量都会指向一个新字符串，旧字符串由垃圾回收器回收**。==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查找子串第一次出现处的位置(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(",");  //得到字符串：“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：<a href="http://www.w3school.com.cn">Free Web Tutorials!</a>
"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

**JavaScript的函数是Function类的一个实例。**函数名为引用类型变量，指向该函数对象。JavaScript实际上是一门函数式语言。

**方法1：定义全局函数**

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

**方法2：字面量定义**

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

**方法3：Function实例**

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

函数体

- 第一种方法定义了window对象的一个方法，也是一个变量。
- 第二种和第三种方法直接把函数定义为一个变量。

- JavaScript的函数就是对象

```
function sum(num1,num2){
    return num1+num2;
}
alert(sum(1,2));                    // 返回 3
var asum = sum;                     // 函数为对象，可以直接赋值
alert(asum(2,3));                   // 返回 5
sum = null;                         // 清除对象sum
alert(asum(4,5));                   // 返回9。asum依然可用
asum=function(num3,num4){           // 重新赋值给asum，原函数变得不可用
    return num3-num4;
}
alert(asum(10,10));                 // 返回0 。 没有重载，直接覆盖
```

- 函数引用

情形1：调用错误：如果sum作为字面量定义，必须先定义再引用

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

情形2：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;
}
```

- 局部变量的作用域

LHS-Left Hand Side

变量被赋值（LHS）时会逐层函数查找其定义(var)，如果没找到，则会被定义为全局变量。变量被引用（RHS）时也会逐层找其定义，没找到则取值为undefined。

```
var y = 10;
function show(x) {
     var y = 2 * x;   //定义且赋值
     z = 2 * y;       //直接赋值
     function add(w){
         return x+y+z+w;
     }
     return add(2 * z);
}
console.log(y, show(2),z); //10,30(2+4+8+16),8
console.log(add(5));       //Reference Error
```

每个作用域中定义的变量：

第一层为全局作用域　y，show，z
第二层为show的作用域　x，y，add
第三层为add的作用域　w

按照ES6之前的标准，JavaScript没有块作用域，因此，尽管下面的变量*i*和变量*j*都在for语句中定义，但是其作用域为整个inc函数。

```
function inc(){
   for(var i=0;i<10;i++){
      var j=20;
   }
   console.log(i,j); //10 20
}
inc();
```

ES6定义了let和const，它们的作用域被局限于块（if，for，while，{}）中。

```
function inc(){
   for(let i=0;i<10;i++){
      const j=20;
   }
   console.log(i,j); // undefined
}
inc();
```

- 递归函数

```
// 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。

```
function add(){                                    // 可变参数
    var c=0;
    for (var i=0; i<arguments.length;i++){
          var c = c + parseInt(arguments[i]);
    }
    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>");
```

- 包装函数和匿名函数

如果函数只被调用一次，可以包装该函数使其从全局变量变为局部变量。

```
(function sum(x,y){              // (function(){})为函数表达式
          console.log(x+y);
})(10,20);                       // 30. (function(){})() 表示立即执行
sum(30,40);                      // undefined. function无包装时取值70
```

省略函数名后该函数成为了匿名函数。

```
(function (x,y){
    console.log(x+y);
})(10,20);                        // 30
```

使用匿名函数会增加调试难度，降低代码可读性，也很难递归调用自己（arguments.callee已过时）。

# 对象

- 创建对象

**方法1、用Object定义**

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

**方法2、用字面量定义**

```
var person1 = { name: "Nicholas",
                age: 26,
                print: function(){alert(person.name)};
              };

var person2 = { "name": "Nicholas",
                "age": 26,
                5: true
              };
person2["5"]=false;
Person2["tel"]="13012345678";    //新属性
```

* 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。

## 方法3、用函数（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
```

- 因为函数是对象，Person本身是对象，也可以作为构造器创建对象。
- 全局函数都是window对象的方法，如果直接调用Person，this代表window对象。

- 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

if(typeof person1.name=="string") alert("1");
if(typeof person1.age=="number") alert("2");


delete person1.name;                          // 删除属性
```

- JS还可以用Object.create()创建对象，见原型一节

- this代表当前对象

this是环境中的一个变量，在对象方法中表示当前对象。由于全局函数是window对象的方法，其中的this为window对象。

```
alert(this === window);           // true
function Person(){
   alert(this === window);
}
Person();                         // true
var p1=new Person("John");        // false

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

如果在一个方法内部定义一个函数，this代表的是什么对象呢？

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

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

- 直接执行构造函数

构造函数Person也是window对象的一个方法，下面构造函数的执行会产生window对象的属性和方法：

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

Person("David",23,"Engineer"); //函数中this为window对象
alert(name);
sayName();                     //同this.sayName()或window.sayName()
```

- 函数绑定对象

如果定义了一个函数，希望被多个对象使用，就可以采用apply方法和call方法取绑定对象。下面例子中函数sayName()既可以用作p1的方法，又可以用作p2的方法。

```
function Person(name){
    this.name=name;
};
var p1 = new Person("John");
var p2 = {name: "David"};

var sayName = function(age,job){
                  alert(this.name + " " + age + " " + job);
               }
sayName.call(p1, 31, "Engineer");          // John 31 Engineer
sayName.apply(p2, [32, "Fireman"]);        // David 32 Fireman
var newSayName = sayName.bind(p2);
newSayName(33, "Engineer");                 // David 33 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。 bind返回在新环境下的一个函数。

- 全局方法作为对象方法

前面创建对象的方法都有一个缺点，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);
}
```

# 原型

- 定义

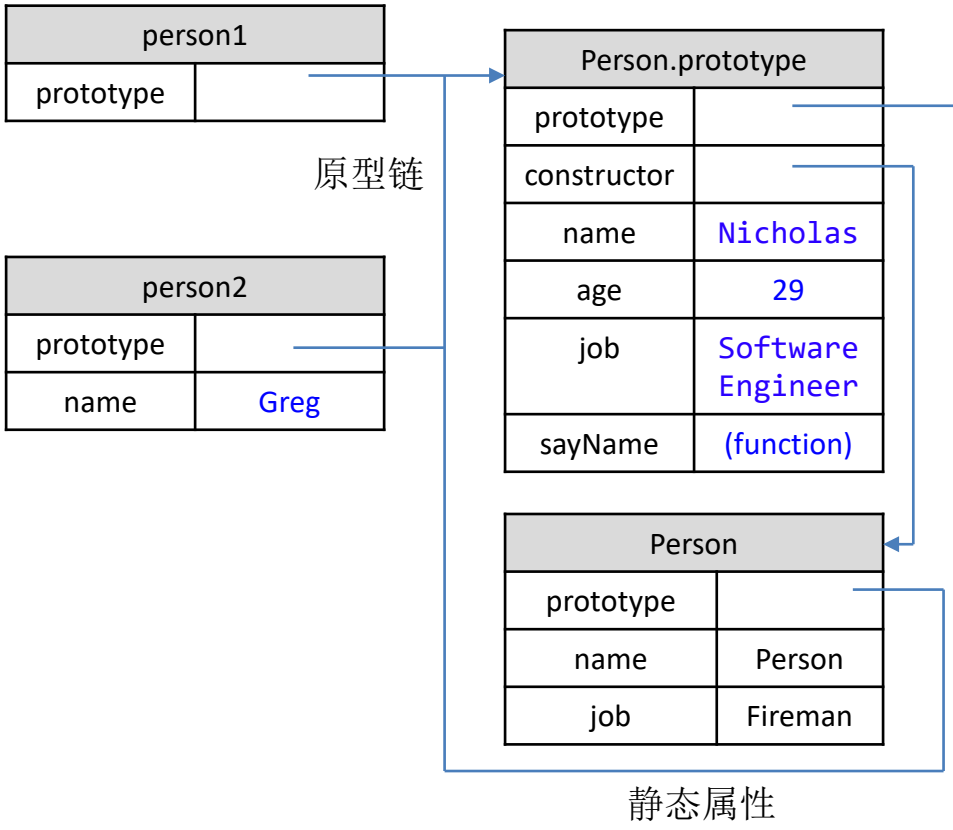
每一个对象都会有一个称为原型的对象属性(prototype或__proto__)。对象或原型上都可以定义属性和方法。引用对象的属性和方法时会先查找对象上定义的属性和方法，如果找不到，再查找其原型上定义的属性和方法。

```
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--来自函数对象
alert(person1.job);                         // Software Engineer--来自原型
alert(Person.name);                         // Person--函数名
```

```
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。
- 原型方法中对当前对象的属性和方法的访问一定要加上this。

- 实例、属性和原型判断

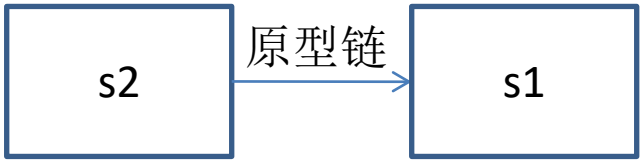
```
//接上面的例子
alert(person1 instanceof Person);         //true
alert(person1 instanceof Object);         //true
alert(person1.hasOwnProperty("name"));    //false. 来自原型，非自有属性
alert(person2.hasOwnProperty("name"));    //true.  是自有属性
alert("name" in person1);                 //true.  是它的属性（可以来自原型）
alert("name" in person2);                 //true
alert(Person.isPrototypeOf(person1));           //false
alert(Person.prototype.isPrototypeOf(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.propertyIsEnumerable("job")); //true.可枚举的
alert(person1.propertyIsEnumerable("job"));//false.只有自定义属性才可枚举
```

- **继承性**

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。

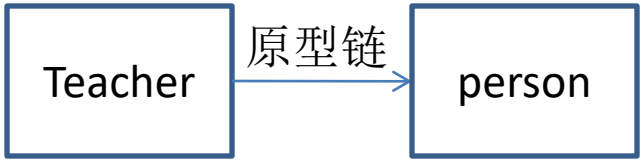
方法1：在创建对象时形成原型链

```
var s1={a:2};
var s2=Object.create(s1);
console.log(s2.a);  //2
console.log(s1.isPrototypeOf(s2));  //true
```




方法2：直接给原型赋值

```
function Person(name) {this.name=name;}
function Teacher() {}
var person = new Person("Nicholas");
Teacher.prototype = person;                    // 继承(构造函数变为Person)
Teacher.prototype.constructor = Teacher;       // 重新设置构造函数
var teacher1 = new Teacher();
console.log(teacher1.name);                    // Nicholas--来自原型
console.log(person.isPrototypeOf(teacher1));   // true
```




参考1 参考2

# 闭包

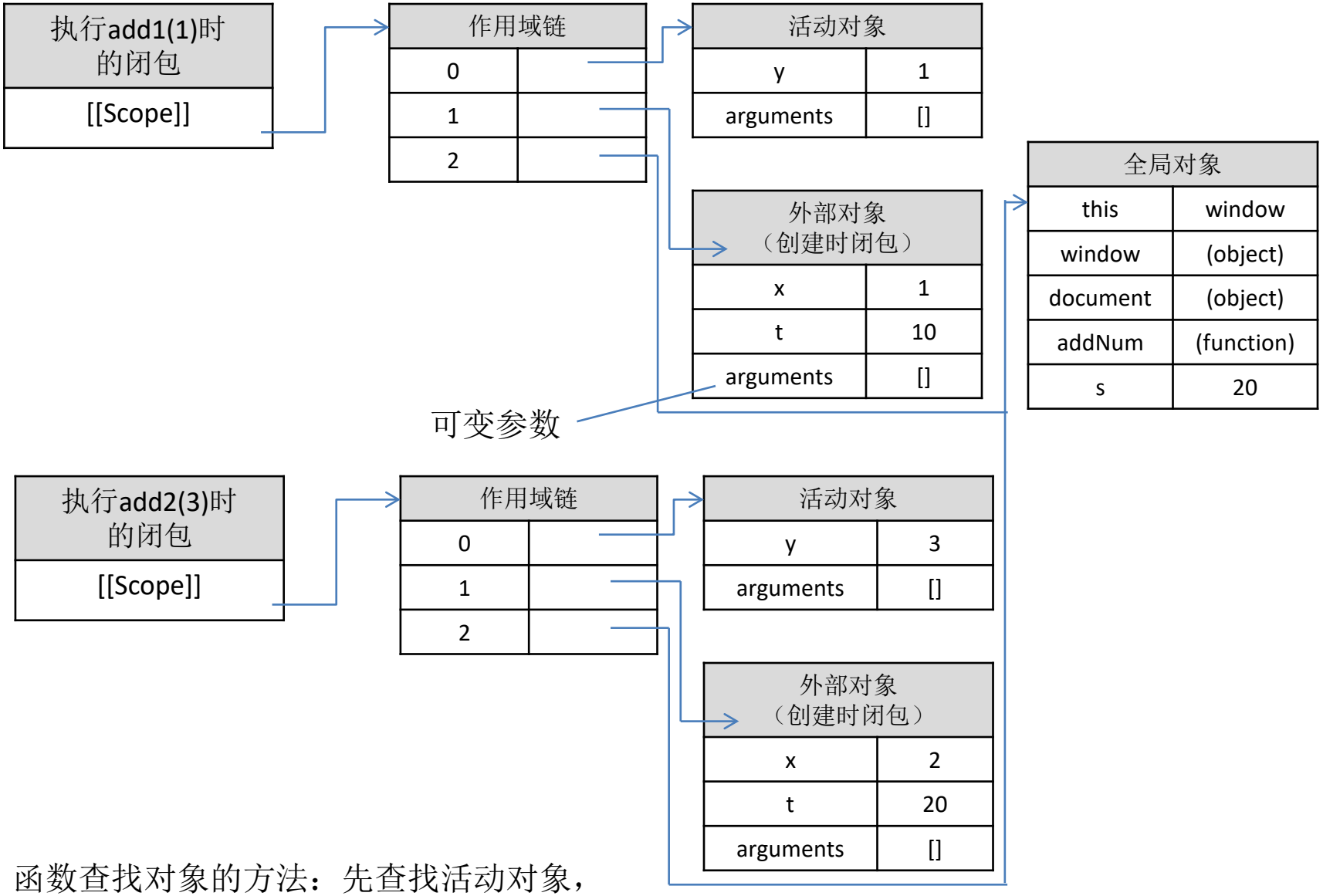
- 定义

创建函数(或对象方法)时自动保存作用域内的上下文环境（参数和局部变量）供以后调用时使用。这里保存上下文环境的对象就是闭包(closure) 。

```
var s = 10;
function addNum(x) {
    var t = s;
    var f = function(y) {
      return s + t + x + y;
    }
    return f;
}
var add1 = addNum(1);
s=20;
var add2 = addNum(3);
alert(add1(2));          // 33
alert(add2(4));          // 47
```

```
 s   t  x   y
20  10  1   2
20  20  3   4
```

* 如果外层为window对象，其闭包由全局变量组成。




函数查找对象的方法：先查找活动对象，然后是外部对象，最后是全局对象。

- 利用包装函数形成闭包

```
var f=(function(x){
        return function(y){console.log(x,y)};
    })(10);
f(20);
```

* 这种做法常用于把参数带入到事件函数中。

参考

# 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(Arrays.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){
   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 |

# 附录1、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.getYear()+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
```

由于有副作用，一般不主张使用eval。

# 附录2、数组元素遍历

```
function prn(){
   for(var i=0;i<arguments.length;i++){document.write(arguments[i]+"<br>");}
}
var colors = ["red","green","blue","yellow","brown","gray","purple"];
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
var s=[1,2,3,4,5].map(function(n) { return (n<=1)?1:arguments.callee(n - 1)*n;});
                                 //1 2 6 24 120  arguments.callee为调用函数
```

# 附录3、对象属性

- 对象属性枚举

```
var person = {name:"Nicholas",  age:26};
for(var prop in person){                                  //列举所有可枚举的属性
   document.write("name:"+prop  +"  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]
                    +"  value:"+person[props[i]]+"<br>");
}
```

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

```
alert("age" in person);                                   //true （age为自有属性）
alert("toString" in person);                              //true （toString为原始属性）
alert(person.propertyIsEnumerable("age"));                //true （自定义属性）
alert(person.propertyIsEnumerable("toString"));//false（原始属性）
```

* toString是一个自Object继承来的原生属性。原始属性默认是不可枚举的，自定义属性默认是可枚举的。
* 在ECMAScript 5中可以修改属性的枚举特征，详细请见附录。

- 创建属性

直接给新属性赋值可以创建属性，还可以通过Object.defineProperty()定义带有配置的属性，包括是否可以在foreach语句中进行枚举 (enumerable)、是否可以改变配置的值(configurable)、是否可以进行赋值(writable)。

```
var person1 = {};
Object.defineProperty(person1,
      "name",
      {
        value:"Nicholas",
        enumerable:true,
        configurable:true,
        writable:true
      }
)
alert(person1["name"];    // Nicholas
```

详细见附录

- 基本类型变量的新属性不可用：

```
var a = "hello";            //原始类型（没有关联任何方法）
var s1 = a.substring(2,4); //ll
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，系统内部实际上要引入String类型的临时变量才可以执行其方法：

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

# 附录4、闭包额外的例子

***closure1.html***

```
var c = 3;
function foo(){
    var b = 2;
    function boo(){
        var a = 1;
        function bar(){
            console.log(a,b,c);
        };
        return bar;
    };
    return boo;
};
var baz=foo();
var eoo=baz();
eoo();    //显示1、2、3
```

***closure2.html***

```
var c = 4;
function foo(){
    var b = 3;
    function boo(x) {
        var a = x;
        this.sayHello=function(){
            console.log(a,b,c);
        };
    };
    return new boo(2);
};
var baz=foo();
baz.sayHello();    //显示2、3、4
```

***closure3.html***

```
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");
        }();                                    // 直接执行函数
alert(typeof book);                             // object
book.sayTitle();                                // The Million Pound Note
var book2= new Book("title","author");          // 出错，因为Book不可见
```

参考

# 附录5、创建带属性特性的对象

```
// Object.create的第一个参数赋值给对象的原型属性
//                第二个参数用于定义其它属性
var Person=function(){};
Person.prototype.name="David Zhang";
Person.prototype.toString=function(){return this.name+" "+this.age;}
var teacher=Object.create(Person.prototype,{
                    age:{
                        configurable:true,
                        enumerable:true,
                        value:"30",
                        writable:true
                    },
                    sayName:{
                        configurable:false,
                        value:function(){return "***"+this.name;}
                    }
            }
)
alert(teacher.name);          // David Zhang
alert(teacher.sayName());     // ***David Zhang
alert(teacher);               // David Zhang 30  ---- teacher.toString()
```

# 总结

- 概述
- 文档树
- JavaScript变量
- 运算符与表达式
- 基本语句
- 字符串
- 函数
- 对象
- 闭包
- 原型
- 数组

- 附录1、对象属性的特性
- 附录2、Json的数据格式
- 附录3、闭包额外的例子
- 附录4、创建带属性特性的对象
- 附录5、数组元素遍历