JS高级程序设计3rd阅读笔记

第一章 在HTML中使用JavaScript

1.1 <script></script>

<body>  
 <script type="text/javascript">  
 **function** sayScript()  
 {  
 alert("</script>");

<!-- </script>解析到这里就结束了，所以后面的都会在浏览器输出 -->

}  
 sayScript();  
 </script>  
</body>

1.2 script标签的位置

<body>  
  
  
 *<!--这里放内容，把JS文件放在最底下，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" src="function.js"></script>  
</body>

1.3 延迟脚本

--------------------------------------------------------------------------------

.html

--------------------------------------------------------------------------------

<body>  
  
 *<!--这里放内容，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" defer="defer" src="function.js"></script>  
 <script>  
 alert("111");  
 </script>  
</body>

--------------------------------------------------------------------------------

function.js

--------------------------------------------------------------------------------

alert("222");

--------------------------------------------------------------------------------

注：defer="defer"：延迟脚本文件的执行，使其遇到</html>标签后再执行，延迟脚本按照它们指定的顺序执行。

使用外部JS文件时不能使用document.write()，Chrome浏览器提示：

Failed to execute 'write' on 'Document': It isn't possible to write into a document from an asynchronously-loaded external script unless it is explicitly opened.

function.js:5 Failed to execute 'write' on 'Document': It isn't possible to write into a document from an asynchronously-loaded external script unless it is explicitly opened.

所以这里先执行alert("111");后执行alert("222");去掉defer属性，执行顺序相反。

1.4 异步脚本

<body>  
  
  
 *<!--这里放内容，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" async="async" src="function.js"></script>  
</body>

注：async="async"：不让页面等待脚本的下载和执行，从而异步加载页面其他内容，虽然不必等待其他脚本，也不会阻塞文档呈现，但不能保证它们按照页面中出现的顺序执行。

使用外部JS文件时不能使用document.write()，Chrome浏览器提示：

Failed to execute 'write' on 'Document': It isn't possible to write into a document from an asynchronously-loaded external script unless it is explicitly opened.

function.js:5 Failed to execute 'write' on 'Document': It isn't possible to write into a document from an asynchronously-loaded external script unless it is explicitly opened.

1.5 兼容XHTML（嵌入代码，非外部JS文件）

<body>  
  
 *<!--(<)号在HTML中会被解析为开始一个新的标签，如<body>中的(<)-->* <script type="text/javascript">  
 //<![CDATA[  
 **function** compare(a,b)  
 {  
 **if**(a<b) //如果不使用CDATA片段，这个<号在XHTML中不会被解析为小于号  
 {  
 a = b;  
 }  
 }  
 //]]  
 </script>  
 *<!--这里放内容，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" async="async" src="function.js"></script>  
</body>

1.6 不推荐的使用语法（嵌入代码，非外部JS文件）

<body>  
  
 *<!--(<)号在HTML中会被解析为开始一个新的标签，如<body>中的< -->* <script type="text/javascript">  
 *<!--* **function** compare(a,b)  
 {  
 **if**(a<b) //如果不使用CDATA片段，这个<号在XHTML中不会被解析为小于号  
 {  
 alert(a);  
 }  
 }  
  
 compare(1,2);  
 //-->  
 </script>  
 *<!--这里放内容，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" async="async" src="function.js"></script>  
</body>

--------------------------------------------------------------------------------

注：所有浏览器都支持了JavaScript脚本，没必要这么做了，相反，这种注释会使脚本在XHTML中被忽略。

1.7 <noscript>标签

<body>  
 <noscript>  
 <p>浏览器不支持JS脚本</p>  
 </noscript>  
 *<!--这里放内容，让JS文件最后加载，保证页面优先加载-->* <script type="text/javascript" async="async" src="function.js"></script>  
</body>

--------------------------------------------------------------------------------

注： 1.浏览器不支持脚本

2.虽然支持脚本，但浏览器禁用了脚本

<noscript>只要满足以上任何一个要求，都会执行标签内的内容。

第二章 基本概念

2.1 标识符

1.第一个字母必须是字母、下划线(\_)、或一个美元符号($)  
2.其他字符可以是字母、下划线、美元符号或数字  
3.驼峰大小写格式： myFirst，yourCar

4.不能把关键字、保留值、布尔值和null用作标识符

2.2 typeof操作符

**var** str = "string";  
**var** fun = **function**(){};  
  
alert(**typeof null**); //object，null被认为是空的对象的引用  
alert(**typeof** 95); //number  
alert(**typeof** str); //string  
alert(**typeof** fun); //function

2.3 undefined、null类型

//声明但却未初始化的变量的默认值是undefined  
//引入undefined是为了区分空对象指针null和未初始化的变量  
//如果养成了声明并初始化的习惯，那么就可以通过typedef来检测变量有没有初始化  
**var** age;  
alert(**typeof** age); //undefined，表明没有初始化  
//如果是没有声明的变量  
alert(**typeof** ame); //undefined，表明没有声明  
alert(**typeof** name);//string ?? 为什么是string，根本没声明过!  
  
  
**var** car = **null**; //一个空对象,建立对象时最好初始化null  
alert(**typeof** car); //object  
**if**(car != **null**)  
{  
 //确保对象非空，然后执行操作  
}  
  
//其实undefined派生自null  
//在ECMAScript第三版本中引入undefined来区分空对象指针和未经初始化的变量  
alert(**null** == undefined); //true  
alert(**null** === undefined);//false

2.4 Boolean类型

------------------------------------------------------------------  
 数据类型 **true false**------------------------------------------------------------------  
 Boolean **true false** String 非空字符 ""  
 Number 任何非零数值(包括无穷大) 0和NaN  
 Object 任何对象 **null** Undefined undefined  
------------------------------------------------------------------  
//布尔值自动类型转换  
**var** a = NaN;  
  
**if**(!a)  
{  
 alert(**false**); //false  
}  
  
**var** b = {};  
  
**if**(b)  
{  
 alert(**true**); //true  
}

2.5 Number类型

//浮点数值的最高精度是17位  
**var** a = 0.1; //存储的二进制数并不能精确的表示0.1  
**var** b = 0.2; //  
  
alert(a+b); //0.30000000000000004  
  
**if**((a + b) == 0.3) {   
 //  
}  
  
**else**{  
 alert("here"); //执行了这条代码  
}

2.6 Number、parseInt and parseFloat

*/\*\*  
 \* function Number(value) {}  
 \* 功能：转换为数字  
 \* 参数：value ——> 可以传入任何形式  
 \*/*document.write(Number(**true**)+"<br/>"); //1  
document.write(Number(1)+"<br/>"); //1  
document.write(Number(undefined)+"<br/>"); //NaN  
document.write(Number("000011")+"<br/>"); //11  
document.write(Number("0x0B")+"<br/>"); //11  
document.write(Number("")+"<br/>"); //0  
  
//其余任何形式都是NaN  
document.write(Number("abc")+"<br/>"); //NaN  
**var** a = {};  
document.write(Number(a)+"<br/>"); //NaN  
document.write("<br/>");  
  
*/\*\*  
 \* parseInt = function(s,radix) {};  
 \* 功能：转换为整型数，规则更合理  
 \* 参数：1. s ——> 可以传入任何形式  
 \* 2. radix ——> 表明需要转换的进制数  
 \* 说明：第2参数可选，默认转换为十进制数  
 \*/*document.write(parseInt(**true**)+"<br/>"); //NaN  
document.write(parseInt(1)+"<br/>"); //1  
document.write(parseInt(undefined)+"<br/>"); //NaN  
document.write(parseInt("000011")+"<br/>"); //11  
document.write(parseInt("0x0B")+"<br/>"); //11  
document.write(parseInt("")+"<br/>"); //NaN  
  
document.write("<br/>");  
document.write(parseInt("1234acd")+"<br/>"); //1234  
document.write(parseInt("")+"<br/>"); //NaN  
document.write(parseInt("0xA")+"<br/>"); //10  
document.write(parseInt(22.5)+"<br/>"); //22  
document.write(parseInt("070")+"<br/>"); //70 ECMAScript5  
//document.write(parseInt("070")+"<br/>"); //56(八进制) ECMAScript3  
  
document.write(parseInt("070",8)+"<br/>"); //56(八进制)  
document.write(parseInt("070",10)+"<br/>"); //70(十进制)  
document.write(parseInt("AF",16)+"<br/>"); //175(十六进制)  
document.write(parseInt("AF")+"<br/>"); //NaN  
  
*/\*\*  
 \* parseFloat = function(x) {};  
 \* 功能：转换为浮点数  
 \* 参数：x ——> 可以传入任何形式  
 \* 说明：该函数只转换十进制数，始终忽略前导的0  
 \*/*document.write("<br/>");  
document.write("<br/>");  
document.write(parseFloat(**true**)+"<br/>"); //NaN  
document.write(parseFloat(1)+"<br/>"); //1  
document.write(parseFloat(undefined)+"<br/>"); //NaN  
document.write(parseFloat("000011")+"<br/>"); //11  
document.write(parseFloat("")+"<br/>"); //NaN  
  
document.write(parseFloat("0x0B")+"<br/>"); //0(不解析十六进制数，始终为0)  
document.write(parseFloat("1234blue")+"<br/>"); //1234(整数)  
document.write(parseFloat("22.5")+"<br/>"); //22.5  
document.write(parseFloat("22.5.6.8")+"<br/>"); //22.5  
document.write(parseFloat("0908.5")+"<br/>"); //908.5  
document.write(parseFloat("3.125e7")+"<br/>"); //31250000

2.7 String类型

*/\*\*  
 \* Number.prototype.toString = function(radix) {};  
 \* 功能：转换为字符串  
 \* 参数：radix ——> 需要转换为radix进制的字符串，默认为10进制  
 \*/***var** num = 10;  
document.write(num.toString()+"<br/>"); //10  
document.write(num.toString(2)+"<br/>"); //1010  
document.write(num.toString(8)+"<br/>"); //12  
document.write(num.toString(10)+"<br/>"); //10  
document.write(num.toString(16)+"<br/>"); //a  
document.write("<br/>");  
  
//null和undefined没有toString的方法，因为不是Number对象  
  
*/\*\*  
 \* function String(value) {}  
 \* 功能：转换为字符串  
 \* 说明：1.如果有toString的方法，先调用没有参数的该方法  
 \* 2.如果是null，返回"null"  
 \* 3.如果是undefined，返回"undefined"  
 \*/*document.write(String(10)+"<br/>"); //10  
document.write(String(**true**)+"<br/>"); //true  
document.write(String(**null**)+"<br/>"); //null  
document.write(String(undefined)+"<br/>"); //undefined

2.8 布尔操作符

/\*逻辑非\*/  
**var** oPerson = {};  
document.write(!oPerson+"<br/>"); //false  
document.write(!""+"<br/>"); //true  
document.write(!"非空"+"<br/>"); //false  
document.write(!0+"<br/>"); //true  
document.write(!3+"<br/>"); //false  
document.write(!Infinity+"<br/>"); //false  
document.write(!**null**+"<br/>"); //true  
document.write(!NaN+"<br/>"); //true  
document.write(!undefined+"<br/>"); //true  
  
/\*逻辑与\*/  
//var found = true;  
//var result = (found && someUndefinedVariable);  
//Uncaught ReferenceError: someUndefinedVariable is not defined  
//alert(result);  
**var** found = **false**;  
**var** result = (found && someUndefinedVariable);  
alert(result); //false  
  
/\*逻辑或\*/  
//var found = false;  
//var result = (found || someUndefinedVariable);  
//Uncaught ReferenceError: someUndefinedVariable is not defined  
//alert(result);  
**var** bFound = **true**;  
**var** result = (bFound || someUndefinedVariable);  
alert(result); //true

2.9 乘性、加减操作符

/\*乘法\*/  
**var** num = NaN \* 3;  
document.write(num+"<br/>"); //NaN  
num = Infinity \* 0;  
document.write(num+"<br/>"); //NaN  
num = Infinity \* 4;  
document.write(num+"<br/>"); //Infinity  
num = Infinity \* -4;  
document.write(num+"<br/>"); //Infinity  
num = Infinity \* Infinity;  
document.write(num+"<br/>"); //Infinity  
num = "3" \* 4;  
document.write(num+"<br/>"); //12  
/\*除法\*/  
num = 3 / NaN;  
document.write(num+"<br/>"); //NaN  
num = NaN / 3;  
document.write(num+"<br/>"); //NaN  
num = Infinity / Infinity;  
document.write(num+"<br/>"); //NaN  
num = 4 / 0;  
document.write(num+"<br/>"); //Infinity  
num = 4 / -0;  
document.write(num+"<br/>"); //-Infinity  
num = Infinity / 4;  
document.write(num+"<br/>"); //Infinity  
num = "12" / 4;  
document.write(num+"<br/>"); //3  
/\*模除\*/  
num = Infinity % 4;  
document.write(num+"<br/>"); //NaN  
num = 4 % 0;  
document.write(num+"<br/>"); //NaN  
num = Infinity % Infinity;  
document.write(num+"<br/>"); //NaN  
num = 55 % Infinity;  
document.write(num+"<br/>"); //55  
num = 0 % 0;  
document.write(num+"<br/>"); //NaN  
num = "12" % 25;  
document.write(num+"<br/>"); //12  
/\*加法\*/  
**var** num = 5;  
**var** message = "Two num add is " + num + num;  
document.write(message+"<br/>"); //Two num add is 55  
**var** message = "Two num add is " + (num + num);  
document.write(message+"<br/>"); //Two num add is 10  
/\*减法\*/  
**var** result = 5 - **true**;  
document.write(result+"<br/>"); //4  
result = NaN - 1;  
document.write(result+"<br/>"); //NaN  
result = 5 - **null**;  
document.write(result+"<br/>"); //5  
result = 5 - "";  
document.write(result+"<br/>"); //5  
result = 5 - "2";  
document.write(result+"<br/>"); //3

2.10 关系操作符

/\*<|>\*/  
**var** result = "23" < 3;  
document.write(result + "<br/>"); //false  
result = "23" < "3"  
document.write(result + "<br/>"); //true  
result = "a" < 3;  
document.write(result + "<br/>"); //false,因为a转换为NaN  
result = NaN < 3;  
document.write(result + "<br/>"); //false  
result = NaN >= 3;  
document.write(result + "<br/>"); //false  
document.write("<br/>");   
  
  
/\*==|!=|===|!==\*/ //=== 严格等于或全等于不进行类型转换  
result = (**null** == undefined);  
document.write(result + "<br/>"); //true  
result = (**null** === undefined);  
document.write(result + "<br/>"); //false  
result = (NaN == NaN);  
document.write(result + "<br/>"); //false  
result = (undefined == 0);  
document.write(result + "<br/>"); //false  
result = (**null** == 0);  
document.write(result + "<br/>"); //false  
result = ("55" == 55);  
document.write(result + "<br/>"); //true  
result = ("55" === 55);  
document.write(result + "<br/>"); //false  
  
/\*,\*/  
result = (1,2,3,4,5,6);  
document.write(result + "<br/>"); //6

第三章 变量、作用域和内存问题

3.1 基本类型和引用类型的复制区别

/\*基本类型(null undefined Number String Boolean)复制\*/  
**var** num1 = 5;  
**var** num2 = num1;  
num1 = 10;  
alert(num2); //5  
  
/\*  
  
复制前  
  
-------------------------  
 |  
-------------------------  
 |  
-------------------------  
 num1 | 5(Number类型)  
-------------------------  
  
复制后  
  
-------------------------  
 |  
-------------------------  
 num2 | 5(Number类型)  
-------------------------  
 num1 | 5(Number类型)  
-------------------------  
  
注：两个不同的内存空间，相互之间的变量互不影响  
  
 \*/  
  
/\*引用类型(Object类型、Array类型、Date类型、Function类型、RegExp类型)复制\*/  
**var** oPerson = {};  
**var** oPerson1 = oPerson;  
oPerson.Name = "ZHUXIANKANG";  
alert(oPerson1.Name); //ZHUXIANKNG  
  
/\*  
  
 复制前  
  
-------------------------  
 |  
-------------------------  
 |  
------------------------- 指向了堆中的一个对象  
 oPerson | (Object类型) ---------------------  
------------------------- |  
 |  
 复制后 Object  
 |  
------------------------- |  
 | |  
-------------------------指向了堆中的同一个对象 |  
 oPerson1 | (Object类型) ---------------------  
------------------------- |  
 oPerson | (Object类型) ---------------------  
-------------------------  
  
 注：复制时，其实内存中存放的是指向对象的指针，两个变量实际引用的是同一个对象  
  
 所以在进行函数传递参数时要注意，进行类型引用的参数传递时，传递的是指向对象的  
  
 指针地址，如果函数内的局部变量发生了变化，会反映到函数的外部。  
  
 \*/  
  
**function** setName(obj)  
{  
 obj.Name = "JavaScript";  
}  
  
**var** oPerson = **new** Object();  
setName(oPerson);  
alert(oPerson.Name); //JavaScript

3.2 类型检测

/\*基本类型检测\*/  
/\*null undefined Number String Boolean\*/  
**var** s = "1235";  
**var** b = **true**;  
**var** c = 22;  
**var** u;  
**var** n = **null**;  
//var o = new Object();  
  
document.write(**typeof** s + "<br/>"); //string  
document.write(**typeof** b + "<br/>"); //boolean  
document.write(**typeof** c + "<br/>"); //number  
document.write(**typeof** u + "<br/>"); //undefined  
document.write(**typeof** n + "<br/>"); //object 空指针对象  
//document.write(typeof o + "<br/>"); //object  
  
  
/\*引用类型检测\*/  
/\*Object类型、Array类型、Date类型、Function类型、RegExp类型\*/  
/\*result = variable instanceof caonstructor\*/  
**var** oDate = **new** Date();  
**var** oObject = **new** Object();  
**var** oArray = **new** Array();  
**var** oFunction = **new** Function();  
**var** oRegExp = **new** RegExp();  
  
document.write((oDate **instanceof** Date) + "<br/>"); //true  
document.write((oObject **instanceof** Object) + "<br/>"); //true  
document.write((oArray **instanceof** Array) + "<br/>"); //true  
document.write((oFunction **instanceof** Function) + "<br/>"); //true  
document.write((oFunction **instanceof** Array) + "<br/>"); //false  
document.write((oRegExp **instanceof** RegExp) + "<br/>"); //true  
  
/\*引用类型的值是保存在内存中的对象，Object是所有对象的基础\*/  
/\*所有对象都有Object的属性和方法\*/  
document.write((oDate **instanceof** Object) + "<br/>"); //true  
document.write((oObject **instanceof** Object) + "<br/>"); //true  
document.write((oArray **instanceof** Object) + "<br/>"); //true  
document.write((oFunction **instanceof** Object) + "<br/>"); //true  
document.write((oFunction **instanceof** Object) + "<br/>"); //true  
document.write((oRegExp **instanceof** Object) + "<br/>"); //true

3.3 作用域链

/\*全局执行环境是Window对象，所以所有的全局变量和函数都是作为Window对象的属性和方法创建的\*/  
**var** color = "red"; //window对象的属性 color  
alert(window.color); // red  
  
**function** alertColor() //window对象的方法 alertColor  
{  
 alert(window.color);  
}  
  
window.alertColor(); //red  
  
  
/\*每个执行环境都有一个与之关联的变量对象  
\* 这个环境中所定义的所有变量和函数都保存在这个对象中  
\* 执行环境中的所有代码执行完毕后，环境被销毁  
\* 全局执行环境直到关闭网页或浏览器才会被销毁\*/  
  
/\*当代码在环境中执行时，会创建变量对象的作用域链  
\* 作用域链的前端始终是当前执行代码所在环境的变量对象  
\* 如果这个环境是对象，则将其活动对象作为变量对象  
\* 活动对象在最开始时只包含一个变量，即arguments对象  
\* 这个对象在全局环境中是不存在的  
\* 作用域链的下一个变量对象来自包含该当前执行代码所在环境变量对象的环境  
\* 一直延续到全局执行环境，所以window始终是作用域链中的最后一个对象\*/  
  
**var** color = "blue"; //window对象  
   
**function** changeColor() { //changeColor对象，也是window对象的方法  
 **var** anotherColor = "red";  
  
 **function** anotherChangeColor(){ //anotherChangeColor对象，也是changeColor对象的方法  
 **var** tempColor;  
 tempColor = color;  
 color = anotherColor;  
 anotherColor = tempColor;  
 }  
  
 anotherChangeColor();  
}  
  
changeColor();

/\*anotherChangeColor执行环境中，作用域链包含三个对象

它自己的变量对象(包括arguments对象)（局部环境）、changeColor对象（局部环境）、

window对象（全局执行环境）\*/

作用域链

window

|

|--------color

|--------changeColor()

|

|--------anotherColor

|--------anotherChangeColor()

|

|---------tempColor

注意：函数参数也被当作变量来对待，访问规则与执行环境中的其他变量相同

/\*延长作用域链\*/

1.try-catch语句的catch块

2.with语句

With语句会将指定对象添加到作用域链中

**function** buildUrl(){  
 **var** qs = "?debug=true";  
 **with**(location){  
 **var** url = href + qs;  
 }  
  
 **return** url;  
}  
  
document.write(buildUrl());  
//http://localhost:63342/%E7%A4%BA%E4%BE%8B%E4%BB%A3%E7%A0%81/index.html?debug=true

//其实href实际是引用了location.href

//使用with将location对象的所有属性和方法添加到了当前执行环境的作用域链的前端

3.4 管理内存

/\*垃圾回收机制\*/  
/\*标记清除和引用计数\*/  
/\*引用计数中如果有循环引用，容易造成内存泄漏\*/

/\*优化内存占用的最佳方式是一旦数据不再有用，通过将其设置为null来释放引用

这个做法叫做解除引用，适用于大多数的全局变量和全局对象的属性

局部变量会在离开执行环境时被自动解除引用\*/

/\*解除引用的真正作用是让值脱离执行环境，以便垃圾收集器下次运行时将其回收\*/

1.基本类型值在内存中占据固定大小的空间，保存在栈内存中

2.引用类型的值是对象，保存在堆内存中

3.包含引用类型值得变量实际上包含的并不是对象本身，而是一个指向该对象的指针

4.从一个变量向另一个变量复制引用类型的值，复制的其实是指针，两个变量最终都指向同一个对象

5.离开作用域的值将被自动标记为可以回收，因此将在垃圾收集期间被删除

6.标记清除是目前主流的垃圾收集算法，这种算法的思想是给当前不使用的值加上标记，然后回收其内存

7.引用计数是跟踪记录所有值被引用的次数

第四章 引用类型

4.1 Object类型

**function** displayName(oArg){  
 **var** output = "";  
  
 **if**(**typeof** oArg.age == "number"){  
 output = oArg.age + " ";  
 }  
  
 **if**(**typeof** oArg.name == "string"){  
 output = output + oArg.name;  
 }  
  
 alert(output);  
}  
  
//这种模式适用于向函数传入大量的可选的参数的情形，使用对象字面量来封装多个可选的参数  
displayName({ name:"zxk",  
 age:29  
});  
  
displayName({  
 name:"vitco"  
});

注：创建object实例的方法 1.var oPerson = new Object();

2.var oPerson = {name: , age: };

3.var oPerson = {};

属性访问的方式： 1.点表示法

2.方括号表示法（可以用于遍历属性、可以通过变量来访问属性、点表示法中不能表示的容易导致错误的字符或属性名中使用的是关键字或保留字）

4.2 Array类型

创建Array实例的方法 1. **var** colors = **new** Array();  
**var** colors = **new** Array(20);  
**var** colors = **new** Array("red","blue","yellow");

2. **var** name = [];

**var** value = [1,2,3];

/\*数组索引\*/  
**var** colors = ["red","yellow","green"];  
colors[3] = "brown"; //新增第四项  
alert(colors[3]); //brown  
  
/\*数组长度属性：length\*/  
**var** colors = ["red","yellow", "green"];  
colors.length = 2;  
alert(colors[2]); //undefined，移除了第三项green  
  
colors[colors.length] = "red"; //在最后添加一项,数组最后一项的索引始终是length-1，所以新添加的最后一项是length  
colors[colors.length] = "black"; //在最后添加另一项  
document.write(colors[2] + "<br/>"); //red  
document.write(colors[3] + "<br/>"); //black  
  
/\*检测数组\*/  
**var** arr = [];  
  
**if**(arr **instanceof** Array) {  
 alert("is array");  
}  
  
//有缺陷，instanceof假定只有一个全局执行环境  
//如果包含多个框架，实际上就有两个以上不同的全局执行环境  
//因为Array是window的属性  
//从一个框架中传入另一个框架中的数组，在第二个框架中使用instanceof，会判断为不是数组  
  
**if**(Array.isArray(arr)) {  
 alert("is array");  
}  
  
//有缺陷，浏览器支持问题  
//重写isArray可以在尚未支持Array.isArray的浏览器中使用  
**function** isArray(value) {  
 **return** Object.prototype.toString.call(value) == "[object Array]";  
}  
  
**if**(isArray(arr)) {  
 alert("is array");  
}  
  
  
//更完美的写法  
**function** isArray(value) {  
  
 **if**(Array.isArray) {  
 **return** Array.isArray(value);  
 } **else** {  
 **return** Object.prototype.toString.call(value) == "[object Array]";  
 }  
}  
  
**if**(isArray(arr))  
{  
 alert("isArray");  
}

/\*数组转换成字符串\*/  
**var** colors = ["red","blue","yellow"];  
  
document.write(colors.toString() + "<br/>");//red,blue,yellow 下面都是这个结果  
document.write(colors.valueOf() + "<br/>");//返回数组，调用数组每一项的toString()方法  
alert(colors); //在后台调用数组每一项的toString()方法  
  
**var** person1 = {  
 toString: **function**() {  
 **return** "person1 toString";  
 },  
  
 toLocaleString: **function**() {  
 **return** "person1 toLocaleString";  
 }  
};  
  
**var** person2 = {  
 toString: **function**() {  
 **return** "person2 toString";  
 },  
  
 toLocaleString: **function**() {  
 **return** "person2 toLocaleString";  
 }  
}  
  
**var** person = [person1,person2];  
alert(person); //person1 toString，person2 toString  
alert(person.toString()); //person1 toString，person2 toString  
alert(person.toLocaleString()); //person1 toLocaleString，person2 toLocaleString  
  
alert(person.join());  
alert(person.join(undefined));  
alert(person.join(",")); //以上三个结果一样，都是以逗号分隔，  
alert(person.join("|")); //person1 toLocaleString|person2 toLocaleString

/\*数组的栈方法\*/  
  
//栈是一种后进先出的数据结构

//数组的后端添加项，数组的后端删除项  
**var** num = **new** Array();  
**var** count = num.push(1,2,3); //返回数组长度，推入3个数  
alert(count);  
**var** item = num.pop(); //推出最后一个数3,返回的是最后一个推出的数  
alert(item); //3  
alert(num); //1,2  
  
num.push(3);  
num.push(4);  
alert(num); //1,2,3,4  
num.pop();  
alert(num); //1,2,3

/\*数组的队列方法\*/  
  
//队列数据结构的访问规则是先进先出  
//数组的后端添加项，数组的前端删除项  
**var** num = **new** Array();  
**var** count = num.push(1,2,3); //3  
**var** item = num.shift(); //取得第一项，返回第一项的值  
alert(num); //2,3 推出了1  
  
num.push(4,5,6);  
num.shift();  
alert(num); //3,4,5,6 推出了2

//数组的前段添加项，数组的末端移除项  
**var** num = **new** Array();  
**var** count = num.unshift(1,2,3); //在前端屠推入3项  
alert(num); //1,2,3 注意不是3，2，1  
  
num.pop();  
alert(num); //1,2  
  
num.unshift(3);  
alert(num); //3,1,2

/\*数组的排序\*/  
  
**var** num = [1,2,3,4,5,6];  
num.reverse(); //反转数组  
alert(num); //6,5,4,3,2,1  
  
num = [0,1,2,5,10,15];  
num.sort(); //默认按升序排列，调用数组的每一项的toString方法，然后比较字符串的大小，注意是字符串  
alert(num); //0 1 10 15 2 5  
  
//sort()可以接受一个比较函数作为传入的参数  
//比较函数返回三个值，第一个参数应该位于第二个参数之前则返回一个负数  
  
**function** compare(value1,value2){ //这种比较适合大多数数据类型  
 **if**(value1 < value2){  
  } **else if**(value1 > value2){  
 **return** 1;  
 } **else** {  
 **return** 0;  
 }  
}  
  
num.sort(compare);  
alert(num); //0,1,2,5,10,15  
  
**function** compare(value1,value2){ //这种比较适合大多数数据类型  
 **if**(value1 < value2){  
 **return** 1; //value1 排在value2之后，降序排列  
 } **else if**(value1 > value2){  
 **return** -1;  
 } **else** {  
 **return** 0;  
 }  
}  
  
num.sort(compare);  
alert(num); //15，10，5，2，1，0  
  
//对于数值类型的或者其valueof()方法会返回数值类型的对象类型，可以使用简单的比较函数  
**function** compare(value1,value2){  
 **return** value2-value1;  
}  
  
num.sort(compare);  
document.write(num); //15,10,5,2,1,0

/\*数组的操作方法\*/  
  
**var** num = [1,2,3,4];  
**var** num1 = num.concat(5,[6,7]);  
  
document.write(num + "<br/>"); //1,2,3,4  
document.write(num1 + "<br/>");//1,2,3,4,5,6,7

//concat 基于当前数组的所有项创建一个新数组  
//将参数添加到新数组的末尾，没有传入参数则只是复制当前的数组

//如果传入的是值或者数组，都将添加到新数组的末尾

**var** num = [1,2,3,4,5,6];  
  
**var** num1 = num.slice(2); //从第二项到最后  
**var** num2 = num.slice(3,5); //从第三项到第四项，不包括第五项  
**var** num3 = num.slice(-3,-1); //传入的参数如果是负数，则加上数组长度  
  
document.write(num1 + "<br/>"); //3,4,5,6  
document.write(num2 + "<br/>"); //4,5  
document.write(num3 + "<br/>"); //4,5  
  
//删除  
**var** num = [1,2,3,4,5,6];  
**var** num1 = num.splice(2,2); //两个参数：要删除的起始项的位置，删除的项数  
document.write(num + "<br/>"); //1,2，5，6 剩下的项  
document.write(num1 + "<br/>"); //3,4 返回删除的项  
  
//插入  
**var** num = [1,2,3,4,5,6];  
**var** num1 = num.splice(2,0,7,8,9); //任意多个参数，第一个参数是要插入的起始位置，第二个是要删除的项为0，第三个开始是要插入的项  
document.write(num + "<br/>"); //1，2，7，8，9，3，4，5，6  
document.write(num1 + "<br/>"); //返回一个空数组  
  
//替换  
**var** num = [1,2,3,4,5,6];  
**var** num1 = num.splice(2,2,7,8,9); //任意多个参数，第一个参数是要替换的起始位置，第二个是要删除的项数，第三个开始是要插入的任意项  
document.write(num + "<br/>"); //1，2，7，8，9，5，6  
document.write(num1 + "<br/>"); //3,4 删除的是3，4

/\*数组的位置方法\*/  
  
**var** num = [1,2,3,4,5,4,3,2,1];  
  
document.write(num.indexOf(4) + "<br/>"); //3 查看数组中元素4的位置  
document.write(num.lastIndexOf(4) + "<br/>"); //5 从后向前索引的第一个元素4  
document.write(num.indexOf(4,4) + "<br/>"); //5 从第五项开始向后索引4项  
document.write(num.lastIndexOf(4,4) + "<br/>");//3 从末端向前的第五项开始索引第一个遇到的4元素  
document.write(num.indexOf(0) + "<br/>");//没有找到索引元素返回-1  
  
**var** person = {name:"zhuxiankang"};  
**var** people = [{name:"zhuxiankang"}]; //people是个数组，数组中的元素是一个对象  
**var** morePeople = [person];  
document.write(morePeople.indexOf(person) + "<br/>");//0  
document.write(people.indexOf(person) + "<br/>"); //-1

/\*数组的迭代方法\*/  
**var** num = [1,2,3,4,5,4,3,2,1];  
  
//every() 对数组中的每一项运行给定函数，如果每一项都返回true，则返回true  
//接收两个参数，要在每一项上运行的函数和运行该函数的作用域对象（可选，影响this的值）  
//运行的给定函数需要传入三个参数，数组项的值item，该项在数组中index的位置和数组对象本身array。  
  
/\*每一项都要返回true\*/  
**var** everyResult = num.every(**function**(item,index,array){  
 **return** (item > 2);  
});  
document.write(everyResult + "<br/>"); //false，返回的是布尔值  
  
/\*只要有一项返回true\*/  
**var** someResult = num.some(**function**(item,index,array){  
 **return** (item > 2);  
});  
document.write(someResult + "<br/>"); //true，返回的是布尔值  
  
/\*删选数组中的元素并返回新的数组\*/  
**var** filterArray = num.filter(**function**(item,index,array){  
 **return** (item > 2);  
});  
alert(filterArray); //3,4,5,4,3 返回的是被删选后的数组  
  
/\*对原数组的每一项进行操作，并返回新的数组\*/  
**var** mapArray = num.map(**function**(item,index,array){  
 **return** (item \* 2);  
})  
alert(mapArray);//2,4,6,8,10,8,4,6,2  
  
/\*遍历数组并操作\*/  
num.forEach(**function**(item,index,array){  
 //这个函数与执行for循环操作是一样的  
});

/\*数组的归并方法\*/  
//reduce()函数会迭代数组中的所有项，而reduceRight()则从数组的最后一项开始，向前遍历到第一项  
//这两个方法接收两个参数，一个在每一项上调用的函数和作为归并的初始值（可选）  
//调用的函数接收4个参数，前一个值，当前值，项的索引和数组对象  
//函数的返回值都会作为第一个参数（前一个值）自动传给下一项，第一次迭代发生的数组的第二项上  
  
**var** num = [1,2,3,4,5];  
**var** sum = num.reduce(**function**(pre,crr,index,Array){  
 **if**(index > 2) {  
 **return** (pre + crr);  
 }**else**{  
 **return** 0;  
 }  
});  
  
alert(sum); //9  
alert(num);  
//第一次 pre=4，crr=5  
  
  
sum = num.reduceRight(**function**(pre,crr,index,Array){ //从最后一项向前  
 alert(pre); // 5 9 12 14 15  
 alert(crr); // 4 3 2 1  
 **return** (pre + crr);  
});  
alert(sum); //15  
//第一次 pre=5，crr=4

4.3 Date类型

/\*Date\*/  
**var** now = **new** Date();//调用构造函数而不传递参数，新创建的对象自动获得当前日期和时间  
document.write(now + "<br/>");//Sun Sep 20 2015 14:18:35 GMT+0800 (中国标准时间)  
  
**var** someDate = **new** Date(Date.parse("May 25,2004"));  
//如果使用Date构造函数传入参数，应该传入的是距离1970.1.1 00.00.00至今的毫秒数  
//而使用Date.parse传入的可以不是毫秒数  
document.write(someDate + "<br/>");//Tue May 25 2004 00:00:00 GMT+0800 (中国标准时间)  
  
someDate = **new** Date("May 25,2004");//在后台调用了Date.parse函数，与前面的效果是一样的  
document.write(someDate + "<br/>");//Tue May 25 2004 00:00:00 GMT+0800 (中国标准时间)  
  
//Date.UTC()作用于Date.parse()类似， 参数为年份、基于0的月份（一月是0，二月是1）、月中的天数（1-31）、小时数（0-23）、分钟、秒以及毫秒数  
//其中参数年和月是必须的，其他可选，不传入其他参数默认情况是天数为1，其他为0  
someDate = **new** Date(Date.UTC(2000,1));  
document.write(someDate + "<br/>");//注意这个是基于0时区的基础上+8的！ Tue Feb 01 2000 08:00:00 GMT+0800 (中国标准时间)  
someDate = **new** Date(Date.UTC(2000,1,5,5,55,55));  
document.write(someDate + "<br/>");//小时需要+8！ 5+8=13 Sat Feb 05 2000 13:55:55 GMT+0800 (中国标准时间)  
  
//如果模仿Date.parse()函数一样，Date构造函数也会模仿Date.UTC(),但是不同的是日期和时间都基于本地时间而非GMT来创建的  
//所以这里的时间不用+8处理的  
//基于本地时区创建的时间  
someDate = **new** Date(2000,1,5,5,55,55);//在后台调用了Date.parse函数，与前面的效果是一样的  
document.write(someDate + "<br/>");//Sat Feb 05 2000 05:55:55 GMT+0800 (中国标准时间),这样使用不需要+8小时  
  
  
  
//取得开始时间,距离1970.1.1 00.00.00的毫秒数  
**var** start = Date.now();  
alert(start);  
//取得结束时间  
**var** stop = Date.now();  
alert(stop);  
**var** result = stop -start;  
alert(result);  
  
//如果浏览器不支持以上函数  
//使用+操作符把Date对象的当前时间转换成字符串  
start = +**new** Date();  
document.write(start + "<br/>");//1442732531517  
stop = +**new** Date();  
document.write(stop + "<br/>");//1442732531517

/\*Date继承的方法\*/  
/\*与其他引用类型一样，Date类型也重写了toLocalString()、toString()和valueof()方法\*/  
  
**var** now = **new** Date();  
  
document.write(now.toLocaleString() + "<br/>");  
//chorme  
//2015/9/20 下午3:20:59  
//360  
//2015年9月20日 下午3:21:50  
document.write(now.toString() + "<br/>");  
//chorme、360  
//Sun Sep 20 2015 15:20:59 GMT+0800 (中国标准时间)  
  
//chorme 因浏览器而异  
document.write(now.toDateString() + "<br/>");//Sun Sep 20 2015  
document.write(now.toTimeString() + "<br/>");//15:24:24 GMT+0800 (中国标准时间)  
document.write(now.toLocaleDateString() + "<br/>");//2015/9/20  
document.write(now.toLocaleTimeString() + "<br/>");//下午3:25:24  
document.write(now.toUTCString() + "<br/>");//Sun, 20 Sep 2015 07:26:06 GMT  
  
document.write(now.getTime() + "<br/>");//1442734060484  
document.write(now.valueOf() + "<br/>");//1442734060484  
now.setTime(1442734060484);  
document.write(now + "<br/>");//Sun Sep 20 2015 15:27:40 GMT+0800 (中国标准时间)  
document.write(now.getFullYear() + "<br/>");//2015  
document.write(now.getUTCFullYear() + "<br/>");//2015  
document.write(now.getMonth() + "<br/>");//8  
document.write(now.getUTCMonth() + "<br/>");//8  
now.setMonth(5);  
document.write(now + "<br/>");//Sat Jun 20 2015 15:27:40 GMT+0800 (中国标准时间)  
document.write(now.getDate() + "<br/>");//20

4.4 RegExp类型

/\*RegExp\*/  
  
//var expression = /pattern/flags  
//g : 表示全局模式，被应用于所有字符串，而非发现第一个匹配时立即停止  
//i ：表示不区分大小写  
//m ：表示多行模式  
  
**var** pattern = /at/g; //匹配字符串中的所有at  
**var** pattern1 = /[bc]at/i; //匹配不区分大小写的bat或cat  
**var** pattern2 = /.at/gi;//匹配所有以at结尾的字符的组合？不区分大小写  
document.write(pattern.test("i at my at and cat") + "<br/>"); //true  
document.write(pattern.test("ejwkefj adfjask") + "<br/>"); //false  
document.write(pattern1.test("bat cat and i") + "<br/>"); //true  
document.write(pattern2.test("baat caat and i") + "<br/>"); //true  
document.write(pattern2.test("btaa ctaa and i") + "<br/>"); //false  
  
**var** pattern3 = **null**;  
**var** i = 0;  
  
**for**(i=0; i<10; i++) {  
 pattern3 = /cat/g;  
 document.write(pattern3.test("cat and i") + "<br/>"); //true ECMAScript5标准规定  
}  
  
**for**(i=0; i<10; i++) {  
 pattern3 = **new** RegExp("cat","g");  
 document.write(pattern3.test("cat and i") + "<br/>"); //true  
}  
  
/\*RegExp的属性\*/  
*/\*\*  
 \* global:布尔值，表示是否设置了g标志  
 \* ignoreCase:布尔值，表示是否设置了i标志  
 \* lastIndex：整数，表示开始搜索下一个匹配项的字符位置，从0算起  
 \* multiline：布尔值，表示是否设置了m标志  
 \* source：正则表达式的字符串表示  
 \*/***var** pattern1 = /\[bc\]at/i;  
document.write("<br/>");  
document.write(pattern1.global + "<br/>"); //false  
document.write(pattern1.ignoreCase + "<br/>"); //true  
document.write(pattern1.multiline + "<br/>"); //false  
document.write(pattern1.lastIndex + "<br/>"); //0  
document.write(pattern1.source + "<br/>"); //\[bc\]at  
  
**var** pattern2 = **new** RegExp("\\[bc\\]at", "i");  
//与pattern1的正则表达式字面量传递不一样，构造函数里只能传递字符串  
//字面量中的\在字符串里需要转义成\\  
document.write(pattern2.global + "<br/>"); //false  
document.write(pattern2.ignoreCase + "<br/>"); //true  
document.write(pattern2.multiline + "<br/>"); //false  
document.write(pattern2.lastIndex + "<br/>"); //0  
document.write(pattern2.source + "<br/>"); //\[bc\]at, souce属性为字面量形式的字符串

/\*RegExp实例方法\*/  
  
//exec()方法，返回一个数组，没有匹配项就返回null  
//数组包含两个额外的属性，index和input  
//index表示匹配项在字符串中的位置  
//input表示应用正则表达式的字符串  
  
//数组中第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串（如果没有捕获组，数组只包含一项）  
  
**var** text = "mom and dad and baby";  
**var** pattern = /mom( and dad( and baby)?)?/gi;  
  
**var** matches = pattern.exec(text);  
document.write(matches.index + "<br/>"); //0  
document.write(matches.input + "<br/>"); //mom and dad and their son  
document.write(matches[0] + "<br/>"); //mom and dad  
document.write(matches[1] + "<br/>"); //and dad  
document.write(matches[2] + "<br/>"); //and baby  
  
**var** text = "cat,bat,sat,fat";  
**var** pattern1 = /.at/; //以at结尾,不是全局模式  
  
**var** matches = pattern1.exec(text);  
document.write(matches.index + "<br/>"); //0  
document.write(matches[0] + "<br/>"); //cat  
document.write(pattern1.lastIndex + "<br/>"); //0  
  
**var** matches = pattern1.exec(text);  
document.write(matches.index + "<br/>"); //0  
document.write(matches[0] + "<br/>"); //cat  
document.write(pattern1.lastIndex + "<br/>"); //0  
  
  
**var** pattern1 = /.at/g; //以at结尾,是全局模式  
  
**var** matches = pattern1.exec(text);  
document.write(matches.index + "<br/>"); //0  
document.write(matches[0] + "<br/>"); //cat  
document.write(pattern1.lastIndex + "<br/>"); //3  
  
**var** matches = pattern1.exec(text);  
document.write(matches.index + "<br/>"); //4  
document.write(matches[0] + "<br/>"); //bat  
document.write(pattern1.lastIndex + "<br/>"); //7  
  
//如果不是全局模式，exec()返回的都是第一个匹配项，而如果是全局模式，每次调用exec()都会返回字符串的下一个匹配项  
//直至匹配结束

//test()方法，返回一个布尔值，匹配到参数返回true  
**var** text = "000-00-0000";  
**var** pattern = /\d{3}-\d{2}-\d{4}/;  
  
**if**(pattern.test(text)){ //true  
 document.write("The pattern was matched." + "<br/>");  
}  
  
**var** pattern = **new** RegExp("\\[bc\\]at","gi");  
document.write(pattern.source + "<br/>"); // \[bc\]at,正则表达式字面量，其实需要匹配的是[bc]at;  
  
document.write(pattern.toString() + "<br/>"); // /\[bc\]at/gi  
document.write(pattern.toLocaleString() + "<br/>"); // /\[bc\]at/gi

/\*RegExp构造函数属性\*/  
**var** text = "this has been a short summer";  
**var** pattern = /(.)hort/g;//匹配任何字符后面跟hort，而且把第一个字符放在了一个捕获组中  
  
**if**(pattern.test(text)){  
 document.write(RegExp.input + "<br/>"); //this has been a short summer,最近一次要匹配的字符串  
 document.write(RegExp.rightContext + "<br/>");//summer lastMatch之后的文本  
 document.write(RegExp.leftContext + "<br/>");//this has been a lastMatch之前的文本  
 document.write(RegExp.lastMatch + "<br/>");//short 最近一次的匹配项  
 document.write(RegExp.lastParen + "<br/>");//s 最近一次匹配的捕获组  
 document.write(RegExp.multiline + "<br/>");//false 表示是否所有表达式都使用多行模式  
}  
  
  
pattern = /(..)or(.)/g;  
  
**if**(pattern.test(text)) {  
 document.write(RegExp.$1 + "<br/>");//sh $1捕获组1  
 document.write(RegExp.$2 + "<br/>");//t $2捕获组2 一直到捕获组9  
}  
/\*模式的局限性\*/  
*/\*\*  
 \* 1.插入^和$来匹配字符的开始和结尾  
 \*/*

4.5 Function类型

/\*Function类型\*/  
  
//Function实际上是对象，函数都是Function类型的实例，与其他引用类型一样，都有属性和方法  
  
**function** sum(sum1,sum2){  
 **return** sum1 + sum2;  
}//无分号  
  
**var** oSum = **function**(sum1,sum2){  
 **return** sum1 + sum2;  
}; //有分号  
  
//使用Function构造函数，不推介,两次解析影响性能，只是说明函数是对象

//函数是对象，函数名是指向函数的指针  
**var** Sum = **new** Function("num1","num2","return num1 + num2");  
  
**function** sum(sum1,sum2){  
 **return** sum1 + sum2;  
}  
document.write(sum(10,15) + "<br/>"); //25  
  
**var** oSum = sum; //函数名是访问函数的指针，这里是指针赋值，使它们指向同一个函数对象  
document.write(oSum(1,15) + "<br/>"); //16  
  
sum = **null**;//把sum设置为空指针  
document.write(oSum(1,15) + "<br/>"); //16

/\*Function类型没有重载\*/  
**function** addSum(num){  
 **return** num + 100;  
}  
  
**function** addSum(num){  
 **return** num + 200;  
}  
  
**var** result = addSum(100);  
alert(result); //300  
  
//以上代码等同于  
**var** addNum = **function**(num){  
 **return** 100 + num;  
}  
  
addNum = **function**(num){  
 **return** 200 + num;  
}  
  
alert(addNum(100)); //300,覆盖了引用第一个函数的变量

/\*Function类型函数声明与函数表达式\*/  
//解析器对于函数声明和函数表达式并非一视同仁  
//解析器会率先读取函数声明，使其在执行任何代码前可用  
//函数表达式则必须等到解析器执行到它所在的代码行，才会真正在解析执行  
  
  
alert(sum(10,10)); //20  
//函数声明  
**function** sum(num1,num2){  
 **return** num1 + num2;  
}  
  
  
alert(Sum(10,10)); //Uncaught TypeError: Sum is not a function  
**var** Sum = **function**(num1,num2){ //函数位于一个初始化语句中，在这之前并没有保存Sum对函数的引用  
 **return** num1 + num2;  
}

/\*作为值得函数\*/  
**function** callSomeFunction(someFunction, arg){ //两个参数，第一个参数是一个函数，第二个参数是传递给该函数的值  
 **return** someFunction(arg);  
}  
  
  
**function** addSum(num){  
 **return** num + 10;  
}  
  
**var** result = callSomeFunction(addSum,10); //20,函数名本身就是变量，函数可以作为值来使用  
alert(result);  
  
**function** getGreetName(name){  
 **return** "Hello," + name;  
}  
  
result = callSomeFunction(getGreetName,"Victor"); //访问的是函数的指针，+上()是执行函数  
alert(result); //Hello,Victor

/\*作为值得函数\*/  
**function** compare(propertyName){  
 **return function**(obj1,obj2){   
 **var** value1 = obj1[propertyName];  
 **var** value2 = obj2[propertyName];  
  
 **if**(value1 < value2){  
 **return** -1;  
 } **else if**(value1 > value2){  
 **return** 1;  
 } **else**{  
 **return** 0;  
 }  
 };  
}  
  
  
**var** date = [{name:"xiankang", age:23},{name:"zhuxiankang",age:22}];  
  
date.sort(compare(name));  
alert(date[0].name); //xiankang,升序排列  
  
date.sort(compare(age));  
alert(date[0].age); //22,升序排列  
  
//属性名可以是字符串，如果属性名是数字，那么会自动转换成字符串。  
//使用方括号语法时，应该将要访问的属性以<字符串>的形式放在方括号中

/\*作为值得函数\*/  
**function** compare(propertyName){  
 **return function**(obj1,obj2){ //返回的是函数  
 **var** value1 = obj1[propertyName]; //使用方括号表示法来取得了给定属性的值  
 **var** value2 = obj2[propertyName];  
  
 **if**(value1 < value2){  
 **return** -1;  
 } **else if**(value1 > value2){  
 **return** 1;  
 } **else**{  
 **return** 0;  
 }  
 };  
}  
  
  
**var** date = [{name:"xiankang", age:23},{name:"zhuxiankang",age:22}];  
  
date.sort(compare("name"));  
alert(date[0].name); //xiankang,升序排列  
  
date.sort(compare("age"));  
alert(date[0].age); //22,升序排列  
  
//属性名可以是字符串，如果属性名是数字，那么会自动转换成字符串。  
//使用方括号语法时，应该将要访问的属性以<字符串>的形式放在方括号中

/\*函数内部的属性\*/  
  
//arguments属性  
**function** factorial(num){  
 **if**(num <= 1){  
 **return** 1;  
 } **else**{  
 **return** num\*factorial(num-1);  
 }  
}  
  
**function** Factorial(num){  
 **if**(num <= 1){  
 **return** 1;  
 } **else** {  
 //document.write(arguments.callee); //指向了arguments所在函数对象  
 */\*\*  
 \* function Factorial(num)  
 \* { if(num <= 1){  
 \* return 1; } else {  
 \* document.write(arguments.callee);  
 \* return num \* arguments.callee(num - 1); } }  
 \*/* **return** num \* arguments.callee(num - 1);  
  
 }  
}  
  
//arguments类数组对象是函数的内部属性，主要用途是保存函数的参数。  
//arguments对象有一个callee属性，该属性是一个指针，指向了拥有这个arguments对象的函数  
alert(Factorial(3)); //6  
  
  
**function** factorial1(num){  
 **if**(num <= 1){  
 **return** 1;  
 } **else** {  
 **return** num \* arguments.callee(arguments[0]- 1); //arguments = [1];数组只有一个元素  
 //arguments[0] = num;  
 }  
}  
  
alert(factorial1(4)); //24  
  
**var** trueFactorial = Factorial;  
  
Factorial = **function**(){  
 **return** 0;  
}  
  
alert(trueFactorial(4)); //24  
  
  
**var** truefactorial = factorial;  
factorial = **function** () {  
 **return** 0;  
}  
  
alert(truefactorial(4)); //0 为什么是0呢？  
//因为factorial函数被重新定义了，返回了0值，这是使用了arguments.callee和不适用这个属性的原因

//this属性  
//全局对象中this对象引用的就是window对象，this引用的是函数据以执行的环境对象,全局函数中this指代window  
  
window.color = "red";  
**function** sayColor(){  
 alert(**this**.color);  
}  
  
sayColor(); //red  
  
**var** o = {color:"blue"};  
o.sayColor = sayColor; //使o的sayColor方法指向了函数sayColor，o.sayColor和sayColor指向的是同一个函数  
o.sayColor(); //blue 调用该方法，this引用的是o对象

//this属性  
//全局对象中this对象引用的就是window对象，this引用的是函数据以执行的环境对象,全局函数中this指代window  
  
window.color = "red";  
**function** sayColor(){  
 alert(**this**.color);  
}  
  
sayColor(); //red  
  
**var** o = {color:"blue"};  
o.sayColor = sayColor; //使o的sayColor方法指向了函数sayColor，o.sayColor和sayColor指向的是同一个函数  
o.sayColor(); //blue 调用该方法，this引用的是o对象  
  
//caller属性  
//调用当前函数的函数的引用  
  
**function** outer(){  
 inner();  
 Inner();  
}  
  
**function** inner(){  
 document.write(inner.caller); //指向了outer，因为是outer调用了inner  
 //function outer(){ inner(); }  
}  
  
outer();  
  
  
**function** Inner(){  
 document.write(arguments.callee.caller);  
 //arguments。callee指向了拥有arguments对象的函数，caller使指向的函数又指向了调用它的函数  
 //function outer(){ inner(); Inner(); }  
}  
  
//严格模式下访问arguments.callee会导致错误，不能为函数的caller属性赋值

//函数的属性和方法  
//每个函数都包含两个非集成而来的方法apply()和call()  
//在特定的作用域中调用函数，实际上等于设置函数体内this对象的值  
//apply接收两个参数。一个是在其中运行的函数的作用域，另一个是参数数组  
//第二个参数也可以是Array实例，也可以是arguments对象  
**function** sum(num1,num2){  
 **return** num1 + num2;  
}  
  
**function** callSum1(num1,num2){ //执行sum函数  
 **return** sum.apply(**this**,arguments);  
 //全局作用域中调用的this指代window，传入arguments对象  
}  
  
**function** callSum2(num1,num2){  
 **return** sum.apply(**this**,[num1,num2]);//传入数组  
}  
  
alert(callSum1(10,10)); //20  
alert(callSum2(10,20)); //30  
  
  
//使用call函数时第一参数不变，第二参数是一一列举的传入的每一个参数，而不能是数组的形式  
  
**function** callSum(num1,num2){  
 **return** sum.call(**this**,num1,num2);  
}  
  
alert(callSum(10,30)); //40  
  
//使用call和apply完全取决于哪种传递参数的方式最方便  
//最强大的地方是可以扩充函数赖以运行的作用域  
  
window.color = "red";  
**var** o = {color:"blue"};  
  
**function** sayColor(){  
 alert(**this**.color);  
}  
  
sayColor(); //red this指代window  
  
sayColor.call(**this**); //red  
sayColor.call(window); //red  
sayColor.call(o); //blue  
//函数的执行环境就不一样了，因为此时函数体内的this对象指向了o  
//使用call扩充作用域的最大好处是对象不需要与方法有任何的耦合关系  
  
  
//bind()函数  
**var** oSayColor = sayColor.bind(o);//sayColor调用了bind并传入了对象o  
//oSayColor的this值指代对象o  
oSayColor(); //blue

4.6 基本包装类型

/\*基本包装类型\*/ //String Number Boolean  
**var** s1 = "string text";  
**var** s2 = s1.substring(2);  
alert(s2);//ring text  
  
//s1包含一个字符串，字符串当然是基本类型  
//调用了s1的方法，从逻辑上来说基本类型是没有方法的  
//后台完成了一系列操作1.创建String类型的一个实例 2.在实例上调用指定的方法 3.销毁这个实例  
  
**var** s = **new** String("string text"); //对于Number和Boolean也适用  
**var** s3 = s.substring(2);  
s = **null**;  
  
//两者的主要区别是对象的生存期  
//引用类型 Object Array Date RegExp Function | 使用new关键字申请的对象 String Number Boolean在执行流离开当前作用域之前都一直保存在内存中  
//基本包装类型 String Number Boolean 只存在于一行代码的执行瞬间，然后立即被销毁  
//不能再运行时为基本类型值添加属性和方法  
**var** string = "some text";  
string.color = "red"; //在执行完这一条时对象立即被销毁了  
alert(string.color); //undefined 又重新创建了最近的String对象，但是没有color属性  
  
  
**var** obj = **new** Object("some text");  
alert(obj **instanceof** String); //true  
  
obj = **new** Object(4);  
alert(obj **instanceof** Number); //true  
  
  
//转型函数和构造函数的区别  
**var** value = "23";  
**var** number = Number(value); //将字符串转换为数字的函数 转型函数,保存的是基本类型值23  
alert(**typeof** number); //number  
  
**var** obj = **new** Number(23); //保存的是Number对象的实例  
alert(**typeof** obj); //object

/\*基本包装类型-Boolean类型\*/  
  
//基本类型和引用类型的布尔值的区别1  
  
**var** result = **new** Boolean(**false**);  
alert(result); //false  
  
**var** Result = result && **true**;  
//result这里是一个Boolean对象，布尔表达式的所有对象都会被转换为true  
alert(Result); //true!  
  
**var** result1 = **false**; //基本数据类型  
Result = result1 && **true**;  
alert(Result); //false  
  
//对基本类型返回boolean，对引用类型返回Object  
document.write(**typeof** result + "<br/>"); //object  
document.write(**typeof** result1 + "<br/>");//boolean  
  
  
**var** re = **new** Boolean(**false**);  
**var** re1 = **true**;  
document.write((re **instanceof** Boolean) + "<br/>"); //true  
document.write((re1 **instanceof** Boolean) + "<br/>");//false

/\*基本包装类型-Number类型\*/  
  
**var** num = **new** Number(3);  
alert(num.valueOf()); //valueof返回的是对象表示的基本类型的数值  
  
//toFixed()方法会按照指定的小数位返回数值的字符串表示  
  
**var** num = 10;  
alert(num.toFixed(2)); //"10.00"  
  
num = 10.902;  
alert(num.toFixed(2)); //"10.90"  
  
num = 90;  
alert(num.toPrecision(1));//9e+1  
alert(num.toPrecision(2));//90  
alert(num.toPrecision(3));//90.0