JavaScript程序设计（上）

2019.4.22

isszym　sysu.edu.cn

w3school runoob ecma mozilla

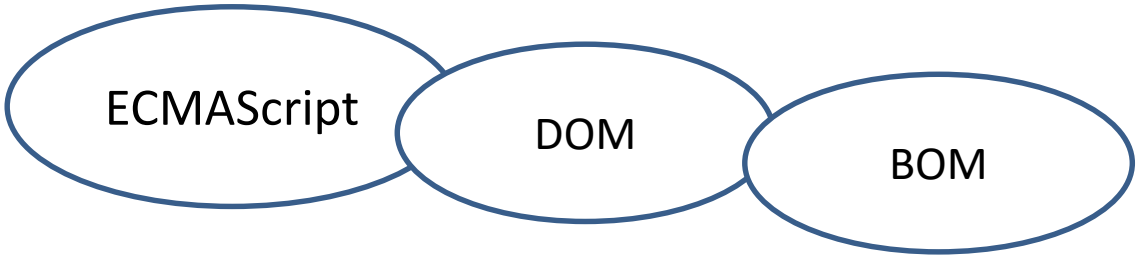
目录

概述

- 在Web开发早期，前台填写的网页表单都要提交给后台进行验证，由于当时网络速度很慢，需要等待几十秒才能得到服务器的反馈信息。如果可以在前台完成一些基本的验证绝对是值得兴奋的。
- 1995年2月，拥有当时最强浏览器Navigator的网景(Netscape)公司决定着手开发一种客户端语言LiveScript，用来处理这样的验证。为了加快进度，网景(Netscape)公司联合Sun公司一起进行开发，并于1995 年12月4日发布了用于 Navigator 2.0的脚本语言JavaScript（简称JS），虽然把名字中的Live替换成了当时热得发烫的Java， 实际上JavaScript与Java关系不大。加入了JavaScript的HTML称为DHTML(Dynamic HTML)。
- 因为 JavaScript如此成功，微软很快在发布的 IE 3.0中搭载了一个 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。随着业界对这种分歧担心的增加，这个语言的标准化变得势在必行。

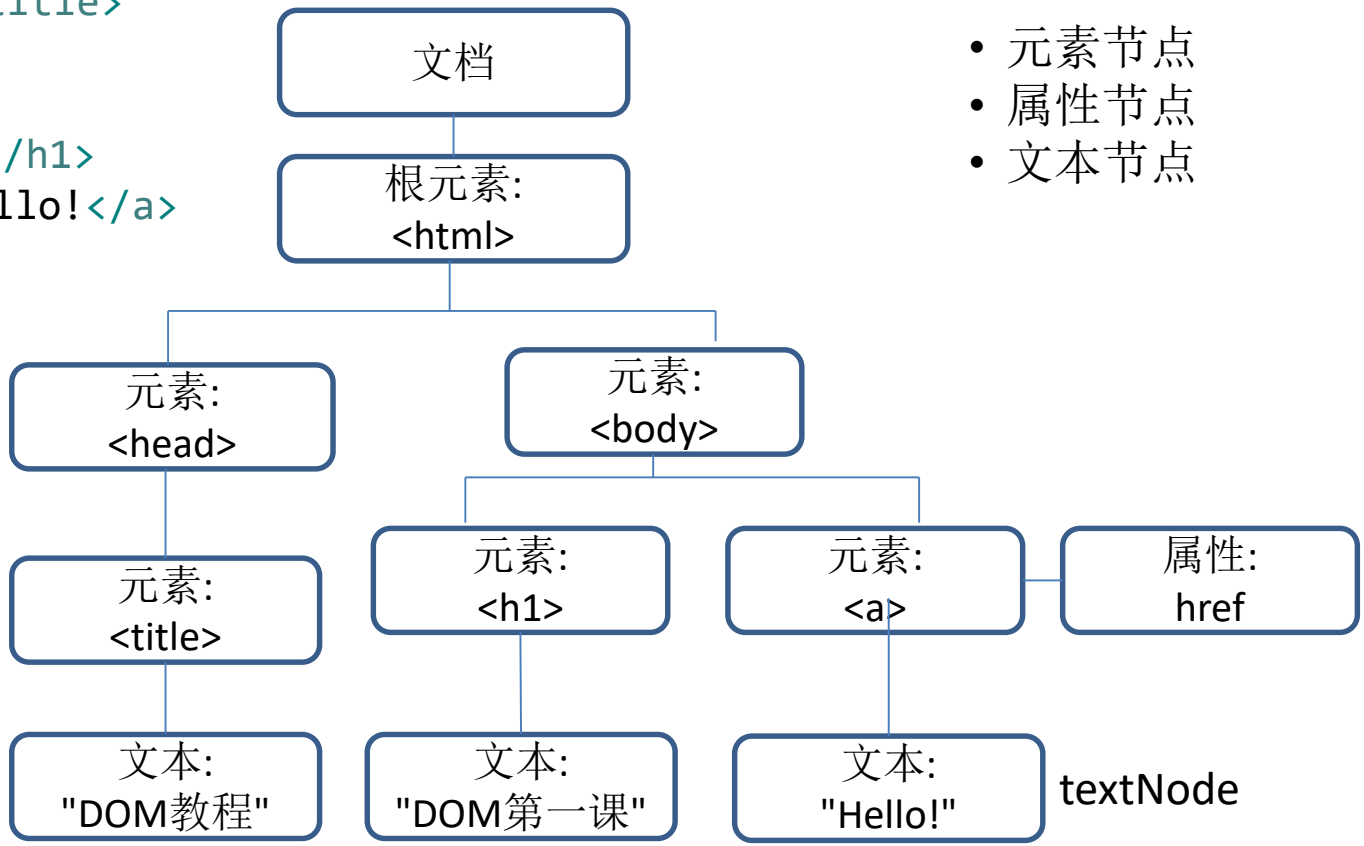
参考

- 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 -- 包含JS可以操作的所有内容
document对象 -- 网页

location对象 -- 地址栏
history对象 -- 历史

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")在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://172.18.187.9:8080/lab/test/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>
```

* 区分大小写

JavaScript变量

- 定义
  - **不经过定义直接被赋值的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;                            // 改变类型（不推荐）
</script>
```

- 基本类型

变量有如下五种基本类型：

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，只有这一个值，是未初始化对象的取值
undefined	未定义，只有这一个值，是未初始化变量的取值

其中，数值型、布尔型、字符串对应的类分别是Number、Boolean、String。下面的"typeof 变量名"可以得到变量类型。

```
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"
```

* 常用isXXX或hasXXX或canXXX来命名布尔变量

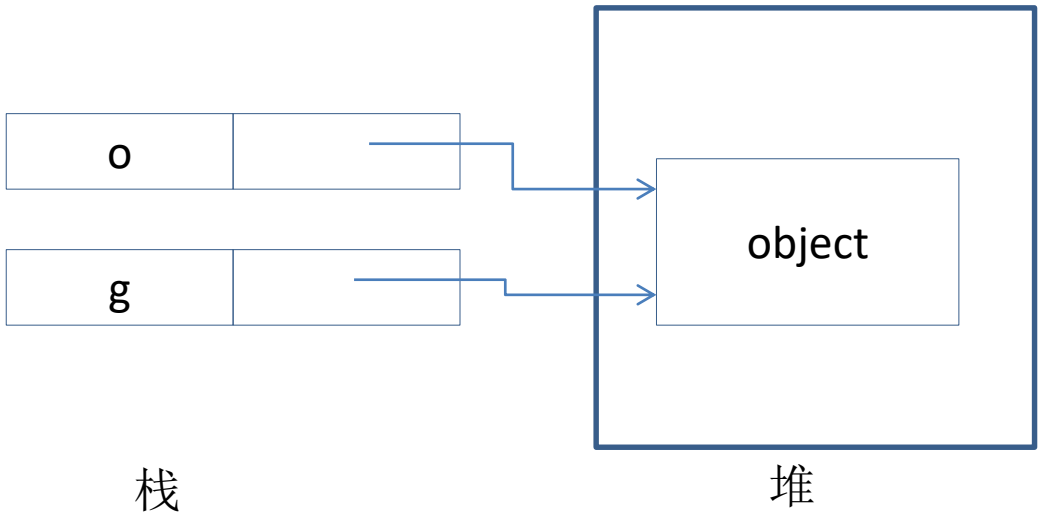
定义常量一般采用全大写方式，并用下划线分割单词，现在一般用const定义

```
var HOST_URL = "chotel.com";
const ROUTE_NUM = 20;
```

- 引用类型

与基本数据类型的变量直接保存内容（值）不同，引用类型的变量保存的是指向内容的指针。JavaScript中的对象(Object)采用引用类型。

```
var o = new Object(); // typeof  o = "object"
var g = o;            // 复制指针 (见下图)
g.name = "Wang";      // 对象的属性是对象的无序列表，采用名和值方式保存。
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、未定义的函数、未定义的变量
true	其它（除了false的那些取值之外的取值）
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, b = "5";       // 可以用一个语句声明多个变量，但是不推荐
var c1 = a + b;           // 35    字符串加法
var c2 = b + a;           // 53     要用有含义的名字或简写作为变量名
var d = a + parseInt(b); // 8,     把字符串转换为整数.
var e = a + (b - 0);      // 8      减法的运算结果为数值型或者出错(NaN)
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;   // 右值为算术表达式，exp计算结果为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，先转换为整数(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) a!==b

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(类型不同或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）
* Math.pow(a,10)和a ** 10都是求a^{10}。不建议使用++和--。三元表达式不应该嵌套，通常写成单行表达式。

基本语句

- 赋值语句和注释语句

```
/** var let const用于变量声明，其数据类型由赋值内容确定，可以改变（不推荐）。
*   var为函数作用域，let和const为块作用域（推荐使用）
*   没初始化的变量的值为undefined。没有这样声明而直接使用的变量为全局变量。
*   注释可以加上TODO或FIXIT，表示下面的代码还没完善或有错
*/
var  x;                     // 变量定义，作用域为整个函数或window对象
x = 20;                     // 赋值语句，左边为变量，右边为表达式。全局变量
let y = 100.2;              // let与var不同的是具有块作用域，而不是函数作用域
x = y;
z = y = 30                  // 单行语句的结束处可以不加分号，js会自动加上
alert(x+" "+y+" "+z);       // 100.2 30 30
const s = x + y + z;        // const与let一样，用于不会改变的变量或常量
console.log(s);             // 160.2 在浏览器控制台中显示(chrome F12)
{ var u = 80;               // 块
  let v = 90;               // 每个变量最好用一个语句进行定义
  const w = 100;
}
console.log(u);             // 80
console.log(v);             // undefined. refereceerror
console.log(w);             // undefined. refereceerror
```

* let和const是ES6增加的内容，推荐使用
* 每个变量定义最好用一个语句

- 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"


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


var total = x || 1000;               // 100.0    短路或--如果x有定义，
                                     //     则total赋值为x，否则为1000
```

*条件嵌套: else属于最靠近它的if

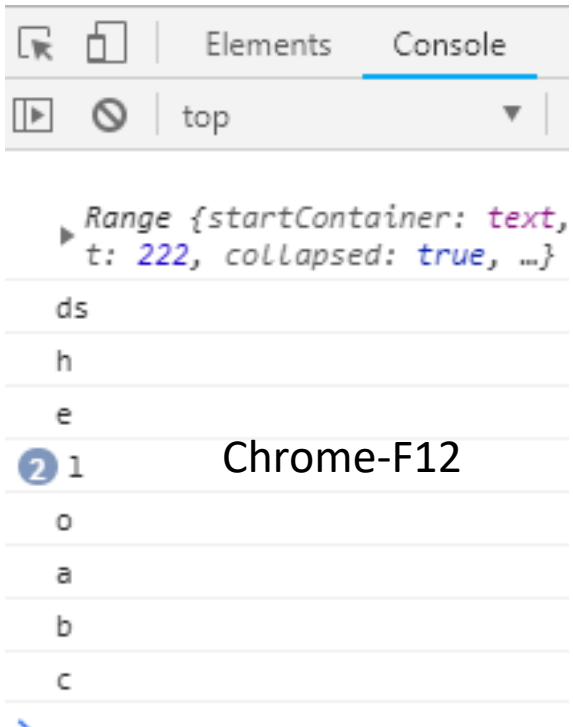
- 循环控制语句

2 个空格作为缩进，大括号前一个空格，小括号前后放一个空格，运算符前后，等号前后均加上一个空格。语句块和下条语句前留一行。文件末尾要有一空行。

```
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) {
  sum += scores[score];
  cnt += 1;                    // 数组或者属性名和方法名
}

avg = sum / cnt; // avg=88.34

let s = "helloabc";
for (let c of s) {
   console.log(c);
}
```

对比Java

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



h
e
l
l
o
a
b
c

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

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
```

* **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。

[for语句大全]

```
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。

```
//concat将两个或多个字符的文本组合起来
var a = 'hello';        // 使用单引号或双引号。谷歌建议只使用单引号
var b = `,world`;       // 使用反引号。如果字符串包含单引号，可以采用反引号
var c = a.concat(b);    // hello,world.     与c=a+b结果相同
var d = `Hi${b}!`;      // Hi,world!.与'Hi,'+b+'!'相同（用反引号才有效）


// 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：  "h,e,l,l,o"

//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。

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 sum =  new Function("num1", "num2",          // num1和num2为参数名
                "return num1+num2;");
sum(2,3);  // 5
```

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;               // 没有重载，直接覆盖
}
alert(asum(10,10));                 // 返回0 。
```

- 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 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(){                          // 使用add(...args)时args替换arguemts
    var c=0;                             //     可以增加固定项：add(x,y,...args)
    for (var i=0; i<argments.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

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){
      console.log(x+y);
})(10,20);
sum(30,40);
(function(x, y){
   console.log(x+y);
})(10, 20); // 30. 匿名函数
```

```
// (function(){})为函数表达式 (可以不要名字sum)
//       最好不要采用匿名方式
// 30. (function(){})() 表示立即执行
// undefined. function无包装时取值70
```

也可以

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

- 可选参数

```
function add(x,y=80){
  console.log(x + y);
}
add(20);   // 100
```

- lamda表达式（函数的简化写法）

```
(((x, y)=>console.log(x + y))(10, 60));      // 70.   与前面的匿名函数类似
(x => {document.write(x + 5); })(10);        // 15.   单参数的简化写法
document.write((x => x + 1)(10));            // 11.   省略了return
```

对象

- 创建对象

* 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1、用Object定义

```
var person = new Object();
person.name = "Nicholas";
person.age = 26;
person.print = function(){alert(person.name)};
var person2 ={};                              // 等同于new Object()，推荐使用
```

方法2、用字面量定义

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

var person2 = { "first-name": "James", //对无效的标识符的属性名使用引号
                5: true
              };
person2["5"] = false;
person2["tel"] = "33093310";   // 用方括号容易增加新属性：
person2.age = person2.age + 1; //      t="tel";person2[tel]="33093310"
```

方法3、用函数（构造器）创建对象

用小驼峰法命名变量、对象、函数和实例。
用Pascal法(大驼峰法)命名构造函数和类。

```
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. person2也一样
alert(person1 instanceof Person);                       // true. person2也一样
alert(person1 instanceof Object);                       // true. person2也一样
alert(person1.sayName===person2.sayName);               // false
delete person1.sayName;                                 // 删除属性
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。

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

- 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");
```

- 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;                   // 用变量储存this，不推荐
        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()。

- 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，只是采用一个数组做参数，适合参数个数可变的情况。 bind用于把一个函数绑定一个对象并获得一个新函数。

- 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

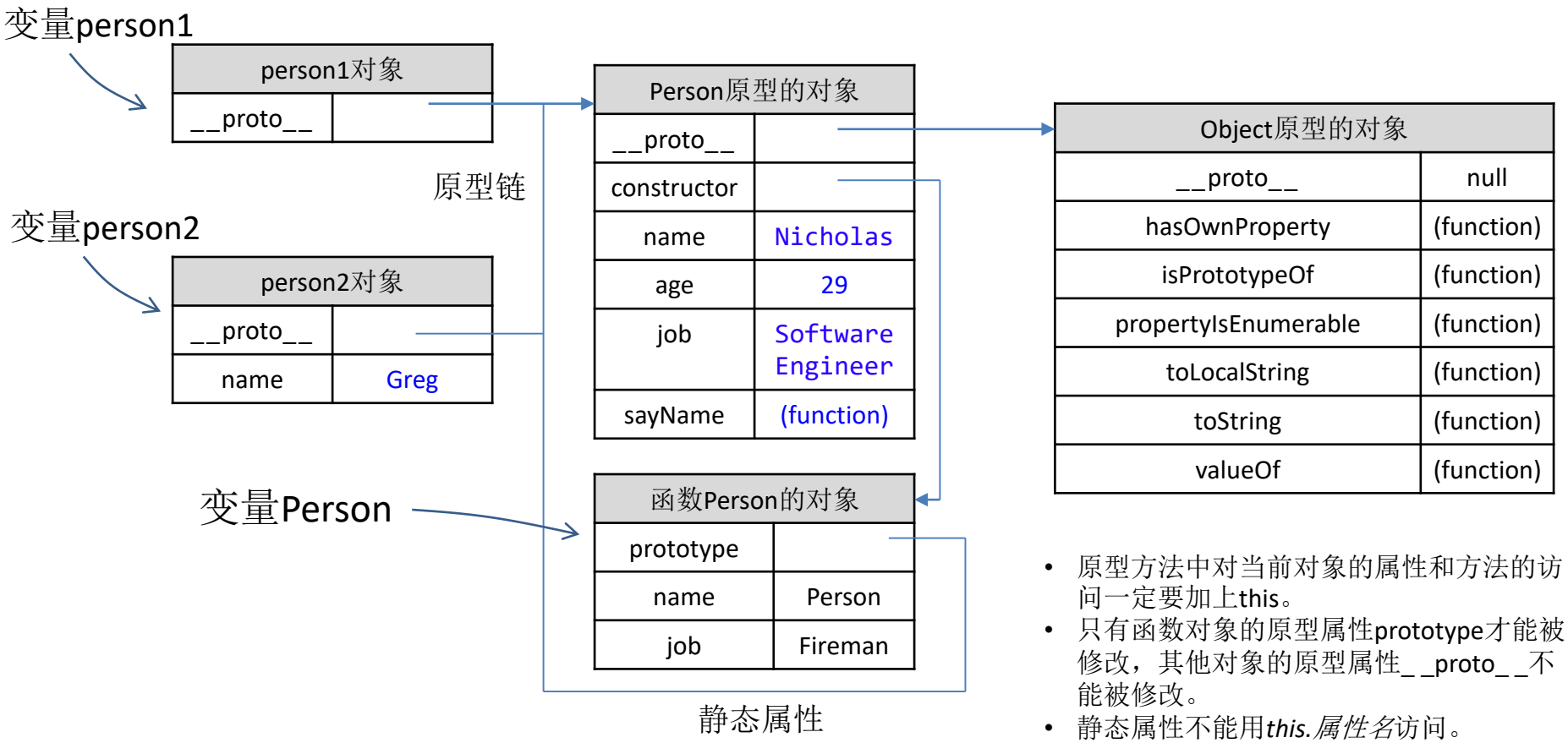
JavaScript是一门基于原型的直译式脚本语言。JavaScript每一个对象都会有一个称为**原型**的对象属性。对象或原型上都可以定义属性和方法。引用对象的属性和方法时会先在对象定义中查找，如果找不到，再在其原型上查找。

```
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（静态属性）

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   静态属性
alert(person1.job);                       // Software Engineer--来自原型
alert(Person.name);                       // Person--函数名   静态属性
```

分析：

```
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
```



- 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。
- 只有函数对象的原型属性prototype才能被修改，其他对象的原型属性__proto__不能被修改。
- 静态属性不能用*this.属性名*访问。

- 实例、属性和原型判断

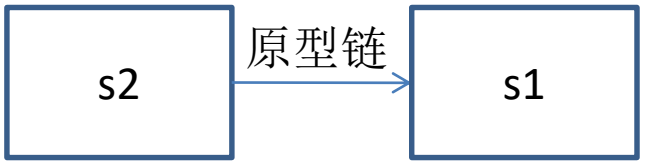
```
/* 接上面的例子 */
alert(person1 instanceof Person);         //true
alert(person1 instanceof Object);         //true
alert(person1.hasOwnProperty("name"));    //false. 来自原型，非自有属性
alert(person2.hasOwnProperty("name"));    //true.  是自有属性
alert("name" in person1);                 //true.  是它的属性（可以来自原型
）
alert("name" 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
alert(person1.__proto__===Person.prototype);    //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。

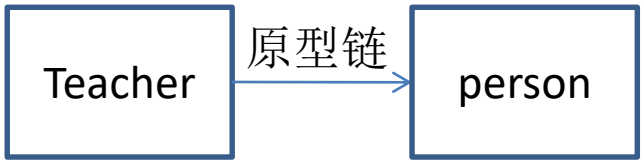
方法1：在创建对象时形成原型链

```
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

- JS的类

ES6定义了一个class关键字来模拟类。js的类没有继承性，不会创建对象副本，只是通过原型做了一个关联，也就是说， js实际上是没有类的。

```
class Point{
        constructor(x,y){
            this.x=x;this.y=y;
        }
        toString(){
          return '('+this.x+', '+this.y+')';
        }
}
var point = new Point(1,2);
alert(point);

// 上面的代码等价于下面的代码
function Point(x,y){
  this.x=x;
  this.y=y;
}
Point.prototype.toString=function(){
  return '('+this.x+', '+this.y+')';
}
var point = new Point(1,2);
alert(point);
```

继承：

```
class Center extends Point{
   constructor(x,y,r){
        super(x,y);
        this.r=r;
   }
}
```

- 链式调用

```
// bad
function Jedi(){
};
Jedi.prototype.jump = function () {
    this.jumping = true;
    return true;
};
Jedi.prototype.setHeight=function(height)
{
    this.height = height;
};

const luke = new Jedi();
luke.jump();
luke.setHeight(20);
alert(luke.jumping);    //true
alert(luke.height);     //20
```

参考

```
// good
class Jedi2 {
    jump() {
       this.jumping = true;
       return this;
    }

    setHeight(height) {
        this.height = height;
        return this;
    }
}

const luke2 = new Jedi2();

luke2.jump()
    .setHeight(22);

alert(luke2.jumping);    //true
alert(luke2.height);     //22
```

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) 。

```
function addNum(x) {
    var f = function(y) {
       return x + y;
    }
    return f;
}
var add1 = addNum(1);
var add2 = addNum(3);
alert(add1(4)); //5  （1+4）
alert(add2(6)); //9  (3+6)
```

生成新函数时，系统会通过闭包保存函数内部使用到的所有外部函数的参数和局部变量（非全局变量），调用新函数时，将会使用到这些参数和变量的值。

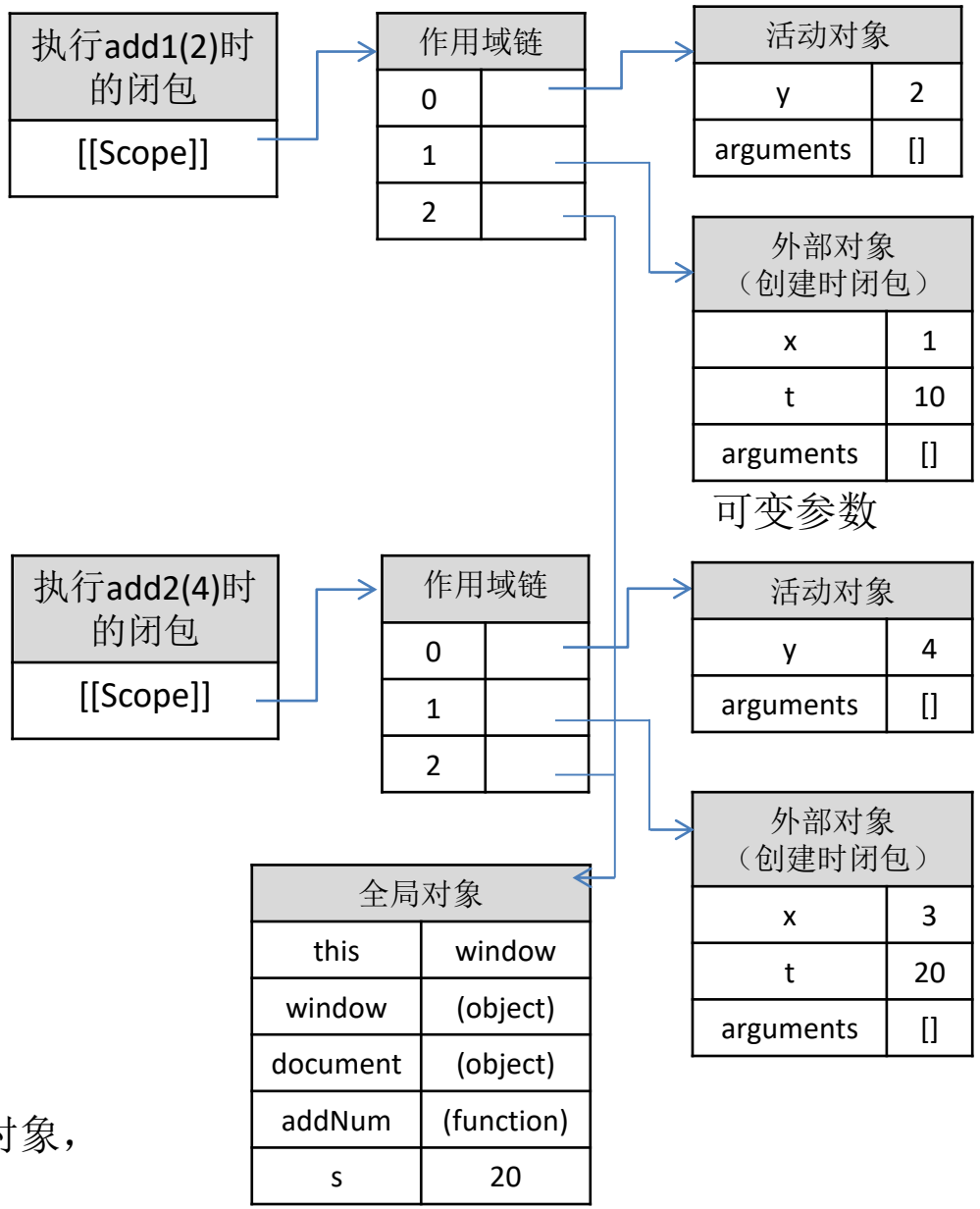
使用多层函数的情况：

```
function addNum(x) {
        function ss(y){
            return function(z) {
                return x + y + z;
             }
        }
    return ss(x+2);
}
var add1 = addNum(1);
var add2 = addNum(3);
alert(add1(4)); //8 （1+3+4）
alert(add2(6)); //14  (3+5+6)
```

使用全局变量会怎么样？

```
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



* 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。

- 产生对象时其方法也会形成闭包

```
function B(y){
   function A(x){
       this.x=x+"*";
       this.sayA=function(){alert(x+" "+y);}
   }
   var b=new A("10");
   return b;
}
var s=B("20");
var t=B("30");
s.sayA(); // 10 20
t.sayA(); // 10 30
```

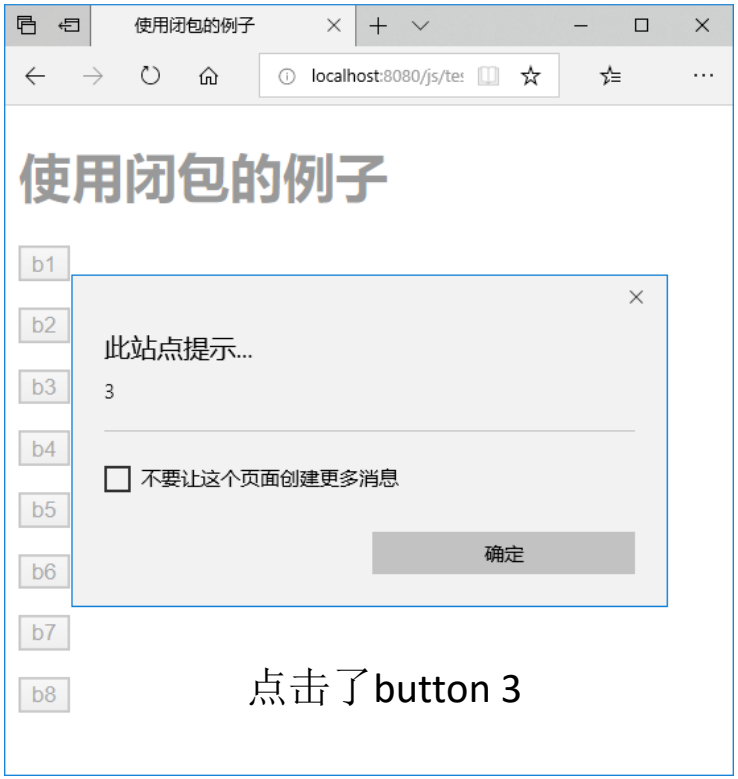
- 利用包装函数形成闭包

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

参考

- 闭包常用于把参数带入到事件处理函数中

```
<!DOCTYPE html><html class="ie" lang="en">
<head> <meta charset="UTF-8">
    <title>使用闭包的例子</title>
</head>
<body>
 <h1>使用闭包的例子</h1>
 <p><input name="btn" type="button" value="b1"/></p>
 <p><input name="btn" type="button" value="b2"/></p>
 <p><input name="btn" type="button" value="b3"/></p>
 <p><input name="btn" type="button" value="b4"/></p>
 <p><input name="btn" type="button" value="b5"/></p>
 <p><input name="btn" type="button" value="b6"/></p>
 <p><input name="btn" type="button" value="b7"/></p>
 <p><input name="btn" type="button" value="b8"/></p>

</body>
</html>
<script>
    var severalObj = document.getElementsByName("btn");
    for (var i = 0; i < severalObj.length; i++) {
        (function (ind) {
            severalObj[ind].addEventListener("click", function () { alert(ind + 1); })
        })(i);
    }
</script>
```



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; 推荐使用
alert(a.length);                        //0
alert(b.length);                        //2
alert(c.length);                        //3
alert(Arrays.isArray(a))                //true
d[3] = "Saab";                          //最好用push方法增加元素
for (var x in mycars){
   document.write(mycars[x] + "<br />")
}
```

- 方法

```
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 返回一个字符串
```

```
alert(colors.push("yellow","brown"));     //5. red,blue,green,yellow,brown
alert(colors.pop());                      //brown. red,blue,green,yellow
alert(colors.shift());                    //red. blue,green,yellow
alert(colors.unshift("brown"));           //4.   brown,blue,green,yellow
alert(colors.sort();                      //blue,brown,green,yellow
alert(colors.reverse());                  //yellow,green,brown,blue
alert(colors.join(";"));                  //yellow;green;brown;blue
alert(colors.slice(1,3));                 //green,brown
alert(colors.splice(1,3,"red","gray"));   //green,brown,blue yellow red gray
```

* 绿色为返回值，橙色为colors的值。
* 只有绿色表示color的值没变。只有橙色表示color的值变了并和返回值一样
* 数组方法的说明：

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。*推荐用于增加元素*
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。
sort	对数组元素进行排序
slice(start,end)	从数组中选出一组连续的元素
splice	删除并添加元素，返回删除的元素. arr.splice(index, many, o1,o2, ...)
toString	把数组转换为字符串，用逗号分隔。同valueOf()。不加方法时默认的方法

• 数组元素的遍历

Array的遍历方法对每个元素执行一次回调函数， forEach类似for循环，map方法返回有函数返回值构成的新数组，filter得到由结果为true的元素构成的新数组。

some方法要求有一个计算结果为true则返回true，every方法要求每个计算结果都返回true才返回true。reduce和reduceRight（从右边开始）的返回值用于下次调用，find和findIndex用于查找。

```
var colors = ["red","green","blue","yellow","brown","gray","purple"];
/* forEach 参数：(匿名)回调函数，用于遍历数组的所有元素，返回：新元素数组。*/
colors.forEach(function(value,index,fullArray){
   prn(value+" is "+(index+1)+"th color of "+fullArray.length+" colors");
}); //回调函数的参数：当前元素的[，数组下标[，当前数组]]

/* map: 由回调函数遍历每个数组元素的返回值构成一个新数组 */
var oneCharColors = colors.map(function(value, index, fullArray) {
                return value.charAt(0);});
prn(oneCharColors.join(","));   // r,g,b,y,b,g,p
```

```
/* filter: 由回调函数返回值为true的原数组元素构成一个新数组 */
var filterColors = colors.filter((value, index, fullArray)=>{
              return value.indexOf("e") >= 0;});
prn(filterColors.join(","));   // red,green,blue,yellow,purple

/* reduce: 回调函数返回的值作为下一次调用的previousValue */
var sum = [0, 1, 2, 3, 4].reduce(function(previousValue, currentValue,
          index, array) { return previousValue + currentValue; });
prn(sum);                          // 10


/* find,findIndex: 查找返回true的第一个元素的值(find)或下标(findIndex) */
prn([1,21,33,44,75].find((value,index,arr)=>{return value>30 }));      // 33
prn([1,21,33,44,75].findIndex((value,index,arr)=>{return value>=21})); // 1


/* every: 每个数组元素都返回true，结果才为true */
var allTwoOrMoreChars=colors.every(function(value,index,fullArray){
   return value.length>2;
});
prn(allTwoOrMoreChars);                    // true
```

```
/* some: 某个数组元素返回true，结果就为true */
function findLongName(value,index,fullArray){ return value.length>=7;}
prn(colors.some(findLongName));          // true

/* map: 对数组每个元素进行递归调用，arguments.callee为调用函数 */
var s=[1,2,3,4,5].map(function(n){return(n<=1)?1:arguments.callee(n-
1)*n;});
prn(s);                              // 1 2 6 24 120

// lambda表达式（箭头函数），见附录
const sum = [1,2,3,4,5,6,7,8].reduce((total, num)=>total + num, 1); //37
prn(s.some(item=>item===2));    // true. 单语句(省略了return)



/* 链式调用 */
prn(colors.filter(c=>c.length>6).map(c=>c.toUpperCase())); //
YELLOW,PURPLE
```

• 扩展运算符

数组的扩展运算符[...]和Array.from()都可以把对象转换为数组。

```
function prn(...args){              // 可变参数的推荐方法，不建议用arguments
   for(var i=0;i<args.length;i++){
       document.write(args[i]+" ");
   }
}
var s = 4;
var e = [s,5,6];                    // 用变量作为数组元素
var f = [...new Set([7,8,9,9])];    // 7,8,9。Set去重，再变成参数，最后变成数组

prn(e);                             // 4 5 6。运算符...把数组变为(可变)参数
prn(...[e,f]);                      // 4,5,6 7,8,9。运算符...把数组变为(可变)参数

const foo = document.querySelectorAll('p');
const nodes = Array.from(foo);     // from把任何对象变为数组。[p1,p2,p3]
prn(...nodes);

var foo2=["a","b","c"];
var bar=value=>value+"1";
const baz = [...foo2].map(bar);      // 不推荐
prn(baz);                            // a1,b1,c1
const baz2 = Array.from(foo2, bar); // 推荐
prn(baz2);                           // a1,b1,c1
const [first, second] = baz;         // 头两个数组元素
prn(first,second);                   // a1,b1
```

```
<!DOCTYPE  html>
<html lang="zh-cn">
   <head>
      <meta charset="utf-8"/>
      <title>This is my title</title>
      <script>...  </script>
</head>
   <body>
      <h1>This is header</h1>
      <p>p1</p>
      <p>p2</p>
      <p>p3</p>
   </body>
</html>
```

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、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3、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4、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()
```

附录5、lambda表达式

- 用lambda表达式定义函数

一般格式：(参数1, 参数2, ...)=>{语句1; 语句2; ...};
无参数：()=>{语句1;语句2,...};
单参数：参数1=>{语句1;语句2,...};　　省略了参数括号
单语句：参数1=>语句1;　　省略了语句括号
单retrun：参数1=>表达式1;　　省略了return

例子：

```
var test1 = (x, y) => {x = x * 2 + y; alert(x)};
test1(1, 3);                                    // 5
//等同  function test1(x,y){
//          x = x * 2 + y; alert(x);
//      }
var test2 = x => {x = x * 2; alert(x)};
test2(5);                                       // 10
var test3 = () => alert(100);
test3();                                        // 100
var test4 = x => x * 200;
alert(test4(2));                                // 400
```

附录6、函数式编程

参考

- 函数式编程要求使用纯函数，即不依赖外部环境的状态，没有任何副作用，对于相同的输入会得到相同的输出。

```
function prn() { for (var i = 0; i < arguments.length; i++)
                        document.write(arguments[i] + "<br>"); }
var arr = [1, 2, 3, 4, 5];

prn(arr.splice(0, 3)); // [1,2,3]   Array.splice不是纯函数
prn(arr.splice(0, 3)); // [4,5]

prn(arr.slice(0, 3));  // [1,2,3]   Array.slice是纯函数
prn(arr.slice(0, 3));  // [1,2,3]

var min = 18;
var checkAge1 = age => age > min;          // 不是纯函数
prn(checkAge1(20));                        // true
min = 30;
prn(checkAge1(20));                        // false

var checkAge2 = age => age > 18;           // 纯函数
prn(checkAge2(20));                        // true
prn(checkAge2(20));                        // true
```

采用纯函数的好处之一是可以**缓存**上一次计算的结果，如果下次调用的参数相同，则可以使用缓存值，不用再次计算。

- 函数式编程另一个好处可以通过函数**柯里化**（curry）简化程序，即，传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数，这也是一种惰性计算。

```
var checkAge = (age, min) => age > min;
var checkAge = min => (age => age > min); // 对上面的函数柯里化
var isAdult = checkAge(18);
alert(isAdult(16));                       // false
alert(isAdult(20));                       // true
alert(checkAge(60)(50));                  // false
```

- 为了去除多重调用g(f(x))，JS可以采用组合调用方式compose(g,f)(x)。

```
//两个函数的组合
var compose = (f,g) => (x => f(g(x)));
var dbl = x => x * 2;
var inc = x => x + 1;
alert(compose(dbl,inc)(8));
```

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、Json的数据格式
- 附录2、对象属性的特性
- 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、lambda表达式
- 附录6、函数式编程