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; //23转换为数字了  
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。和C指针类似。

\*/  
  
**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){  
  **return -**1;  
 } **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]; //[]表示法的属性名必须是字符串  
 **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

/\*基本包装类型-String类型\*/  
  
//charAt() charCodeAt()，按索引值查找字符串元素  
**var** string = "hello world";  
document.write(string.charAt(1) + "<br/>");//e  
document.write(string.charCodeAt(1) + "<br/>"); //101 字母e的字符编码  
  
//indexOf() lastIndexOf()，按元素查找索引值,没有找到子字符串则返回-1  
document.write(string.indexOf("o") + "<br/>"); //4  
document.write(string.lastIndexOf("o") + "<br/>"); //7 从后向前索引  
  
document.write(string.indexOf("o",6)+ "<br/>"); //7 从字符串的位置6开始向后索引  
document.write(string.lastIndexOf("o",6) + "<br/>"); //4 从字符串的位置6开始向前索引  
  
**var** stringValue = "hello love like and lucky";  
**var** position = **new** Array();  
  
**var** pos = stringValue.indexOf("l");  
  
**while**(pos > -1){ //没有找到子字符串则返回-1  
 position.push(pos);  
 pos = stringValue.indexOf("l",pos+1);  
}  
alert(position); //2,3,6,11,20  
  
//concat(),返回拼接得到的新字符串  
document.write(string.concat(" zhuxiankang") + "<br/>"); //hello world zhuxiankang  
**var** stringNew = string.concat(" hi"," world");  
document.write(stringNew + "<br/>"); //hello world hi world  
//使用 + 操作符更方便  
  
//slice() substr() substring() 返回一个子串  
//第一个参数指定字符串的起始位置，第二个参数表示字符串到哪里结束  
//没有给定第二个参数，则将字符串的长度作为结束位置  
document.write(string.slice(3)+ "<br/>"); //lo world  
document.write(string.substring(3)+ "<br/>"); //lo world  
document.write(string.substr(3)+ "<br/>"); //lo world  
  
document.write(string.slice(3,7)+ "<br/>"); //lo w  
document.write(string.substring(3,7)+ "<br/>"); //lo w， 第二参数指定的是位置，结果中不包含第7个位置  
document.write(string.substr(3,7)+ "<br/>"); //lo world, 第二参数指定的是返回的个数是7个  
  
//trim() 创建新的字符串副本，删除前置和后置空格  
**var** stringValue = " hello world ";  
**var** trimString = stringValue.trim(); //hello world 去掉了前后的空格  
  
//toLowerCase() toUpperCase() 转换字符串的大小写  
alert(trimString.toLowerCase()); //hello world  
alert(trimString.toUpperCase()); //HELLO WORLD  
alert(trimString.toLocaleLowerCase()); //hello world  
alert(trimString.toLocaleUpperCase()); //HELLO WORLD

//match() 字符串的模式匹配 本质上与RegExp的exec()方法相同  
**var** text = "tat,cat,bat,sat,fat";  
**var** pattern = /.at/;  
  
**var** array = text.match(pattern);  
document.write(array.input + "<br/>"); //tat,cat,bat,sat,fat  
document.write(array.index + "<br/>"); //0  
document.write(array.lastIndex + "<br/>"); //undefined  
document.write(array[0] + "<br/>"); //tat  
  
//exec() -> Reg.exec(Str) exec是RegExp对象的方法  
//match() ->Str.match(Reg) match是String对象的方法  
//两个函数的返回都一样， 包括index属性、input属性和匹配项的数组，没有匹配项则返回null  
//index表示匹配项在字符串中的位置  
//input表示应用表达式的字符串  
//lastIndex表示开始搜索的下一个匹配项的字符位置  
  
  
//search() 始终是字符串开头向后索引，返回字符串中第一个匹配项的索引，没有找到返回-1  
text = "hello cat";  
pattern = /at/;  
**var** pos = text.search(pattern);  
alert(pos); //7 cat  
  
//replace() 替换字符串 两个参数1.RegExp对象或者字符串 2.可以是一个字符串或者一个函数  
text = "cat,bat,sat,fat";  
pattern = /at/g;  
**var** result = text.replace("at","ond"); //如果只是字符串，则是替换第一个子字符串  
alert(result); //cond,bat,sat,fat  
  
result = text.replace(pattern,"ond"); //正则必须指定全局g标志  
alert(result); //cond,bond,sond,fond  
　　　　　　 　  
//如果第二参数是函数  
//函数本身接收三个参数：1.模式的匹配项 2.模式匹配项在字符串中的位置 3.原始字符串  
**function** htmlReplace(text,pattern){  
 **return** text.replace(pattern,**function**(match,pos,originalText){  
 **switch**(match){  
 **case** "<":  
 **return** "&lt;";  
 **case** ">":  
 **return** "&gt;";  
 **case** "&":  
 **return** "&amp;";  
 **case** "\"":  
 **return** "&quot;";  
 }  
 });  
}  
  
pattern = /[<>"&]/g;  
document.write(htmlReplace("<p class=\"greeting\">hello world!</p>",pattern) + "<br/>"); //<p class="greeting">hello world!</p>  
alert(htmlReplace("<p class=\"greeting\">hello world!</p>",pattern));  
//&lt:p class=&quot;greeting&quot;&gt;hello world!&lt;/p&gt  
  
  
//split() 将一个字符串分割成多个子字符串，并将结果放在一个数组中。  
//基于指定的分隔符，这个分隔符可以是字符串，也可以是RegExp对象（这个方法不会将字符串看成是一个正则表达式，不推荐使用）  
//第二个参数用于指定分割的数组的大小  
**var** colorText = "red,blue,green,yellow";  
**var** colors1 = colorText.split(",");  
document.write(colors1[0] + "<br/>"); //red  
**var** colors2 = colorText.split(",",2); //red blue  
document.write(colors2 + "<br/>"); //["red","blue"] 数组

//localeCompare()  
  
**function** determineOrder(preValue,Value){  
 **var** result = preValue.localeCompare(Value);  
  
 **if**(result < 0){  
 alert("The string " + preValue + " comes before the string " + Value + ".");  
 } **else if**(result > 0){  
 alert("The string " + preValue + " comes after the string " + Value + ".");  
 } **else**{  
 alert("The string " + preValue + " is equal to the string " + Value + ".");  
 }  
}  
  
determineOrder("yellow","blue"); //after  
determineOrder("yellow","zoo"); //before  
  
//fromCharCode() 与charCodeAt()执行相反的操作  
alert(String.fromCharCode(104,101,108,108,111)); //hello

4.7 单体内置对象

/\*单体内置对象\*/  
//前面提了大多数内置对象 例如：Object、Array、String  
//ECMA-262还定义了两个单体内置对象：Global和Math,这两个都是全局对象  
//例如isNaN() isFinite() parseInt() parseFloat() 实际上都是Global对象的方法  
  
//encodeURI() 主要用于整个URI，不会对本身属于URI的特殊字符进行编码，例如冒号。正斜杠 问号 和#号  
//encodeURIComponent() 主要用于对URI中的某一段进行编码，对任何非标准字符进行编码  
  
**var** uri = "http://www.wrox.com/illegal vlaue.htm#start";  
document.write(encodeURI(uri) + "<br/>"); //http://www.wrox.com/illegal%20vlaue.htm#start  
//只有空格被替换成20%  
document.write(encodeURIComponent(uri) + "<br/>"); //http%3A%2F%2Fwww.wrox.com%2Fillegal%20vlaue.htm%23start  
//替换了所有非字母数字字符，使用encodeURIComponent比encodeURI多  
  
**var** URI = encodeURI(uri);  
document.write(URI + "<br/>"); //http://www.wrox.com/illegal%20vlaue.htm#start  
document.write(decodeURI(uri) + "<br/>"); //http://www.wrox.com/illegal vlaue.htm#start  
  
**var** URI = encodeURIComponent(uri);  
document.write(URI + "<br/>"); //http%3A%2F%2Fwww.wrox.com%2Fillegal%20vlaue.htm%23start  
document.write(decodeURIComponent(URI) + "<br/>"); //http://www.wrox.com/illegal vlaue.htm#start  
  
//eval() eval方法就像是一个完整的ECMAScript解析器，只接受一个参数，既要执行的ECMAScript字符串  
eval("alert('hi')"); //alert("hi");  
//把字符串当做实际的ECMAScript语句来执行  
//执行代码具有与该执行环境相同的作用域链  
  
**var** msg = "hello world";  
eval("alert(msg)"); //hello world  
  
eval("function sayHi() {alert('hi');}"); //eval调用被替换成了实际代码  
sayHi(); //hi  
  
eval("var zmsg = 'zhu xiankang';"); //使用eval要谨慎，防止代码注入  
alert(zmsg); //zhu xiankang  
  
//严格模式下，外部访问不到eval创建爱你的的任何变量或函数  
  
  
//Global对象的属性  
//例如：undefined NaN Infinity Object等等  
  
//window对象 ECMAScript虽然没有指出如何直接访问Global对象，但WEB浏览器都是将这个对象作为window对象的一部分加以实现的。  
//window对象除了扮演ECMAScript规定的Global对象的角色外，还承担了很多别的任务

/\*单体内置对象 - Math对象\*/  
//max() min()  
  
**var** Max = Math.max(3,54,53,6);  
alert(Max); //54  
  
**var** Min = Math.min(3,54,53,6);  
alert(Min); //3  
  
//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  
**var** values = [10,30,4565,78];  
  
**var** max = Math.max.apply(**this**,values); //this在这里指代Math  
alert(max); //4565  
  
min = Math.min.apply(Math,values); //也可以直接用Math  
alert(min); //10  
  
  
//ceil() floor() round()  
//ceil 向上舍入 floor 向下舍入 round 四舍五入  
  
document.write(Math.ceil(29.6) + "<br/>"); //30  
document.write(Math.ceil(29.1) + "<br/>"); //30  
document.write(Math.floor(29.6) + "<br/>");//29  
document.write(Math.floor(29.1) + "<br/>");//29  
document.write(Math.round(29.6) + "<br/>");//30  
document.write(Math.round(29.1) + "<br/>");//29  
  
//random()  
document.write(Math.random() + "<br/>"); //产生0-1的随机数0.9378104046918452  
document.write((Math.floor(Math.random() \* 10 + 1)) + "<br/>"); //产生1-10的随机数  
document.write((Math.floor(Math.random() \* 10 + 1)) + "<br/>"); //产生1-10的随机数  
document.write((Math.floor(Math.random() \* 10 + 1)) + "<br/>"); //产生1-10的随机数  
document.write((Math.floor(Math.random() \* 9 + 2)) + "<br/>"); //产生2-10的随机数  
  
**function** randomDigital(lowerDigital,upperDigital){ //选择[lowerDigital,upperDigital]中的随机数  
 **var** value = upperDigital - lowerDigital + 1;  
 **return** Math.floor(Math.random() \* value + lowerDigital);  
}  
  
document.write(randomDigital(100,1000) + "<br/>"); //产生100-1000的随机数  
document.write(randomDigital(100,1000) + "<br/>");  
document.write(randomDigital(100,1000) + "<br/>");  
  
**var** colors = ["red","blue","yellow","black","green"];  
  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");  
document.write(colors[randomDigital(0,4)] + "<br/>");

第五章 面向对象的程序设计

5.1 理解对象

/\*对象\*/  
  
//对象字面量创建对象  
**var** person = {  
 name: "zhuxiankang",  
 age: 25,  
 sayName: **function**(){  
 alert(**this**.name);  
 }  
};  
  
person.sayName(); //zhuxiankang  
  
/\*属性类型\*/  
  
//数据属性  
**var** Person = {};  
//defineProperty()传递三个参数：1.属性所在的对象，属性的名字，一个描述符对象、  
//描述符对象的属性必须是 configurable、enumerable、writable和value，可以设置一个或多个属性值  
//configurable：能否通过delete删除属性从而重新定义属性，能否修改属性的特性，能否把属性修改成访问器属性  
//enumerable：能否通过for-in循环返回属性  
//writable：能否修改属性的值  
//value：读取属性值得时候从这个位置读，写入属性值得时候把新值保存在这个位置  
//IE8不建议使用此方法  
//{}内的是一个描述符对象  
Object.defineProperty(Person,"name", {  
 configurable:**false**,  
 writable: **false**,//不可写  
 value: "zhuxiankang"  
});  
  
  
alert(Person.name); //zhuxiankang  
Person.name = "tanghaixia";  
alert(Person.name); //zhuxiankang 只读的，不能写，所以值不会变  
  
**delete** Person.name;  
alert(Person.name);//zhuxiankang 不能从对象中删除属性  
  
//一旦把属性定义为不可配置，就不能在把它变为可配置的，此时如果修改除writable之外的特性都会导致错误  
/\*Object.defineProperty(Person,"name", { //Uncaught TypeError: Cannot redefine property: name  
 configurable:true,  
 writable: false,//不可写  
 value: "zhuxiankang"  
});\*/  
  
//可以多次修改同一个属性，但是把configurable设置为false以后就会有限制了  
  
//访问器属性,常见方式是设置访问器属性的值会使其他属性发生变化  
  
//configurable：能否通过delete删除属性从而重新定义属性，能否修改属性的特性，能否把属性修改成访问器属性  
//enumerable：能否通过for-in循环返回属性  
//get:在读取属性时调用的函数，默认为undefined  
//set:在写入属性时调用的函数，默认为undefined  
  
**var** book = {  
 \_year: 2004,  
 version: 1  
};  
  
Object.defineProperty(book,"year",{  
 **get**:**function**(){  
 **return this**.\_year;  
 },  
 **set**:**function**(newValue){  
 **if**(newValue > 2004){  
 **this**.\_year = newValue;  
 **this**.version = newValue - 2004;  
 }  
 }  
});  
  
//读取访问其属性时会调用get函数，而这里是写入访问器属性的值，调用了setter函数并写入了新值  
book.year = 2005;//访问器属性year的值修改了以后导致了其他属性也修改了  
alert(book.\_year); //2005  
alert(book.version); //1  
  
//只制定getter意味着属性是不能写，尝试写入属性会被忽略  
//只指定setter函数的属性也不能读  
  
  
//定义访问器属性的旧有方法  
  
**var** BOOK = {  
 \_year: 2004,  
 version:1  
};  
  
BOOK.\_\_defineGetter\_\_("year",**function**(){  
 **return this**.\_year;  
});  
  
BOOK.\_\_defineSetter\_\_("year",**function**(newValue){  
 **if**(newValue > 2004){  
 **this**.\_year = newValue,  
 **this**.version = newValue - 2004  
 }  
});  
  
BOOK.year = 2009;  
alert(BOOK.version); //5  
  
  
//定义多个属性  
  
**var** my= {}  
Object.defineProperties(my, {  
 \_year:{  
 value:2018  
 },  
  
 edition:{  
 value:1  
 },  
  
 year:{  
 **get**:**function**(){  
 **return this**.\_year;  
 },  
  
 **set**:**function**(newValue){ //newValue = 2012  
 **if**(newValue > 2004){  
 **this**.\_year = newValue; //  
 **this**.edition = newValue - 2004;  
 }  
 }  
 }  
});  
  
  
my.year = 2012;  
alert(my.\_year); //2018?为什么值没有变  
alert(my.edition);//1

/\*读取属性的特性\*/  
  
**var** book= {};  
  
Object.defineProperties(book, {  
 \_year:{  
 value:2018  
 },  
  
 edition:{  
 value:1  
 },  
  
 year:{  
 **get**:**function**(){  
 **return this**.\_year;  
 },  
  
 **set**:**function**(newValue){ //newValue = 2012  
 **if**(newValue > 2004){  
 **this**.\_year = newValue; //  
 **this**.edition = newValue - 2004;  
 }  
 }  
 }  
});  
  
//getOwnPropertyDescriptor接收两个参数，1.属性所在的对象 2.读取其描述符的属性名称  
//返回是一个对象 如果是数据属性，这个返回对象的属性有configurable、enumerable、writable、value  
//如果是访问器属性，返回对象的属性有configurable、enumerable、get和set  
**var** des = Object.getOwnPropertyDescriptor(book,"\_year");  
document.write(des.value + "<br/>"); //2018  
document.write(des.writable + "<br/>"); //false  
document.write(des.configurable + "<br/>"); //false  
document.write(des.get + "<br/>"); //undefined  
  
**var** des = Object.getOwnPropertyDescriptor(book,"year");  
document.write(des.value + "<br/>"); //undefined  
document.write(des.writable + "<br/>"); //undefined  
document.write(des.configurable + "<br/>"); //false  
document.write(des.get + "<br/>"); //function() {return this.\_year;}

5.2 创建对象

/\*读取属性的特性\*/  
  
/\*工厂模式\*/  
//抽象了创建具体对象的过程，解决了创建多个相似对象的问题，但是没有解决对象识别的问题  
//即怎么样知道一个对象的类型  
**function** creatPerson(name,age,job){  
 **var** person = **new** Object();  
 person.name = name;  
 person.age = age;  
 person.job = job;  
 person.sayName = **function**(){  
 alert(**this**.name);  
 }  
 **return** person;  
}  
  
**var** person1 = creatPerson("zhuxiankang",38,"student");  
**var** person2 = creatPerson("tanghaixia",38,"student");  
alert(person1.name); //zhuxiankang  
  
/\*构造函数模式\*/  
**function** Person(name,age,job){  
 **this**.name = name;  
 **this**.age = age;  
 **this**.job = job;  
 **this**.sayName = **function**(){  
 alert(**this**.name);  
 }  
}  
  
**var** person = **new** Person("ZHUXIANKANG",25,"STUDENT");  
person.sayName(); //ZHUXIANKANG  
  
//与工厂函数相比，没有显示的创建对象，没有返回对象，直接将属性和方法赋值给了this对象，构造函数的首字母必须大写  
//构造函数是用来创建对象的函数  
//使用new操作符创建Person的新实例 执行过程 1.创建一个新对象 2.将构造函数的作用域赋给新对象（this指向了这个新对象）  
//3.执行构造函数中的代码 4.返回新对象  
//构造函数申请的对象都有一个constructor（构造函数）属性  
alert(person.constructor == Person); //true  
  
//检测对象类型还是instanceof操作符更可靠  
document.write((person **instanceof** Object) + "<br/>");//true  
document.write((person **instanceof** Person) + "<br/>");//true  
  
//创建构造函数可以将它的实例标识为一种特定的类型  
//多有对象均继承自Object对象  
//以这种方式定义的构造函数是定义在Global对象中的（浏览器中是window对象）  
  
//如果构造函数不适用new操作符，就跟普通函数没什么区别了  
Person("Victor",24,"doctor"); //this指代了window对象，而在全局作用域中调用一个函数，this对象总是指向了Global对象  
//Global对象在浏览器中就是window对象  
window.sayName(); //Victor  
alert(window.name); //Victor  
  
//如果利用call或者apply来扩展对象的作用域  
**var** oPerson = **new** Object();  
Person.call(oPerson,"Hugo",35,"doctor");  
oPerson.sayName(); //Hugo  
  
**var** person1 = **new** Person("tanghaixia",24,"nurse");  
alert(person.sayName == person1.sayName); //false  
  
//每次调用构造函数都是创建新实例，每个Person实例都包含了一个不同的Function实例  
//this.sayName = function(){alert(this.name);} 等价于 this.sayName = new Function("alert(this.name)");  
//然后创建两个完全是同样任务的Function是没有必要的，如果函数同样，我们只要一个函数就可以了  
//这个问题可以通过原型模式来解决

/\*原型模式\*/  
//prototype（原型）属性是一个指针，指向一个对象  
//该对象的用途是包含可以由特定类型的所有实例共享的属性和方法，从而解决每个实例的方法都不是同一个但是用途一样的问题  
//prototype就是通过调用构造函数而创建的那个对象实例的原型对象  
//使用原型对象的好处可以让所有对象实例共享它所包含的属性和方法  
//不必在构造函数中定义对象实例的信息，而是可以将这些信息直接添加到原型对象中  
  
**function** Person(){}; //空的构造函数  
//Person.prototype 对象实例的原型对象  
Person.prototype.name = "zhuxiankang";  
Person.prototype.sayName = **function**(){  
 write(**this**.name);  
}  
  
**var** oPerson = **new** Person(); //Array等原生对象就有原型属性和原型方法  
oPerson.sayName(); //zhuxiankang  
  
**var** oPerson1 = **new** Person();  
  
*/\*\*  
 \* function write(str){  
 return document.write(str + "<br/>");  
 }  
 \*/*write(oPerson.sayName == oPerson1.sayName); //true 虽然是两个不同的实例对象，但是调用了同一个方法，这个方法是原型对象的方法  
//原型对象的方法和属性对它们来说是同享的  
  
//原型对象  
//只要创建了一个新函数，就会在后台根据特定规则为新函数创建一个prototype属性，这个属性（在构造函数里的属性）指向函数的原型对象  
//在默认情况下，所有原型对象也会自动获得一个constructor（构造函数）属性，这个属性包含一个指向prototype属性所在函数(构造函数)的指针  
write(Person.prototype.constructor == Person); //true  
  
  
//创建了自定义的构造函数后，其原型对象的默认只会取得constructor属性，其余都是从Object继承而来  
//当调用构造函数创建新实例后，该实例内部将包含一个指向构造函数原型对象的指针（[[Prototype]] 内部属性，注意是实例的属性而非构造函数的属性）  
//脚本中没有标准的方式访问[[Prototype]]，但在一些浏览器诸如Firefox、Safari、Chrome在每个对象上都支持属性\_\_proto\_\_  
//这个指针连接存在于实例与构造函数的原型对象之间，不是实例与构造函数之间  
  
//调用构造函数创建的实例都有[[Prototype]]属性，但是无法访问，但是可以通过isPrototypeOf()方法来确定实例对象和对象原型之间是否存在这种关系  
write(Person.prototype.isPrototypeOf(oPerson)); //true  
  
//Object.getPrototypeOf = function(obj) {}; 返回的对象实际上是obj实例对象的原型对象  
write(Object.getPrototypeOf(oPerson) == Person.prototype); //true  
write(Object.getPrototypeOf(oPerson).name); //zhuxiankang  
  
  
//读取对象的原型和方法时，会执行搜索，首先搜索实例对象本身有没有给定的属性和方法，有则返回  
//如果没有找到，那么继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性和方法  
  
//不能通过对象实例重写原型中的值  
oPerson.name = "Victor";  
write(oPerson.name); //Victor 来自对象实例，屏蔽了原型对象的属性，根据搜索规则返回  
write(oPerson1.name); //zhuxiankang 来自原型对象，说明原型对象没有被重写  
  
//实例对象添加和原型对象的同名属性只会阻止访问原型中的属性，但不会修改属性，可以使用delete操作符删除实例对象的属性  
**delete** oPerson.name;  
write(oPerson.name); //zhuxiankang 又恢复了对原型对象属性的访问  
  
  
//Object.prototype.hasOwnProperty = function(propertyName) {}; 检测属性是否存在于实例中  
write(oPerson.hasOwnProperty("name")); //false  
  
oPerson.name = "Victor"; //实例对象的属性  
write(oPerson.hasOwnProperty("name")); //true ，参数必须是字符串  
  
  
//原型与in操作符  
write(oPerson.hasOwnProperty("name")); //true  
write("name" **in** oPerson); //true 来自实例  
write(oPerson1.hasOwnProperty("name")); //false  
write("name" **in** oPerson1); //true name是原型对象的属性, 无论实例或者原型 in访问  
  
//hasOwnProperty是检测属性是否存在于实例中，in可以检测属性是否存在于实例或者原型对象中  
//如果实例中不存在而原型中存在，那么即可判断属性属于原型对象而非实例对象  
  
**function** hasPrototypePrpperty(obj,property){  
 **return** !obj.hasOwnProperty(property) && (property **in** obj);  
}  
  
write(hasPrototypePrpperty(oPerson1,"name")); //true 来资原型的属性  
write(hasPrototypePrpperty(oPerson,"name")); //false 1.来自实例对象的属性 2.不存在属性  
  
//for-in循环  
//返回的是所有能够通过对象访问的、可枚举的(enumerated)的属性，包括实例和原型对象的中的属性  
//屏蔽了原型中不可枚举属性{将[Enumerable]标记为false的属性}的实例属性也可以在for-in中返回，例如上面的oPerson.name  
  
**var** o = {  
 toString: **function**(){ //屏蔽了Object.toSting（不可枚举）方法  
 **return** "My Object";  
 }  
};  
  
  
**for**(**var** prop **in** o){  
 **if**(prop == "toString"){  
 write("Found toString"); //Chrome IE9 11中显示了 IE8不行  
 }  
}  
  
//Object.keys = function(obj) {} 取得对象上所有可枚举的实例属性，参数为对象，返回一个包含所有可枚举属性的字符串数组  
**var** keys = Object.keys(Person.prototype); //构造函数所指向的原型对象  
write(keys); //name,sayName,注意keys是数组，可枚举的属性不包括原型对象的constructor属性，该属性默认不可枚举  
  
**var** p1 = **new** Person();  
p1.name = "Rod";  
p1.age = 23;  
  
keys = Object.keys(p1); //name,age  
write(keys);  
  
keys = Object.getOwnPropertyNames(Person.prototype); //无论是否可枚举  
write(keys); //constructor,name,sayName,其中constructor属性是不可枚举的  
  
//更简单的原型语法  
//原型对象添加属性和方法可以使用对象字面量的方法  
**function** Person() { //构造函数  
};  
  
//重写了默认的prototype对象，constructor属性也就变成了新对象的constructor属性（指向Object构造函数）  
Person.prototype = { //但此时constructor(构造函数指针)属性不再指向Person了  
 name:"Vq",  
 age: 23,  
 sayName:**function**(){  
 write(**this**.name);  
 }  
};  
  
  
write(Person.prototype.constructor == Person); //false  
write(Person.prototype.constructor == Object); //true  
  
**var** p2 = **new** Person();  
write(p2.constructor == Person); //false  
write(p2 **instanceof** Person); //true  
write(p2 **instanceof** Object); //true  
  
  
  
  
  
//constructor属性特意将它设置回适当的值  
**function** Person(){  
};  
  
Person.prototype = {  
 constructor:Person, //设置为Person  
 name:"Vq",  
 age:32,  
 sayName:**function**(){  
 write(**this**.name);  
 }  
};  
  
write(Person.prototype.constructor == Person); //true  
  
**var** des = Object.getOwnPropertyDescriptor(Person.prototype,"constructor");  
write(des.enumerable); //true constructor可枚举  
  
  
**var** p3 = **new** Person();  
**var** props = [];  
  
**for**(**var** prop **in** p3){ //不能缺少var  
 props.push(prop);  
}  
  
write(props); //constructor,name,age,sayName  
  
//查看可枚举属性也可以使用getOwnPropertyDescriptor读取属性的特性，P52页面有讲  
**var** des = Object.getOwnPropertyDescriptor(Person.prototype,"constructor");  
write(des.enumerable); //true constructor可枚举  
  
  
  
//如果是默认的constructor属性是不可枚举的  
**function** Per() {  
};  
Per.prototype.age = 23;  
**var** des = Object.getOwnPropertyDescriptor(Per.prototype,"constructor");  
write(des.enumerable); //false constructor不可枚举  
  
//说明通过特意设置为适当值是会改变constructor的可枚举属性的  
  
//配置为不可枚举 Object.defineProperty = function(obj,prop,desc) {}; ECAMScript5  
Object.defineProperty(Person.prototype,"constructor",{  
 enumerable:**false**,  
 value:Person  
});  
  
//查看可枚举属性也可以使用getOwnPropertyDescriptor读取属性的特性，P52页面有讲  
**var** des = Object.getOwnPropertyDescriptor(Person.prototype,"constructor");  
write(des.enumerable); //false constructor不可枚举啦，棒棒哒  
  
//使用Object.getOwnPropertyDecriptor()方法  
//constructor的属性默认是不可枚举的，但是现在通过for-in语句发现它的属性变成了可枚举

//原型的动态性  
//先创造实例后修改原型也能够立即从原型上体现出来  
**function** Person(){ //构造函数  
};  
  
**var** friend = **new** Person(); //先创造了实例  
  
Person.prototype.sayHi = **function**(){  
 write("hi");  
};  
  
friend.sayHi(); //hi 实例和原型之间的连接只不过是一个指针，搜索的时候先搜索实例对象的属性和方法，  
//没有，然后搜索原型对象的属性和方法，这个时候指针指向的内容变了，那么会立即反应。就跟C语言一样，指针指向的内容变了，就是本质变了。  
//也可以这么理解，搜索的方法在实例对象中没找到，就去找原型对象的属性和方法，虽然先创造了实例，但是原型对象的方法通过声明已经在内存中存在，  
//只要去搜索，指针总能找到这个内存的内容  
//注意以上所讲的指针是实例对象指向原型的指针[[Prototype]],或者也可以说是\_\_proto\_\_  
  
//调用构造函数的时候会为实例添加一个指向最初原型的[[Prototype]]指针  
//如果重写整个原型对象，即把原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系。  
//实例中的指针仅指向原型，而不指向构造函数  
  
**function** Person(){  
};  
  
Person.prototype.name = 11;  
**var** friend = **new** Person(); //调用构造函数产生了一个实例对象与原型连接的[[Prototype]]指针  
  
Person.prototype = { //重写了一个新的原型对象，并没有建立这个新的原型和旧的实例之间的指针连接  
 constructor:Person,  
 name:"zhuxiankang",  
 age:29  
};  
  
write(friend.name); //11 与旧的原型还保持连接状态  
write(friend.age); //undefined  
  
**var** f = **new** Person();  
write(f.age); //29

//原生对象的原型  
  
//原型模式不仅体现在创建自定义类型方面，所有的原生的引用类型都是采用这种模式创建的。  
  
write(Array.prototype.sort); //function sort() { [native code] }  
write(**typeof** Array.prototype.sort); //function  
  
//给基本包装类型String添加新的方法，不推荐这种做法  
String.prototype.startWith = **function**(text){  
 **return this**.indexOf(text) == 0; //索引的是不是字符串的开始  
}  
  
**var** msg = "hello world";  
write(msg.startWith("hello")); //true  
write(msg.startWith("hel")); //true  
  
//原型对象的问题  
//忽略了构造函数传递初始化参数这一环节  
//所有的实例在默认情况下都将获得相同的属性值  
  
//造成共享  
  
**function** Person(){  
};  
  
Person.prototype ={  
 constructor: Person,  
 age:24,  
 name:["ZHU","XIAN"]  
};  
  
**var** per1 = **new** Person();  
**var** per2 = **new** Person();  
per1.age = 25;  
write(per2.age); //24  
per1.name.push("KANG"); //修改了Per1.name引用的数组  
write(per2.name); //ZHU,XIAN,KANG! 由于name数组存在于Person.prototype而非Per1中，Per2.name（与Per1.name）指向同一个数组  
//所以Per1.name的变化立马反应在了Per2.name中  
  
//共享对于相同方法的函数来说非常合适  
//对于基本类型的属性例如上面的age也是没多大问题  
//如果包含引用类型值得属性，问题就来了！  
  
//所以很少有人单独使用原型模式

//  
/\*组合使用的构造函数模式和原型模式\*/  
//创建自定义类型的最常见方式就是组合使用构造函数与原型模式  
//构造函数定义实例属性  
//原型模式定义方法和共享的属性  
//可以向构造函数传递参数  
//目前产用的最流行的定义引用类型的方式  
  
**function** Person(name,age,job){ //构造函数模式  
 **this**.name = name,  
 **this**.age = age,  
 **this**.job = job,  
 **this**.friends = ["Bob","Jonny"]  
}  
  
Person.prototype = { //原型模式  
 constructor:Person,  
 sayName:**function**(){  
 write(**this**.name);  
 }  
};  
  
**var** oPerson = **new** Person("zhuxiankang",23,"student");  
**var** oPerson1 = **new** Person("victor",23,"doctor");  
  
oPerson.friends.push("Kit");  
write(oPerson.friends); //Bob,Jonny,Kit  
write(oPerson1.friends);//Bob,Jonny  
  
oPerson.sayName(); //zhuxiankang  
oPerson1.sayName();//victor  
write(oPerson.sayName == oPerson1.sayName);//true

/\*动态原型模式\*/  
**function** Person(name,age,job){  
 //属性  
 **this**.name = name,  
 **this**.age = age,  
 **this**.job = job,  
 //方法  
 **if**(**typeof this**.sayName != "function"){  
 Person.prototype.sayName = **function**(){  
 write(**this**.name);  
 };  
 }  
}  
  
//只有在sayName方法不存在的情况下才有可能将它添加到原型中  
//只会在初次调用构造函数时才会执行  
  
//注意：不能使用对象字面量重写原型，会切断已有的实例（重写之前）的新原型的连接

/\*寄生构造函数模式\*///其实类似工厂模式 尽量不要使用这种模式  
  
**function** Person(name,age,job){  
 **var** Per = **new** Object();  
 Per.name = name;  
 Per.age = age;  
 Per.job = job;  
 Per.sayName = **function**(){  
 write(**this**.name);  
 };  
  
 **return** Per;  
}  
  
//寄生  
**var** Per1 = **new** Person("Victor",25,"doctor"); //使用了new操作符，使用的包装函数叫做构造函数  
Per1.sayName(); //Victor  
  
//工厂  
**var** Per2 = Person("Jonny",23,"student"); //普通函数  
Per2.sayName(); //Jonny  
  
  
**function** SpecialArray(){  
 **var** arr = **new** Array();  
 //Function.prototype.apply = function(thisArg,argArray) {};  
 //为什么要用apply,是因为平常push接收的是一个或多个参数，但是全部列举是不可能的，用apply可以把接收到的参数以数组的形式传入  
 arr.push.apply(arr,arguments);  
 //为什么这里是arr ,而不是arr.push，第一个参数是在其中运行的函数的作用域，当前也可以是对象，详见P40，扩充了函数赖以运行的作用域  
 //第二参数是参数数组，arguments是SpecialArray的内部属性，是一个参数对象，数组  
 //apply的作用说穿了就是给函数传入全局例举的参数不现实，而改成传入数组参数  
  
  
 arr.toPipedString = **function**(){  
 **return this**.join("|"); //一定要返回数组  
 };  
 **return** arr;  
}  
  
  
**var** arr = **new** SpecialArray("red","blue","yellow");  
write(arr.toPipedString()); //red|blue|yellow  
  
//构造函数返回的对象与在构造函数外部创建的对象没有什么不同，不能依赖instanceof操作符来确定对象的类型  
write(arr **instanceof** SpecialArray); //false result = variable instanceof constructor P15  
write(arr **instanceof** Array); //true

/\*稳妥造函数模式\*/  
**function** Person(name,age,job){  
 **var** o = **new** Object();  
  
 o.sayName = **function**(){  
 write(name); //没有使用this，除了使用sayName方法，没有其他方法可以访问name属性  
 };  
  
 **return** o;  
}  
  
**var** friend = Person("Victor",23,"student");  
write(friend.name); //undefined  
friend.sayName(); //Victor  
  
//与寄生构造函数类似，这种模式创建的对象与构造函数之间也没有什么关系  
  
//但是没有办法访问传入到构造函数中的原始数据，比较安全

5.3 继承

/\*原型链\*/  
  
//继承分为接口继承和实现继承  
//ECMAScript只支持实现继承，实现继承主要依靠原型链  
//原型链的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法  
//每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针[[Prototype]]  
//假如我们让原型对象等于另外一个类型的实例

，此时的原型的对象将包含一个指向另一个原型对象的内部指针[[Prototype]]  
//相应的，另一个原型中也包含着一个指向另一个构造函数的指针，假如另一个原型又是另一个类型的实例，就构成了实例与原型的链条  
//这就是原型链  
  
**function** SuperType(){ //构造函数  
 **this**.property = **true**;  
}  
  
SuperType.prototype.getSuperValue = **function**(){  
 **return this**.property;  
}  
  
**function** SubType(){ //构造函数  
 **this**.subProperty = **false**;  
}  
  
write(SubType.prototype.constructor); //function SubType(){ //构造函数 this.subProperty = false; }  
  
SubType.prototype = **new** SuperType(); //当调用SuperType构造函数创建新实例后  
//该实例内部将包含一个指向SuperType构造函数原型对象的指针[[Prototype]]  
//注意是实例的属性,也就是SubType的原型对象有一个内部指针，指向了SuperType构造函数的原型对象  
//这个指针连接存在于实例与构造函数的原型对象之间  
//重写了SubType的原型对象，代之一个新类型的实例  
//前面说过，如果原型对象以对象字面量形式创建的新对象，constructor属性便不在指向原来的构造函数了，而是指向了Object构造函数  
//这里重写了原型对象后，constructor属性就不再指向SubType构造函数了  
  
  
write(SubType.prototype.constructor); //function SuperType(){ //构造函数 this.property = true; }  
//SubType的原型指向了另一个对象SuperType的原型，这个原型对象的constructor属性指向的是SuperType  
//所以Subtype的constructor属性指向的也是SuperType  
  
  
SubType.prototype.getSubValue = **function**(){  
 **return this**.subProperty;  
}  
  
**var** instance = **new** SubType();  
write(instance.getSuperValue()); //true  
//读取对象的原型和方法时，会执行搜索，首先搜索实例对象本身有没有给定的属性和方法，有则返回  
//如果没有找到，那么继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性和方法  
  
//SubType继承了SuperType,原来存在于SuperType的实例中的所有属性和方法，现在也存在于SubType.prototype中了。  
//给SubType.prototype添加一个方法就等与在继承了SuperType的属性和方法的基础上又添加了一个新方法。  
  
//没有使用SubType默认提供的原型，而是给它换了一个新原型，这个新原型就是SuperType的实例。  
//新原型不仅拥有了SuperType的实例所拥有的全局属性和方法，还拥有了一个内部指针[[Prototype]],指向了SuperType的原型  
//instance指向了SubType的原型，SubType的原型又指向了SuperType的原型。  
  
//property是一个实例属性，getSuperValue()是一个原型方法  
//property位于SubType.prototype中，因为SubType.prototype现在是SuperType的实例  
  
write(instance.constructor); //function SuperType(){ //构造函数 this.property = true; }  
//var instance = new SubType();  
//但是instance指向的是SuperType()  
  
write(SubType.prototype.constructor);//function SuperType(){ //构造函数 this.property = true; }  
  
**var** instan = **new** SuperType();  
write(instan.property); //true  
write(instan.constructor); //function SuperType(){ //构造函数 this.property = true; }  
  
//实现原型链，本质上就是扩展了原型搜索机制  
//读取模式访问一个实例属性时。首先搜索该属性是否在实例中存在，如果没有，则继续搜索实例的原型。  
//在原型链中，搜索过程就是沿着原型链向上进行的  
  
//1.搜索实例 2.搜索实例的原型，同时也是另一个类型的实例 3.搜索另外一个类型的原型，同时也是......  
//一直向上搜索，直到第一次搜索到属性或者方法为止  
//搜索不到，到原型链的末端停止  
  
//Object  
//所有的引用类型都继承至Object  
//而这个继承也是通过原型链实现的  
//所有函数的默认原型都是Object的实例，一次默认原型都有一个内部指针[[Prototype]]指向Object.prototype  
//这也是自定义类型都会继承toString(),valueOf()方法的原因  
write(SubType.prototype); //[object Object]

/\*确定原型和实例的关系\*/  
  
**function** SuperType(){ //构造函数  
 **this**.property = **true**;  
}  
  
SuperType.prototype.getSuperValue = **function**(){ //原型方法  
 **return this**.property;  
}  
  
  
**function** SubType(){ //构造函数  
 **this**.subProperty = **false**;  
}  
  
  
SubType.prototype = **new** SuperType(); //实例,这一条构成了原型链  
  
**var** instance = **new** SubType(); //实例  
//instanceof 测试实例与原型链中出现过的构造函数，结果就是true  
//由于原型链的关系，可以说instance是Object、SuperType或SubType中的任何一个类型的实例  
write(instance **instanceof** Object); //true  
write(instance **instanceof** SuperType); //true  
write(instance **instanceof** SubType); //true  
  
//isPrototypeOf()方法 检测原型链所派生的实例的原型  
//Object.prototype.isPrototypeOf = function(o) {};  
write(Object.prototype.isPrototypeOf(instance)); //true  
write(SuperType.prototype.isPrototypeOf(instance)); //true  
write(SubType.prototype.isPrototypeOf(instance)); //true  
  
  
/\*谨慎的定义方法\*/  
  
//添加新方法  
SubType.prototype.getSubValue = **function**(){  
 **return this**.subProperty;  
}  
  
//重写超类型中的方法  
SubType.prototype.getSuperValue = **function**(){  
 **return false**;  
}  
  
**var** instance = **new** SubType();  
write(instance.getSuperValue()); //false 重写了超类型中的方法屏蔽了原来的方法  
  
**var** instan = **new** SuperType();  
write(instan.getSuperValue()); //true 调用原来的方法  
//其实以上就是根据原型链的搜索机制

//通过原型链实现继承时，不能使用对象字面量创建原型方法，这样就会重写原型链  
//使用对象字面量创建原型方法，使原型的constructor属性指向了Object构造函数  
//从而打断了原来的原型链  
  
**function** SuperType(){ //构造函数  
 **this**.property = **true**;  
}  
  
SuperType.prototype.getSuperValue = **function**(){ //原型方法  
 **return this**.property;  
}  
  
  
**function** SubType(){ //构造函数  
 **this**.subProperty = **false**;  
}  
  
  
SubType.prototype = **new** SuperType(); //实例,这一条构成了原型链  
  
  
**var** Sub = **new** SubType();  
write(Sub.getSuperValue()); //true  
write(Sub.constructor); //function SuperType(){ //构造函数 this.property = true; }  
  
//使用对象字面量创建原型对象  
SubType.prototype = {  
 getSubValue:**function**(){  
 **return this**.subProperty;  
 },  
  
 getOtherValue:**function**(){  
 **return false**;  
 }  
};  
  
**var** Sub1 = **new** SubType();  
write(Sub1.constructor); //function Object() { [native code] },此时已经再是SuperType构造函数了  
write(Sub1.getSuperValue()); //Uncaught TypeError: Sub1.getSuperValue is not a function  
//原型链已经断了

/\*原型链的问题\*/  
//就是实例共享引用类型值属性的问题  
//原型模式中共享方法和基本类型值属性是么有问题的  
//但是实例如果共享引用类型的值作为属性是有问题的，因为其实共享的是指针变量，而不是副本，真正的内存都是同一个  
//如果其中一个指向内存的值发生了变化，会立马反应给实例的指针的，只要操作了其中一个指针的内容，其他指针指向的内容都变了！  
//所以才有了组合使用构造函数和原型模式  
//在构造函数中定义属性，在原型对象中定义方法（方法共享没有问题，当然有些是可以共享的属性诸如String的length等）  
  
  
//但是在原型链中，原型实际上回变成另一个类型的实例，于是实例所拥有的属性也就变成了原型拥有的属性，  
//就相当于本来要避免把属性放在原型对象中，现在却无形中不得不把属性放在了原型对象中，从而实现了属性共享  
  
**function** SuperType(){  
 **this**.colors= ["red","yellow","blue"];  
}  
  
**function** SubType(){  
}  
  
SubType.prototype = **new** SuperType(); //SubType的原型对象继承了引用类型属性colors  
  
**var** instance = **new** SubType();  
instance.colors.push("black");  
  
**var** instan = **new** SubType();  
write(instan.colors); //red,yellow,blue,black  
  
//问题出来啦，引用类型的属性之间实现了共享，因为它们其实共享的是同一个内存  
  
//实践中很少单独使用原型链  
//在创建子类型的实例时，不能向超类型的构造函数传递参数。  
//没有办法在不影响所有对象实例的情况下，给超类型的构造函数传递参数

/\*借用构造函数\*/  
//解决原型中包含引用类型值得问题  
//借用构造函数有时候也叫伪造对象或经典继承  
//在子类的构造函数中调用超类型的构造函数  
  
**function** SuperType(){  
 **this**.colors = ["red","yellow"]; //构造函数内创建属性，每个实例都会有自己的一份实例属性的副本  
}  
  
  
**var** a = **new** SuperType();  
a.colors.push("black");  
write(a.colors); //red,yellow,black  
  
**var** b = **new** SuperType();  
write(b.colors); //red,yellow  
  
//因为调用了构造函数创建了两个不一样的对象，所以两个引用类型的副本互不影响  
  
SuperType.prototype.name = ["red","yellow"];  
  
**var** c = **new** SuperType();  
**var** d = **new** SuperType();  
c.name.push("black");  
write(c.name); //red,yellow,black  
write(d.name); //red,yellow,black 共享了原型对象的属性

/\*借用构造函数\*/  
//解决原型中包含引用类型值的问题  
//借用构造函数有时候也叫伪造对象或经典继承  
//在子类的构造函数中调用超类型的构造函数  
  
**function** SuperType(){ //父构造函数  
 **this**.colors = ["red","yellow"]; //构造函数内创建属性，每个实例都会有自己的一份实例属性的副本  
}  
  
//使用new操作符调用构造函数实际上经历4步 1.创建一个新对象 2.将构造函数的作用域赋给新对象（this指向新对象）  
//3.执行构造函数的代码，对新对象进行添加属性和方法，例如this.colors就是新对象.colors  
//4.返回一个初始化好的新对象  
  
**function** SubType(){ //子构造函数  
 //继承了SuperType  
 SuperType.call(**this**); //注意这里的this并不是window，构造函数内的this和普通全局环境的函数的this不同  
 //自我理解  
 //由于调用构造函数实际上经历了4步，其中第二部是将构造函数的作用域赋给新对象，即this指向了新对象，这个对象是关联SubType的  
 //虽然这个新对象没有SuperType的方法，但是通过call将方法SuperType借用给了这个新对象，于是又调用了SuperType构造函数  
 //SuperType函数中的上下文执行环境发生了改变，this指代的并不是关联SuperType构造函数4步走的新对象，而是SubType中4步走的对象  
 //于是返回SubType构造函数创建的对象时对象拥有了colors属性  
 //实现了继承SuperType  
  
 //SuperType(); 这样是不行的 执行下面的write(instance.colors); //undefined  
}  
  
  
**var** instance = **new** SubType();  
// write(instance.colors); //undefined  
  
instance.colors.push("black");  
write(instance.colors); //red,yellow,black  
  
**var** inst = **new** SubType();  
inst.colors.push("red");  
write(inst.colors); //red,yellow,red  
  
//两个不同的副本，在新SubType对象上执行SuperType()函数中定义的所有对象初始化代码  
//结果SubType的每个实例都会具有自己的colors属性的副本了。  
  
**function** Super(name,age){  
 **this**.name = name;  
 **this**.age = age;  
}  
  
**function** Sub(){  
 //继承了Super,同时还传递了参数  
 Super.apply(**this**,arguments);  
  
 //实例属性  
 **this**.job = arguments[2];  
}  
  
**var** Sub = **new** Sub("zhuxiankang",23,"doctor");  
  
write(Sub.name); //zhuxiankang  
write(Sub.age); //23  
write(Sub.job); //doctor  
  
  
//借用构造函数的很大优势可以在子类型构造函数中向父类型构造函数传递参数  
  
//但是如果仅仅是借用构造函数，无法避免构造函数模式存在的问题，方法都在构造函数中定义，方法共享就无法实现了  
//在父类型的原型中定义的方法，对子类型而言是不可见的  
//单独使用借用构造函数的技术很少用  
  
  
//call和apply加深理解  
//例如  
  
**function** sayColor(){  
 write(**this**.color); //全局环境的this默认指代window对象.如果不是默认可能会改变  
}  
  
window.color = "red";  
**var** o = {color:"blue"};  
  
sayColor(); //red  
  
sayColor.call(**this**); //red this指代window环境  
sayColor.call(o); //blue 传入的是o对象，改变了sayColor函数中的上下文对象this 从window ——> o  
  
//此时的o对象与sayColor方法可是没有任何直接的耦合关系，sayColor方法可不是o对象的方法  
//简直是”无中生有”,对象的方法可以任意指派,而对象本身一直都是没有这方法的,  
//注意是指派,通俗点就是,方法是借给另一个对象的调用去完成任务,原理上是方法执行时上下文对象改变了.

5.4 组合继承（重要）

/\*组合继承\*/ //最常用的继承模式  
  
//将原型链和借用构造函数技术组合在一起  
//即解决了原型链中方法可以共享在借用构造函数中的限制问题  
//也解决了引用类型值属性共享同一个副本在原型链中的问题  
  
**function** SuperType(name){  
 **this**.name = name;  
 **this**.colors = ["red","blue"];  
}  
  
SuperType.prototype.sayName = **function**(){  
 write(**this**.name);  
}  
  
**function** SubType(name,age){  
 SuperType.call(**this**,name);//借用构造函数  
 **this**.age = age;  
}  
  
//继承方法  
SubType.prototype = **new** SuperType(); //原型链，  
write(SubType.prototype.constructor); //function SuperType(name){ this.name = name; this.colors = ["red","blue"]; }  
SubType.prototype.constructor = SubType; //调整constructor构造函数指针  
  
SubType.prototype.sayAge = **function**(){  
 write(**this**.age);  
};  
  
  
**var** Sub = **new** SubType("Victor",23); //可以使用借用构造函数技术传递参数给父构造函数  
//实现了原型链的技术  
Sub.sayName(); //调用继承来的方法  
Sub.sayAge(); //调用了该实例原型对象的方法  
  
//实现了借用构造函数的技术  
write(Sub.colors); //red,blue  
  
//colors虽然是引用类型值属性  
//但是这里创建了两个不同的副本，解决了引用类型值属性公用一个内存的问题  
**var** Sub1 = **new** SubType("Hugo",23);  
Sub1.colors.push("yellow");  
write(Sub1.colors); //red,blue,yellow  
write(Sub.colors); //red,blue  
  
  
//回顾一下  
//构造函数模式  
//可以创建不同实例属性的副本，不会共享副本，包括引用值类型的属性，但是不能共享方法，而是实例对象各自创建了自己的方法  
//原型模式  
//原型模式中的引用值类型的属性其实是共享同一个内存副本，如果改变了这个内存副本的值，那么所有实例对象的该属性都改变了，但是可以共享方法，所有对象引用的是同一个方法  
  
//原型链  
//让一个构造函数的原型对象指向了另一个类型的实例  
//那么这个原型对象也拥有了那个实例的属性和方法，包括引用值类型的属性，从而实现了方法和属性的继承  
//由于原型模式中存在引用值类型共享同一个内存副本的问题  
//也把这个问题带到了原型链中  
  
//借用构造函数  
//解决了继承中的引用值类型共享的问题，但是继承的方法和属性必须写在构造函数内，这样的话方法又不能共享了  
  
//组合继承  
//属性的继承使用借用构造函数技术  
//方法的继承使用原型链技术  
//即解决了引用值类型共享的问题  
//又实现了方法的共享  
//是目前最流行的继承模式

5.5 寄生组合式继承（一）

/\*原型式继承\*/  
//借助原型可以基于已有对象创建新对象  
//同时不必因此创建自定义类型（所谓的自定义类型是不是构造函数类型？）  
  
**function** object(o){  
 **function** F(){}  
 F.prototype = o;  
 **return new** F(); //返回的是一个实例对象，原型对象是传入的参数对象  
}  
  
//object对传入的参数进行了浅复制  
  
**var** person = {  
 name: "Victor",  
 friends:["Bob","Jonny"]  
}  
  
**var** anOtherPerson = object(person);  
write(anOtherPerson.name); //Victor  
anOtherPerson.friends.push("Jack");  
write(anOtherPerson.friends); //Bob,Jonny,Jack  
write(person.friends); //Bob,Jonny,Jack  
  
//跟原型模式一样，引用类型的值的属性在不同的实例对象中共用一个内存，毕竟实例对象的引用类型属性都是指针  
**var** yetAnoterhPerson = object(person);  
yetAnoterhPerson.name = "Hugo";  
write(yetAnoterhPerson.name); //Hugo  
write(person.name); //Victor  
  
//基本类型值是两个不同的副本，和引用类型值属性是不同的，所以他们是互不影响的  
yetAnoterhPerson.friends.push("kite");  
write(person.friends);//Bob,Jonny,Jack,kite 都是共享一个内存的  
  
//Object.create()方法  
//两个参数，1.用作新对象原型的对象 2.为新对象定义额外属性的对象（可选）  
//在传入一个参数的情况下，Object.create()与object()方法的行为相同  
  
**var** per = {  
 name:"Jack",  
 friends:["Jon","Tom"]  
}  
  
**var** per1 = Object.create(per);  
write(per1.friends); //Jon,Tom  
per1.friends.push("Jim");  
write(per.friends); //Jon,Tom,Jim  
write(per1.friends); //Jon,Tom,Jim  
  
**var** per2 = Object.create(per);  
write(per2.friends); //Jon,Tom,Jim  
per2.friends.push("Jom");  
write(per2.friends); //Jon,Tom,Jim,Jom  
  
//覆盖同名属性 第二参数和 Object.defineProperties的第二参数类似，详细见P50  
**var** per3 = Object.create(person,{  
 name:{  
 value:"Victor-2"  
 }  
});  
  
write(per3.name); //Victor-2  
write(person.name); //Victor

/\*寄生式继承\*/  
  
**function** object(o){  
 **function** F(){}  
 F.prototype = o;  
 **return new** F(); //返回的是一个实例对象，原型对象是传入的参数对象  
}  
  
  
**function** creatAnother(original){  
 **var** clone = object(original);  
 clone.sayHi = **function**(){  
 write("Hi");  
 };  
 **return** clone;  
}  
  
**var** person = {  
 name:"Victor",  
 age:23,  
 friends:["Jack","Max"]  
};  
  
**var** anotherPerson = creatAnother(person);  
anotherPerson.sayHi(); //Hi  
  
//新对象不仅拥有person的属性和方法，而且还拥有自己的方法  
//由于方法不是共享的，与构造函数类似，所以会造成效率低下  
  
  
/\*寄生组合式继承\*/  
//组合继承也有缺点，就是会调用两次超类型（父类型）构造函数  
//1次是在创建子类型原型的时候，另外一次是在子类型构造函数内部  
  
**function** Super(name){  
 **this**.name = name;  
 **this**.colors = ["red","yellow","blue"];  
}  
  
  
Super.prototype.sayName = **function**(){  
 write(**this**.name);  
}  
  
  
**function** Sub(){  
 //继承Suoer  
 Super.call(**this**, arguments[0]); //第二次调用Super构造函数，覆盖了原型对象中的同名属性（根据原型链的搜索机制）  
 **this**.age = arguments[1];  
}  
  
Sub.prototype = **new** Super(); //第一次调用Super Sub构造函数的原型对象拥有了name和colors两个属性  
Sub.prototype.constructor = Sub;

**var** Sub1 = **new** Sub("Peter",23);  
write(Sub1.name); //Peter  
write(Sub1.age); //23  
write(Sub1.colors); //red,yellow,blue  
  
//write(Sub1.prototype.name);  
  
write(Sub.prototype.colors); //red,yellow,blue 属性还在的  
  
**var** Super1 = **new** Super();  
Super1.colors.push("green");  
write(Super1.colors); //red,yellow,blue,green  
  
Sub.prototype.colors.push("white");  
write(Sub.prototype.colors); //red,yellow,blue,white  
  
//注意引用类型值共享只会在原型的属性中才会发生  
//构造函数中是会有不同的副本的，colors是在Super构造函数中，虽然Sub对应的原型对象也拥有该属性  
//但是它和Super实例的同名属性是不同的副本（不同的内存）  
  
//但是问题是Sub的实例和原型对象中都有了这两个属性  
//只是在使用Sub实例中的name和colors属性时屏蔽了原型对象中的同名属性，根据的是原型链的搜索机制  
  
//寄生组合式解决了这个问题

5.5 寄生组合式继承（二）

/\*寄生组合式继承\*/  
  
  
//返回实例对象，传入的参数是实例对象的原型对象，这样实例对象就有个内部指针[[Prototype]]指向了原型对象  
//实例对象共享了原型对象o的所有属性和方法  
//必须有一个对象是另外一个对象的基础  
//返回的新对象将o作为原型  
**function** object(o){  
 **function** F(){}  
 F.prototype = o;  
 **return new** F(); //返回的是一个实例对象，原型对象是传入的参数对象  
}  
  
**function** inheritPrototype(subType,superType){  
 **var** prototype = object(superType.prototype); //创建对象  
 //返回的新对象prototype将superType.prototype作为原型，即新对象有一个内部指针[[prototype]]指向了superType.prototype  
 //但是没有调用SuperType构造函数  
 //新对象有了SuperType的原型对象的方法和属性，注意新对象是没有SuperType构造函数的实例的方法和属性的  
 //write(prototype.sayName()); //Super.prototype's function,有SuperType的原型对象的方法和属性  
 //write(prototype.job); //undefined  
 //write(prototype.colors); //undefined  
 //因为SuperType构造函数的方法和属性利用借用构造函数技术继承了，所以不需要再次继承，只需要继承原型的方法和属性即可  
  
 //write(prototype.prototype);//undefined 只有内部指针，这样是做不到的，实例没有指向原型对象的可显示指向的指针，只有一个内部的[[Prototype]]  
 //write(prototype.\_\_proto\_\_);//[object Object]  
  
 //与SubType.prototype = new SubType();不同的是这一句代码不仅有一个指向SubType构造函数的原型对象的内部指针[[Prototype]]  
 //还调用了一次构造函数，从而使SubType.prototy中多了SubType实例的属性和方法  
  
  
  
 //write(prototype.constructor); //function SuperType(name){ this.name = name; this.colors = ["red","yellow"]; this.job = "doctor"; }  
 //查看object()  
 //F.prototype = o;  
 //F = super = 传入的参数 SuperType  
 //F.prototype = superType.prototype = SuperType.prototype;  
 //return new F() = return new SuperType()  
 //prototype.constructor == SuperType  
  
  
  
 prototype.constructor = subType; //增强对象  
 //使prototype的构造函数指针指向为SubType构造函数  
  
  
  
  
 subType.prototype = prototype; //指定对象  
 // 类似于subType.prototype = new SuperType()  
 // 但是不同的是prototype没有SuperType实例对象的属性和方法  
 // 只有SuperType的原型对象的属性和方法  
 // 所以subType.prototype避免了二次调用SuperType构造函数  
 // SubType.prototype也没有SuperType的实例对象的属性和方法  
 //subType.prototype.sayName();//Super.prototype's function 有SuperType的原型对象的方法和属性  
 // write(subType.prototype.job); //undefined  
  
 //SubType构造函数的原型设置为prototype  
 //prototype.constructor = subType  
 //subType.prototype = prototype  
 //subType.prototype.constructor = subType  
  
  
  
 //write(subType.prototype.constructor);  
 //如果屏蔽prototype.constructor = subType  
 //function SuperType(name){ this.name = name; this.colors = ["red","yellow"]; this.job = "doctor"; }  
 //如果不屏蔽prototype.constructor = subType  
 //function SubType(){ //继承SuperType SubType.call(this,arguments[0]); this.age = arguments[1]; }  
  
}  
  
  
  
**function** SuperType(name){  
 **this**.name = name;  
 **this**.colors = ["red","yellow"];  
 **this**.job = "doctor";  
}  
  
SuperType.prototype.sayName = **function**(){  
 write("Super.prototype's function");  
}  
  
**function** SubType(){  
 //继承SuperType  
 SuperType.call(**this**,arguments[0]);  
 **this**.age = arguments[1];  
}  
  
//SubType.prototype = new SubType(); //这里本来有一次调用父类的构造函数的，遗留的问题是使原型对象多了父类实例的属性  
//SubType.prototype.constructor = SubType; //这两句被下面替代了  
  
  
inheritPrototype(SubType,SuperType);  
//解决了原型对象多了父类实例对象的属性的问题  
//除了实现了以上两句的功能以外，使SubType的原型摒弃了SuperType实例对象的属性和方法  
  
  
SubType.prototype.sayAge = **function**(){  
 write(**this**.age);  
};  
  
  
**var** Sub1 = **new** SubType("Victor",23);  
**var** Sub2 = **new** SubType("Hugo",24);  
  
write(SubType.prototype.job); //undefined  
//如果屏蔽inheritPrototype(SubType,SuperType);，使用前两句 显示为doctor  
//说明SubType构造函数对应的原型对象已经不再有SuperType的实例对象的属性和方法了  
  
//利用借用构造函数技术继承了SuperType构造函数的初始化属性  
write(Sub1.name); //Victor  
//初始化了SubType自己的实例属性  
write(Sub1.age);//23  
  
//确实也继承了SuperType的原型对象的方法  
Sub1.sayName(); //Super.prototype's function  
  
  
//两个实例对象也不在共享同名引用类型值属性内存空间了，而是各自产生了一个互不影响的内存空间  
Sub1.colors.push("yellow");  
write(Sub1.colors); //red,yellow,yellow  
write(Sub2.colors); //red,yellow  
  
  
**var** Super1 = **new** SubType("Jim");  
**var** Super2 = **new** SubType("Tom");  
  
write(Super1.colors); //red,yellow  
  
SuperType.prototype.sayName = **function**(){  
 write("Super.prototype's change function");  
}  
  
Sub1.sayName(); //Super.prototype's change function  
//因为赋值的是指针，所以一旦原型对象的方法发生了改变立马反应到了实例对象的继承的方法  
  
Super1.sayName(); //Super.prototype's change function  
  
  
  
  
  
//只调用了一次SuperType构造函数，并且避免了SubType.prototype上创建了不必要的、多余的属性，于此同时，原型链还能保持不变  
//寄生组合式继承是引用类型最理想的继承形式

第六章 函数表达式

6.1 函数声明与函数表达式的区别

//函数声明  
**function** sayName(){  
}  
write(sayName.name); //sayName  
  
//函数声明提升，执行代码之前会先读取函数声明  
sayHi(); //Hi  
  
**function** sayHi(){  
 write("Hi");  
}  
  
//函数表达式  
//oFunction(); //不能再函数表达式之前调用它，就象C语言的变量声明，不能在声明前去用他，oFunction在这里是一个函数对象（变量）  
// Uncaught TypeError: oFunction is not a function  
  
**var** oFunction = **function**(){ //匿名函数，匿名函数的name属性是空字符串  
 write("oFunction");  
}  
  
oFunction();//在函数表达式之后可以调用，oFunction

6.2 递归

**function** factorial(num){  
 **if**(num < 1){  
 **return** 1;  
 }**else**{  
 **return** num \* factorial(num-1);  
 }  
}  
  
write(factorial(4)); //24  
  
//var factor = factorial;  
//factorial = null;  
//write(factor(4)); //Uncaught TypeError: factorial is not a function  
  
//因为factorial赋为了空指针，所以找不到了factorial函数中的factorial，因为此时它是null  
  
  
**function** factorial1(num){  
 **if**(num < 1){  
 **return** 1;  
 }**else**{  
 **return** num \* arguments.callee(num - 1); //arguments.callee正在执行的函数的指针，在严格模式下行不通  
 }  
}  
  
**var** factor1 = factorial1;  
factorial1 = **null**;  
write(factor1(4)); //24 ,factorial1函数内部不影响  
  
//使用命名函数表达式，严格非严格模式都可以行得通  
**var** factorial2 = (**function** f(num){  
 **if**(num < 1){  
 **return** 1;  
 }**else**{  
 **return** num \* f(num - 1);  
 }  
});  
  
write(factorial2(3)); //6  
  
**var** factor2 = factorial2;  
write(factor2(4)); //24

6.3 闭包

//闭包是指有权访问另一个函数作用域中的变量的函数  
//创建闭包的常见方式就是在一个函数内部创建另一个函数  
  
**function** campareFunction(propertyName){  
 **return function**(obj1,obj2){ //一个匿名的内部函数  
 **var** value1 = obj1[propertyName];  
 **var** value2 = obj2[propertyName];  
  
 **if**(value1 < value2){  
 **return** -1;  
 }**else if**(value1 > value2){  
 **return** 1;  
 }**else**{  
 **return** 0;  
 }  
 }  
}  
  
//即使内部的匿名函数被返回了，并且在其他地方被调用了，但它仍然可以访问propertyName  
//因为内部函数中的作用域链包含了campareFunction()的作用域  
  
  
//作用域链  
//当某个函数被调用时会创建一个执行环境及相应的作用域链。  
//然后使用arguments和其他命名参数的值来初始化函数的活动对象。但是在作用域链中，外部函数的活动对象始终处于第二位  
//外部函数的外部函数的活动对象处于第三位.....直至作为作用域链终点的全局执行环境  
  
//例如  
**function** compare(value1,value2){  
 **if**(value1 < value2){  
 **return** -1;  
 }**else if**(value1 > value2){  
 **return** 1;  
 }**else**{  
 **return** 0;  
 }  
}

**var** result = compare(5,10);  
  
//第一次调用compare函数时会创建包含this、arguments、value1和value2的活动对象  
//全局执行环境的变量对象（包含this、result和compare）在compare()执行环境的作用域链中则处于第二位

/\*  
compare执行环境 <--------------------------------------------------------  
(scope chain) --------> scope Chain |  
 1 --------------> Global variable object |  
 0 ------ compare ------  
 | result undefined  
 |  
 |--------> compare() activation object  
 arguments [5,10]  
 value1 5  
 value2 10  
\*/

//后台每个执行环境都有一个表示变量的对象----变量对象，全局环境的变量对象始终存在，  
//而像compare()函数这样的局部环境的变量对象，则只在函数的执行过程中存在  
//在创建compare()函数时，会创建一个预先包含全局变量对象的作用域链，这个作用域链被保存在内部的[[Scope]]属性中  
//当调用compare()函数时，会为函数创建一个执行环境，然后通过赋复制函数的[[Scope]]属性中的对象构建执行环境的作用域链  
//此后又有一个compare活动对象(在此作为变量对象使用)被创建并被推入执行环境作用域的<前端>！  
//在这里的compare执行环境作用域链包含两个变量对象，本地活动对象和全局变量对象。  
//作用域链本质上是一个指向变量对象的指针列表，它只引用但不实际包含变量对象  
  
  
//一般来说，函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域（Global variable object）。  
//但是闭包的情况却不同。

**function** campareFunction(propertyName){  
 **return function**(obj1,obj2){ //一个匿名的内部函数  
 **var** value1 = obj1[propertyName];  
 **var** value2 = obj2[propertyName];  
  
 **if**(value1 < value2){  
 **return** -1;  
 }**else if**(value1 > value2){  
 **return** 1;  
 }**else**{  
 **return** 0;  
 }  
 }  
}  
  
  
//在另一个函数内部定义的函数会将包含函数（即外部函数）的活动对象添加到它的作用域链中  
//因此在campareFunction()函数内部定义的匿名函数的作用域链中，实际上会包含外部函数campareFunction()的活动对象  
  
**var** compare = campareFunction("name");  
**var** result = compare({name:"Victor"},{name:"Hugo"});  
write(result); //1  
  
  
/\*  
campareFunction执行环境  
 （scope chain） ----> Scope Chain  
 1 -----------> Global variable object  
 0 --- campareFunction ->[campareFunction执行环境]  
 | result  
 |  
 |  
 --------> campareFunction() activation object  
 arguments  
 propertyName  
  
 annoymous（匿名函数）执行环境  
（scope chain） ---------> Scope Chain  
 2 ------------> Global variable object(和上面一样)  
 1 ------------> campareFunction() activation object  
 0 ------------> Closure activation object  
 arguments  
 obj1  
 obj2

\*/

//在匿名函数从campareFunction()函数中被返回后，它的作用域链初始化为包含campareFunction活动对象和全局变量对象  
//匿名函数就可以访问在campareFunction()函数中定义的所有变量  
//并且campareFunction()函数在执行完毕后活动对象也不会被销毁，  
//因为返回的是匿名函数，匿名函数的作用域链仍然在引用这个(campareFunction()函数的)活动对象  
//campareFunction返回后，campareFunction执行环境中的作用域链被销毁了，但是它的活动对象仍然会留在内存中,  
//直到匿名函数被销毁，campareFunction的活动对象才会被销毁  
  
  
//创建函数  
**var** compareNames = campareFunction("name");  
//调用函数  
**var** result = compareNames({name:"Victor"},{name:"Hugo"});  
  
write(result); //1  
  
//解除对匿名函数的引用（以便释放内存）  
compareNames = **null**;//通知垃圾回收例程将其清除，随着匿名函数的作用域链被销毁，其他作用域链（除了全局作用域）也都可以  
//安全的被销毁  
  
//由于闭包会携带包含它的函数的作用域  
//会比其他函数占用更多的内存  
//过度使用闭包会导致内存占用过多  
//在绝对必要时考虑使用闭包

/\*闭包与变量\*/  
**function** creatFunction(){  
 **var** result = **new** Array();  
  
 **for**(**var** i=0; i<10; i++){  
 result[i] = **function**(){  
 **return** i; //注意i是外部函数的活动对象的属性，而不是匿名函数对象的属性  
 };  
 }  
 **return** result; //返回的是一个函数数组，这个数组里的元素都是函数  
}  
  
**var** result = [];  
result = creatFunction();  
  
write(result[0]()); //10  
  
**for**(**var** i=0; i<10; i++){  
 write(result[i]()); //每一个都是10  
}  
  
//闭包只能取得包含函数中任何变量的最后一个值  
//闭包保存的是整个变量对象，而不是某个特殊的变量  
//每个函数都返回10  
//因为每个函数的作用域链中都保存着creatFunction()函数的活动对象  
//所以它们引用的都是同一个变量i  
//当creatFunction函数返回后，变量i的值都是10  
  
  
//！！！！！！！重要理解  
//此时每个函数（result数组里的函数）都引用着保存变量i的同一个变量对象，所以每个函数内部的i的值都是10  
  
//展开上面这句压缩的神解释！  
//也可以这样理解 result[i]数组里存放的是函数指针，指向匿名函数function(){return i};  
//所以creatFunction函数返回的是指针，指向内部某个匿名的函数  
  
//creatFunction()函数在执行完毕后它的活动对象不会被销毁，因为匿名函数的作用域链仍然在引用这个活动对象  
//creatFunction()函数在执行完毕前的最后一次需要返回的匿名函数为function(){return 10}; 最后一次的i的值为10  
//creatFunction()函数在返回后，creatFunction执行环境被销毁，它的活动对象仍然保留在内存中，此时活动对象中的i的值为10  
//活动对象中保存了最后一次i的值10，这个i的值是活动对象的属性  
//所以每个数组函数变量的指针指向的都是function(){return 10};  
//所以返回的都是10  
  
//总结一下就是返回外部函数的时候，因为返回的是内部的匿名函数，根据匿名函数的作用域链包含着全局对象和包含它的外部函数的活动对象  
//所以匿名函数的作用域链仍然在引用这个外部函数的活动对象，这个外部函数的活动对象在外部函数执行完毕后仍然不会销毁  
//但是匿名函数指针只能指向包含外部函数最后一次执行情况的对应的活动对象里的属性值的匿名函数  
  
//闭包保存的是整个外部函数的活动对象，而不是某个变量值，这个活动对象包括arguments,函数参数以及函数内的局部变量等  
  
  
//理解闭包，以下这种不是闭包

//不是闭包  
**function** creatFunction(){  
 **var** result = **new** Array();  
  
 **for**(**var** i=0; i<10; i++){  
 result[i] = **function**(){  
 **return** i;  
 }();  
 }  
  
 **return** result;  
 //返回的是一个数组，数组里的是基本数据类型  
 //这种就不是闭包了，调用完这个函数后活动对象和执行环境都会被释放掉  
 //因为没有一个外部对象在引用内部的匿名函数，所谓的闭包就是返回内部函数，因为内部函数的作用域链中包含着外部函数的作用域  
 //并且这个内部函数包含外部函数的活动对象的属性  
}  
  
write(creatFunction()); // 0,1,2,3,4,5,6,7,8,9

//闭包问题解决方法  
**function** creatFunction(){  
 **var** result = **new** Array();  
  
 **for**(**var** i=0; i<10; i++){  
 result[i] = **function**(num){ //这个不是闭包了，这个是闭包的外部函数  
 **return function**(){ //返回的是这个内部函数，这个是闭包，这个闭包可以访问num  
 **return** num;  
 };  
 }(i); //当前强制执行，这个和带闭包问题的外部函数有本质区别，那个只是赋值指针，然后执行完外部函数后再执行指向内部函数的指针变量，这样一来指针变量指向的是同一个内部函数

//而这里是最大的外部函数还没调用完毕，就强制执行了十次num闭包对应的外部匿名函数，所以返回的指针函数都不是同一个，而是返回了十个不同的函数指针变量，而每次都是保存外面这个匿名函数对应的活动对象的属性值num  
 }  
 **return** result;  
}  
  
  
**var** result = [];  
result = creatFunction();  
write(result[0]()); //0  
write(result[1]()); //1

/\*闭包中的this对象\*/  
//匿名函数的执行环境具有全局性，因此this对象通常指向window  
**var** f = **function**(){  
 **return function**(){  
 write(**this**);  
 }();  
}  
  
f();//[object Window]  
  
  
**var** name = "The Window";  
  
**var** object = {  
 name: "The Object",  
  
 getNameFun: **function**(){  
 **return function**(){  
 **return this**.name;  
 }  
 }  
};  
  
  
write(object.getNameFun()()); //The Window  
  
//为什么匿名函数没有取得其包含作用域（或外部作用域）的this对象呢？  
//每个函数在被调用时都会自动取得两个特殊变量：this和arguments  
//因为这个是函数的内部属性，所以内部函数在搜索这两个变量时，只会搜索到其活动对象为止，  
//详细见P87的示例图，每个活动对象都有自己的arguments和this属性  
//因此永远不可能直接访问外部函数中的这两个变量  
//有因为匿名函数中的this对象通常指代window对象，所以返回的是The Window  
  
//补救方法  
**var** age = 13;  
**var** obj = {  
 age:14,  
 getAgeFun:**function**(){  
 **var** that = **this**; //调试结果：that = Object {age: 14} this指代的是上下环境中的对象本身  
 **return function**(){  
 **return** that.age;  
 };  
 }  
};  
  
write(obj.getAgeFun()()); //14  
  
  
//this和arguments都存在同样的问题，如果想访问作用域中的arguments对象，  
//必须将对该对象的引用保存到另一个闭包能够访问的变量中

**var** name = "The Window";  
  
**var** object = {  
 name: "The Object",  
  
 getName: **function**(){  
 **return this**.name;  
 }  
};  
  
write(object.getName()); //The Object  
write((object.getName)()); //The Object  
write((object.getName = object.getName)()); //The Window 赋值表达式是函数本身，this的值不能得到维持

/\*内存泄漏\*/ //高级编程P183 遗留问题

6.4 模仿块级作用域（不是特别明朗某些地方）

**function** outputNumber(){  
 **for**(**var** i=0; i<10;i++){  
 write(i); //1,2,3,4,5,6,7,8,9  
 }  
  
 **var** i; //10 对后续的声明视而不见  
 write(i); //10  
}  
  
outputNumber(); //i是被添加到outputNumber()的活动对象中的  
  
//匿名函数可以用来模仿块级作用域并避免以上问题  
//用作块级作用域的（通常称为私有作用域）的匿名函数的语法如下  
(**function**(){  
 //这里是块级作用域  
})();  
  
/\*  
function (){  
  
}(); //Uncaught SyntaxError: Unexpected token (  
\*/  
  
//JS将function关键字当做一个函数声明的开始，函数声明后面不能跟圆括号  
//函数表达式的后面可以跟圆括号  
//要将函数声明转换成函数表达式只要给它加上一对圆括号即可  
(**function**(){  
 //这里是块级作用域  
})();

//如果在某些地方只是临时需要一些变量，就可以私有作用域  
**function** outputNumbers(count){  
 (**function**(){  
 //块级作用域  
 **for**(**var** i=0;i<count;i++){  
 write(i);  
 }  
 })();  
  
 write(i); //Uncaught ReferenceError: i is not defined  
}  
  
outputNumbers(10);  
  
//在这个函数中，在for循环外部插入了一个私有作用域  
//在匿名函数中定义的任何变量，都会在执行结束时被销毁  
//所以匿名函数下的i被报错  
  
//这种技术经常在全局作用域中被用在函数外部，从而限制向全局作用域中添加过多的变量和函数  
//通过创建私有作用域，每个开发人员既可以使用自己的变量，又不必担心捣乱全局作用域  
  
  
(**function**(){  
 //都变成了局部变量  
 //调用完即销毁变量  
   
 **var** now = **new** Date();  
 **if**(now.getMonth() == 0 && now.getDate() == 1){  
 write("Happy new year");  
 }  
})();  
//now是局部变量，不必在全局作用域中创建它

/\*私有变量\*/  
//在任何函数中定义的变量都可以认为是私有变量  
//私有变量包括函数的参数、局部变量和函数内部定义的其他函数  
  
//如果在函数内部创建闭包，那么闭包通过自己的作用域链可以访问私有变量（函数外部不能访问它们）  
//这样就可以创建用于访问私有变量的共有方法了  
//  
  
**function** MyObject(){  
 //私有变量  
 **var** privateVar = 1;  
 //私有函数  
 **var** privateFun = **function**(){  
 **return false**;  
 }  
  
 //特权方法  
 **this**.publicFun = **function**(){  
 privateVar ++;  
 **return** privateFun();  
 }  
}  
  
  
//特权方法作为闭包有权访问在构造函数中定义的所有变量和函数  
//并且想想闭包的特性，外部活动对象的内存在引用完之前是不会被释放的！  
  
//除了使用publicFun()这一个途径外，没有任何办法可以直接访问privateVar和privateFun  
  
//利用私有和特权成员，可以隐藏那些不应该被直接修改的数据  
**function** Person(name){ //构造函数  
 //特权方法  
 **this**.getName = **function**(){  
 **return** name;  
 };  
 //特权方法  
 **this**.setName = **function**(value){  
 name = value;  
 };  
}  
  
  
**var** Per = **new** Person("Victor");  
write(Per.getName()); //Victor 之所以name还保存着Victor，是因为闭包的特性导致活动对象的name属性不会被释放  
Per.setName("Hugo");  
write(Per.getName()); //Hugo  
  
  
//getName和setName方法可以在构造函数的外部使用  
//而且都有权访问私有变量name  
//这两个方法是在构造函数内部定义的，他们作为闭包能够通过作用域链访问name  
//缺点，方法使用了构造函数，针对每个实例都会创建同样的一组新方法，需要使用原型对象来共享方法  
  
/\*静态私有变量\*/  
(**function**(){  
 **var** name = "";  
  
 //定义构造函数时并没有使用函数声明，而是使用了函数表达式  
 //函数声明创建的是局部函数，这不是我们想要的  
 //这里需要的是全局函数  
 //函数执行完后构造函数不会被释放  
 //初始化未经声明的变量（没有使用var关键字声明）总是会创建一个全局变量  
 Person = **function**(value){ //构造函数，没有使用var说明是全局函数，能够在私有作用域外被访问，在严格模式中会报错  
 name = value;  
 };  
  
 Person.prototype.getName = **function**(){ //私有变量和函数是由实例共享的，特权方法是在原型上定义的，所有实例都是使用同一个函数  
 **return** name; //特权方法作为一个闭包，总是保存着对包含作用域的引用  
 };  
  
 Person.prototype.setName = **function**(value){  
 name = value;  
 };  
}());  
  
//这个例子中Person构造函数与getName和setName方法一样都有权访问私有变量name  
//在这种模式下，name就变成了一个静态的、由所有实例共享的属性  
//这种方式创建的私有变量会因为使用原型而增进了代码复用，但每个实例都没有自己的私有变量  
//多查找作用域链中的一个层次，就会在一定程度上影响查找速度，这是使用闭包和私有变量的不足之处  
  
**var** per = **new** Person("Victor");  
write(per.getName()); //Victor  
**var** per2 = **new** Person("Hugo");  
write(per.getName()); //Hugo  
write(per2.getName()); //Hugo 所有实例对象共享了name属性

/\*模块模式\*/  
  
//模块模式是为单例创建私有变量和特权方法  
//单例值得就是只有一个实例对象  
//按照惯例，JS是以对象字面量的方式来创建单例对象的  
  
//例如  
**var** singleObject = {  
 name:"Single",  
 method:**function**(){  
 //这里是方法的代码  
 }  
}  
  
//模块模式通过为单例添加私有变量和特权方法能够使其得到增强  
**var** single = **function**(){ //返回的是一个对象的匿名函数  
 //私有变量和私有属性  
 **var** privateVar = 10;  
  
 **function** privateFun(){  
 **return false**;  
 }  
  
 //特权、公有方法和属性  
 **return** {  
 publicVar: **true**,  
 publicFun: **function**(){  
 privateVar ++;  
 **return** privateFun();  
 }  
 };  
}();

第七章 BOM

7.1 window对象