引用类型

# Object类型

## 对象的创建方式

|  |
| --- |
| 使用new 操作符：  var person = new Object();  person.name = "Nicholas";  person.age = 29; |
| 使用对象字面量表示法：  var person = {  name : "Nicholas",  age : 29  };//属性名一般使用“age”  //一般我们是使用对象字面量表示法。 |

## 访问对象的属性方式

|  |
| --- |
| 方括号法：alert(person["name"]); //"Nicholas"  点表示法：alert(person.name); //"Nicholas"  //当属性字符中包含非法字符时候，一般使用方括号表示法。 |

# Array类型

ECMAScript 数组的每一项可以保存任何类型的数据。而且，ECMAScript 数组的大小是可以动态调整的，即可以随着数据的添加自动增长以容纳新增数据。数组创建的两种方式：构造器和数组字面量表示法。

|  |
| --- |
| 第一种方式：使用Array 构造函数  var colors = new Array(3); // 创建一个包含3 项的数组  var names = new Array("Greg"); // 创建一个包含1 项，即字符串"Greg"的数组  //另外，在使用Array 构造函数时也可以省略new 操作符。  var colors = Array(3); |
| 第二种方式：数组字面量表示法。  var colors = ["red", "blue", "green"]; // 创建一个包含3 个字符串的数组  var names = []; // 创建一个空数组  var values = [1,2,]; // 不要这样！这样会创建一个包含2 或3 项的数组  var options = [,,,,,]; // 不要这样！这样会创建一个包含5 或6 项的数组 |

数组的特点：

* alert(colors[0]); // 显示第一项,如果超出数组的长度，返回undefined
* colors[2] = "black"; // 修改第三项，访问的长度在length范围内
* colors[5] = "brown"; // 新增第四五六项，超出数组length的时候会新增新项。中间未声明的变量为undefined。
* 可以通过colors.length 来访问数组的长度。也可以使用colors.length=2来改变数组的长度。当长度大于colors.length时候，超出的部分为undefined；长度小于数组长度时候，数组的有效元素为改变后的个数。

|  |
| --- |
| var colors = ["red", "blue", "green"]; // 创建一个包含3 个字符串的数组  colors.length = 4;  alert(colors[3]); //undefined |

## 检测数组

|  |
| --- |
| 方式一：  if (value instanceof Array){  //对数组执行某些操作  }//在多全局执行环境的时候会出现问题  方式二：  if (Array.isArray(value)){  }//支持Array.isArray()方法的浏览器有IE9+、Firefox 4+、Safari 5+、Opera 10.5+和Chrome。 |

## 转换方法

|  |
| --- |
| var colors = ["red", "blue", "green"]; // 创建一个包含3 个字符串的数组  alert(colors.toString()); // red,blue,green  alert(colors.valueOf()); // red,blue,green（返回数组，然后每一项调用toString方法。  alert(colors); // red,blue,green（alert方法接收的是字符串，所以colors会自动调用toString方法。  alert(colors.toLocaleString());//  alert(colors.join("||")); //red||green||blue（显示所有数组项字符以join函数接收的分隔符断开。） |

## 栈方法（先进后出）

|  |
| --- |
| push()方法：可以接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度。例如：  var colors = new Array(); // 创建一个数组  var count = colors.push("red", "green"); // 推入两项  alert(count); //2 |
| pop()方法：从数组末尾移除最后一项，减少数组的length 值，然后返回移除的项。例如：  var item = colors.pop(); // 取得最后一项  alert(item); //" green "  alert(colors.length); //1 |

## 队列方法（先进先出）

|  |
| --- |
| push()方法：可以接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度 |
| shift()方法：移除数组中的第一个项并返回该项，同时将数组长度减1。  例如：  var item = colors.shift(); //取得第一项  alert(item); //"red"  alert(colors.length); //长度0 |

## 重排列方法

|  |
| --- |
| reverse()方法反转数组项的顺序：  var values = [1, 2, 3, 4, 5];  values.reverse();  alert(values); //5,4,3,2,1 |
| sort()方法：按升序排列数组项——即最小的值位于最前面，最大的值排在最后面。为了实现排序，sort()方法会调用每个数组项的toString()转型方法，然后比较得到的字符串，以确定如何排序。  var values = [0, 1, 5, 10, 15];  values.sort();  alert(values); //0,1,10,15,5  可以通过比较函数产生降序排序的结果，只要交换比较函数返回的值即可。  function compare(value1, value2) {  if (value1 < value2) {  return 1;  } else if (value1 > value2) {  return -1;  } else {  return 0;  }  }  var values = [0, 1, 5, 10, 15];  values.sort(compare);  alert(values); // 15,10,5,1,0 |

## 操作方法

|  |
| --- |
| concat方法：如果传递给concat()方法的是一或多个数组，则该方法会将这些数组中的每一项都添加到结果数组中。如果传递的值不是数组，这些值就会被简单地添加到结果数组的末尾。例如：  var colors = ["red", "green", "blue"];  var colors2 = colors.concat("yellow", ["black", "brown"]);  alert(colors); //red,green,blue  alert(colors2); //red,green,blue,yellow,black,brown |
| slice方法：可以接受一或两个参数，即要返回项的起始和结束位置。在只有一个参数的情况下，slice()方法返回从该参数指定位置开始到当前数组末尾的所有项。如果有两个参数，该方法返回起始和结束位置之间的项，但不包括结束位置的项。例如：  var colors = ["red", "green", "blue", "yellow", "purple"];  var colors2 = colors.slice(1);  var colors3 = colors.slice(1,4);  alert(colors2); //green,blue,yellow,purple  alert(colors3); //green,blue,yellow  如果slice()方法的参数中有一个负数，则用数组长度加上该数来确定相应的位  置。例如，在一个包含5 项的数组上调用slice(-2,-1)与调用slice(3,4)得到的  结果相同。如果结束位置小于起始位置，则返回空数组。 |
| splice(起始位置,0（要删除的项数）,要插入的项（可以多个)：  var colors = ["red", "green", "blue"];  var removed = colors.splice(0,1); // 删除第一项  alert(colors); // green,blue  alert(removed); // red，返回的数组中只包含一项  removed = colors.splice(1, 0, "yellow", "orange"); // 从位置1 开始插入两项  alert(colors); // green,yellow,orange,blue  alert(removed); // 返回的是一个空数组  removed = colors.splice(1, 1, "red", "purple"); // 插入两项，删除一项  alert(colors); // green,red,purple,orange,blue  alert(removed); // yellow，返回的数组中只包含一项 |

## 位置方法

|  |
| --- |
| indexOf()和lastIndexOf()。这两个方法都接收两个参数：要查找的项和（可选的）表示查找起点位置的索引。参数只有一个时候，返回指定索引（0开始）位置的值；参数有两个的时候，返回索引匹配索引的值（全等===），没有找到返回-1）。例如：  var numbers = [1,2,3,4,5,4,3,2,1];  alert(numbers.indexOf(4)); //3  alert(numbers.lastIndexOf(4)); //5  alert(numbers.indexOf(4, 4)); //5  alert(numbers.lastIndexOf(4, 4)); //3  var person = { name: "Nicholas" };  var people = [{ name: "Nicholas" }];  var morePeople = [person];  alert(people.indexOf(person)); //-1  alert(morePeople.indexOf(person)); //0 |

## 迭代方法（一共有5个都接受3个参数：item，index,array）

|  |
| --- |
| //item每一项的值，index当前索引，array数组本身  var numbers = [1,2,3,4,5,4,3,2,1];  •every()：对数组中的每一项运行给定函数，如果该函数对每一项都返回true，则返回true。  例如：  var everyResult = numbers.every(function(item, index, array){  return (item > 2);  });  alert(everyResult); //false  •filter()：对数组中的每一项运行给定函数，返回该函数会返回true 的项组成的数组。  例如：  var filterResult = numbers.filter(function(item, index, array){  return (item > 2);  });  alert(filterResult); //[3,4,5,4,3]  • forEach()：对数组中的每一项运行给定函数。这个方法没有返回值。  • map()：对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。例如：  var numbers = [1,2,3,4,5,4,3,2,1];  var mapResult = numbers.map(function(item, index, array){  return item \* 2;  });  alert(mapResult); //[2,4,6,8,10,8,6,4,2]  some()：对数组中的每一项运行给定函数，如果该函数对任一项返回true，则返回true。例如：  var someResult = numbers.some(function(item, index, array){  return (item > 2);  });  alert(someResult); //true |

## 归并方法

|  |
| --- |
| reduce()和reduceRight()接收四个参数：前一个值、当前值、项的索引和数组对象。中，reduce()方法从数组的第一项开始，逐个遍历到最后。reduceRight()则从数组的最后一项开始，向前遍历到第一项。  例子：var values = [1,2,3,4,5];  var sum = values.reduce(function(prev, cur, index, array){  return prev + cur;  });  alert(sum); //15  ArrayReductionExample01.htm  第一次执行回调函数，prev 是1，cur 是2。第二次，prev 是3（1 加2 的结果），cur 是3（数组的第三项）。这个过程会持续到把数组中的每一项都访问一遍，最后返回结果。 |

# Date类型

var now = new Date();//在调用Date 构造函数而不传递参创建的对象自动获得当前日期和时间。如果想根据特定的日期和时间创建日期对象，必须传入表示该日期的毫秒数。ECMAScript 提供了两个方法：Date.parse()和Date.UTC()，他们都是返回毫秒数。

|  |
| --- |
| 方法1. Date.parse()接收一个个表示日期的字符串参数：   * “月/日/年”，如6/13/2004； * “英文月名 日,年”，如January 12,2004； * “英文星期几 英文月名 日 年 时:分:秒 时区”，如Tue May 25 2004 00:00:00 GMT-0700。 * ISO 8601 扩展格式YYYY-MM-DDTHH:mm:ss.sssZ（例如2004-05-25T00:00:00）。只有兼容ECMAScript 5 的实现支持这种格式。   注意：如果传入Date.parse()方法的字符串不能表示日期，那么它会返回NaN。实际上，如果直接将表示日期的字符串传递给Date 构造函数，也会在后台调用Date.parse()。  var someDate = new Date("May 25, 2004");等价于var someDate = new Date(Date.parse("May 25, 2004"));  日期对象及其在不同浏览器中的实现有许多奇怪的行为。其中有一种倾向是将超出范围的值替换成当前的值，以便生成输出。例如，在解析"January 32, 2007"时，有的浏览器会将其解释为"February 1, 2007"。而Opera 则倾向于插入当前月份的当前日期，返回"January 当前日期，2007"。也就是说，如果在2007 年9 月21 日运行前面的代码，将会得到"January 21, 2007"（都是21 日）。 |
| 方法2. Date.UTC()的参数分别是年份、基于0 的月份（一月是0，二月是1，以此类推）、月中的哪一天（1 到31）、小时数（0 到23）、分钟、秒以及毫秒数。在这些参数中，只有前两个参数（年和月）是必需的。如果没有提供月中的天数，则假设天数为1；如果省略其他参数，则统统假设为0。  // GMT 时间2000 年1 月1 日午夜零时  var y2k = new Date(Date.UTC(2000, 0));  // GMT 时间2005 年5 月5 日下午5:55:55  var allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));  注意：Date 构造函数也会模仿Date.UTC()，但有一点明显不同：日期和时间都基于本地时区而非GMT 来创建。不过，Date 构造函数接收的参数仍然Date.UTC()相同。  // 本地时间2000 年1 月1 日午夜零时  var y2k = new Date(2000, 0);  // 本地时间2005 年5 月5 日下午5:55:55  var allFives = new Date(2005, 4, 5, 17, 55, 55); |
| 附加方法：ECMAScript 5 添加了Data.now()方法，返回表示调用这个方法时的日期和时间的毫秒数。例如：  //取得开始时间  var start = Date.now();  注意：支持Data.now()方法的浏览器包括IE9+、Firefox 3+、Safari 3+、Opera 10.5 和Chrome。在不支持它的浏览器中，使用+操作符把Data 对象转换成字符串，也可以达到同样的目的。  //取得开始时间  var start = +new Date(); |

## 继承方法

### toLocaleString()方法

Date 类型的toLocaleString()方法会按照与浏览器设置的地区相适应的格式返回日期和时间。根据浏览器不同而不同。

toLocaleString() — Thursday, February 01, 2007 12:00:00 AM

### toString()方法

toString()方法则通常返回带有时区信息的日期和时间，其中时间一般以军用时间（即小时的范围是0 到23）表示。根据浏览器不同而不同。

toString() — Thu Feb 1 00:00:00 PST 2007

### valueOf()方法

Date 类型的valueOf()方法，则根本不返回字符串，而是返回日期的毫秒表示。因此，可以方便使用比较操作符（小于或大于）来比较日期值。请看下面的例子：

|  |
| --- |
| var date1 = new Date(2007, 0, 1); //"January 1, 2007"  var date2 = new Date(2007, 1, 1); //"February 1, 2007"  alert(date1 < date2); //true  alert(date1 > date2); //false |

## 日期格式化方法

Date 类型还有一些专门用于将日期格式化为字符串的方法，这些方法如下。

* toDateString()——以特定于实现的格式显示星期几、月、日和年；
* toTimeString()——以特定于实现的格式显示时、分、秒和时区；
* toLocaleDateString()——以特定于地区的格式显示星期几、月、日和年；
* toLocaleTimeString()——以特定于实现的格式显示时、分、秒；
* toUTCString()——以特定于实现的格式完整的UTC 日期。

## 日期/时间组件方法

* getTime() 返回表示日期的毫秒数；与valueOf()方法返回的值相同
* setTime(毫秒) 以毫秒数设置日期，会改变整个日期
* setFullYear(年) 设置日期的年份。传入的年份值必须是4位数字（如2007）
* getFullYear() 取得4位数的年份（如2007而非仅07）
* getUTCFullYear() 返回UTC日期的4位数年份
* getMilliseconds() 返回日期中的毫秒数
* getUTCMilliseconds() 返回UTC日期中的毫秒数
* setMilliseconds(毫秒) 设置日期中的毫秒数

# RegExp类型

|  |
| --- |
| 正则表达式的创建方式：  方式一：字面量形式  var expression = / pattern / flags ;  //flag的种类   * g：表示全局（global）模式，即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止； * i：表示不区分大小写（case-insensitive）模式，即在确定匹配项时忽略模式与字符串的大小写； * m：表示多行（multiline）模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项。   //匹配字符串中所有"at"的实例  var pattern1 = /at/g; |
| 方式二：使用构造函数创建（注意要使用双重转义）  var pattern2 = new RegExp("[bc]at", "i");  注意：在ECMAScript 3 中，正则表达式字面量始终会共享同一个RegExp 实例，而使用构造函数创建的每一个新RegExp 实例都是一个新实例。 |

## RegExp实例属性

* 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，按照字面量形式而非传入构造函数中的字符串模式返回。

|  |
| --- |
| var pattern1 = /\[bc\]at/i;  alert(pattern1.global); //false  alert(pattern1.ignoreCase); //true  alert(pattern1.multiline); //false  alert(pattern1.lastIndex); //0  alert(pattern1.source); //"\[bc\]at"  var pattern2 = new RegExp("\\[bc\\]at", "i");  alert(pattern2.global); //false  alert(pattern2.ignoreCase); //true  alert(pattern2.multiline); //false  alert(pattern2.lastIndex); //0  alert(pattern2.source); //"\[bc\]at" |

## RegExp实例方法

### exec()方法

exec()接受一个参数：应用模式的字符串。然后返回包含第一个匹配项信息的数组；或者在没有匹配项的情况下返回null。返回的数组虽然是Array 的实例，但包含两个额外的属性：index 和input。其中，index 表示匹配项在字符串中的位置，而input 表示应用正则表达式的字符串。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。

|  |
| --- |
| var text = "mom and dad and baby";  var pattern = /mom( and dad( and baby)?)?/gi;  var matches = pattern.exec(text);  alert(matches.index); // 0  alert(matches.input); // "mom and dad and baby"  alert(matches[0]); // "mom and dad and baby"  alert(matches[1]); // " and dad and baby"  aler t(matches[2]); // " and baby"（组匹配()）  注意：如果正则表达式不是是全局模式，每次调用exec()返回的都是第一个匹配项。而如果是全局模式，因此每次调用exec()都会返回字符串中的下一个匹配项，直至搜索到字符串末尾为止。 |

### test方法

它接受一个字符串参数。在模式与该参数匹配的情况下返回true；否则，返回false。

|  |
| --- |
| var text = "000-00-0000";  var pattern = /\d{3}-\d{2}-\d{4}/;  if (pattern.test(text)){  alert("The pattern was matched.");  } |

### toLocaleString()和toString()方法

RegExp 实例继承的toLocaleString()和toString()方法都会返回正则表达式的字面量，与创建正则表达式的方式无关。例如：

|  |
| --- |
| var pattern = new RegExp("\\[bc\\]at", "gi");  alert(pattern.toString()); // /\[bc\]at/gi  alert(pattern.toLocaleString()); // /\[bc\]at/gi |

## RegExp构造函数属性

|  |
| --- |
| （长属性名 短属性名 说明）  input $\_ 最近一次要匹配的字符串。Opera未实现此属性  lastMatch $& 最近一次的匹配项。Opera未实现此属性  lastParen $+ 最近一次匹配的捕获组。Opera未实现此属性  leftContext $` Input字符串中lastMatch之前的文本  multiline $\* 布尔值，表示是否所有表达式都使用多行模式。IE和Opera未实现此属性  rightContext $' Input字符串中lastMatch之后的文本  另外，有多达9 个用于存储捕获组的构造函数属性：RegExp.$1、RegExp.$2…RegExp.$9。 |

4. 模式的局限性

匹配字符串开始和结尾的\A 和\Z 锚①

向后查找（lookbehind）②

并集和交集类

原子组（atomic grouping）

Unicode 支持（单个字符除外，如\uFFFF）

命名的捕获组③

s（single，单行）和x（free-spacing，无间隔）匹配模式

条件匹配

正则表达式注释

# Function类型

|  |
| --- |
| 方式一：函数声明：  function sum (num1, num2) {  return num1 + num2;  }  方式二：函数表达式定义函数：  var sum = function(num1, num2){  return num1 + num2;  };  方式三：使用Function 构造函数定义函数：  var sum = new Function("num1", "num2", "return num1 + num2"); // 不推荐。因为这种语法会导致解析两次代码（第一次是解析常规ECMAScript 代码，第二次是解析传入构造函数中的字符串），从而影响性能。不过，这种语法对于理解“函数是对象，函数名是指针”的概念倒是非常直观的。。 |

## 没有重载

将函数名想象为指针，也有助于理解为什么ECMAScript 中没有函数重载的概念。后面创建的函数会覆盖前面创建的函数。（地址被覆盖）

## 函数创建方式

可以通过函数声明的方式来创建函数：代码开始执行之前，解析器就已经通过一个名为函数声明提升（function declaration hoisting）的过程，读取并将函数声明添加到执行环境中。

函数表达式创建函数：在创建之前不能使用函数。但是函数声明的方式创建函数，它可以在创建之前就使用函数。

## 函数的调用（传值的过程）

因为ECMAScript 中的函数名本身就是变量，所以函数也可以作为值来使用。也就是说，不仅可以像传递参数一样把一个函数传递给另一个函数，而且可以将一个函数作为另一个函数的结果返回。

|  |
| --- |
| function callSomeFunction(someFunction, someArgument){  return someFunction(someArgument);  }  function add10(num){  return num + 10;  }  var result1 = callSomeFunction(add10, 10);  alert(result1); //20  function getGreeting(name){  return "Hello, " + name;  }  var result2 = callSomeFunction(getGreeting, "Nicholas");  alert(result2); //"Hello, Nicholas" |

## 函数的内部属性

### arguments属性

它是一个类数组对象，包含着传入函数中的所有参数。虽然arguments 的主要用途是保存函数参数。但这个对象还有一个名叫callee 的属性，该属性是一个指针，指向拥有这个arguments 对象的函数。使用arguments.callee()来调用本身。

|  |
| --- |
| function factorial(num){  if (num <=1) {  return 1;  } else {  return num \* arguments.callee(num-1)  }  } |

### this属性

this引用的是函数据以执行的环境对象——或者也可以说是this 值（当在网页的全局作用域中调用函数时，this 对象引用的就是window）

### caller 属性

ECMAScript 5 也规范化了另一个函数对象的属性：caller。这个属性中保存着调用当前函数的函数的引用，如果是在全局作用域中调用当前函数，它的值为null。

注意：IE、Firefox、Chrome 和Safari 的所有版本以及Opera 9.6 都支持caller 属性。当函数在严格模式下运行时，访问arguments.callee 会导致错误。ECMAScript 5 还定义了arguments.caller 属性，但在严格模式下访问它也会导致错误，而在非严格模式下这个属性始终是undefined。定义这个属性是为了分清arguments.caller 和函数的caller 属性。以上变化都是为了加强这门语言的安全性，这样第三方代码就不能在相同的环境里窥视其他代码了。

严格模式还有一个限制：不能为函数的caller 属性赋值，否则会导致错误。

## 函数的属性和方法

### 属性

length ：属性表示函数希望接收的命名参数的个数（函数定义的形参个数）

prototype 属性：ECMAScript 中的引用类型而言，prototype 是保存它们所有实例方法的真正所在。

### 方法

|  |
| --- |
| 方法1. apply()，apply()方法接收两个参数：一个是在其中运行函数的作用域，另一个是参数数组。其中，第二个参数可以是Array 的实例，也可以是arguments 对象。例如：  function sum(num1, num2){  return num1 + num2;  }  function callSum1(num1, num2){  return sum.apply(this, arguments); // 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(10,10)); //20 |
| 方法二：call()  第一个参数是this 值没有变化，变化的是其余参数都直接传递给函数。  function sum(num1, num2){  return num1 + num2;  }  function callSum(num1, num2){  return sum.call(this, num1, num2);  }  alert(callSum(10,10)); //20 |

注意：事实上，传递参数并非apply()和call()真正的用武之地；它们真正强大的地方是能够扩充函数赖以运行的作用域。使用call()（或apply()）来扩充作用域的最大好处，就是对象不需要与方法有任何耦合关系。下面来看一个例子。

|  |
| --- |
| window.color = "red";  var o = { color: "blue" };  function sayColor(){  alert(this.color);  }  sayColor(); //red  sayColor.call(this); //red  sayColor.call(window); //red  sayColor.call(o); //blue |

# 基本包装类型

var s1 = "some text";

var s2 = s1.substring(2);// (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。

注意：引用类型与基本包装类型的主要区别就是对象的生存期。使用new 操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存在内存中。**而自动创建的基本包装类型的对象，则只存在于一行代码的执行瞬间，然后立即被销毁。**

## Boolean类型

Boolean 类型的实例重写了valueOf()方法，返回基本类型值true 或false；重写了toString()方法，返回字符串"true"和"false"。

var booleanObject = new Boolean(true);

alert(typeof booleanObject); //object

### 使用Boolean对象产生的误解：

|  |
| --- |
| var falseObject = new Boolean(false);  var result = falseObject && true;// falseObject对象默认会解析成true  alert(result); //true  建议：我们永远不要使用Boolean 对象。 |

## Number类型

var numberObject = new Number(10);

alert(typeof numberObject);//object(引用类型)

alert(numberObject instanceof Number); //true

与Boolean 类型一样，Number 类型也重写了valueOf()、toLocaleString()和toString()方法。重写后的valueOf()方法返回对象表示的基本类型的数值，另外两个方法则返回字符串形式的数值。可以为toString()方法传递一个表示基数的参数，告诉它返回几进制数值的字符串形式。

### toFixed()方法

可以表示带有0 到20 个小数位的数值。但这只是标准实现的范围，有些浏览器也可能支持更多位数。

var num = 10.005;

alert(num.toFixed(2)); //"10.01"

### toExponential()，

该方法返回以指数表示法（也称e 表示法）。表示的数值的字符串形式。

var num = 10;

alert(num.toExponential(1)); //"1.0e+1"

### toPrecision()方法,接收参数为几位有效数字

var num = 99;

alert(num.toPrecision(1)); //"1e+2"

alert(num.toPrecision(2)); //"99"

aler t(num.toPrecision(3)); //"99.0"

## String类型

var stringObject = new String("hello world");

String 对象的方法也可以在所有基本的字符串值中访问到。其中，继承的valueOf()、toLocaleString()和toString()方法，都返回对象所表示的基本字符串值。

### 字符方法

|  |
| --- |
| 方法一：charAt(基于0的位置)，返回字符  var stringValue = "hello world";  alert(stringValue.charAt(1)); //"e"  方法二：charCodeAt（基于0的位置），返回对应的字符编码。  var stringValue = "hello world";  alert(stringValue.charCodeAt(1)); //输出"101" |

### 字符串操作方法

|  |
| --- |
| 方法一: concat()方法可以接受任意多个参数，也就是说可以通过它  拼接任意多个字符串。一般使用+来代替。  var stringValue = "hello ";  var result = stringValue.concat("world", "!");  alert(result); //"hello world!" |
| 方法二：slice()、substr()和substring()。这三个方法都会返回被操作字符串的一个子字符串，而且也都接受一或两个参数。第一个参数指定子字符串的开始位置，第二个参数（在指定的情况下）表示子字符串到哪里结束。具体来说，slice()和substring()的第二个参数指定的是子字符串最后一个字符后面的位置。而substr()的第二个参数指定的则是返回的字符个数。如果没有给这些方法传递第二个参数，则将字符串的长度作为结束位置。在传递给这些方法的参数是负值的情况下，它们的行为就不尽相同了。其中，slice()方法会将传入的负值与字符串的长度相加，substr()方法将负的第一个参数加上字符串的长度，而将负的第二个参数转换为0。最后，substring()方法会把所有负值参数都转换为0。  var stringValue = "hello world";  alert(stringValue.slice(3)); //"lo world"  alert(stringValue.substring(3)); //"lo world"  alert(stringValue.substr(3)); //"lo world"  alert(stringValue.slice(3, 7)); //"lo w"  alert(stringValue.substring(3,7)); //"lo w"  alert(stringValue.substr(3, 7)); //"lo worl" |

### 字符串位置方法

indexOf()和lastIndexOf():。这两个方法都是从一个字符串中搜索给定的子字符串，然后返子字符串的位置（如果没有找到该子字符串，则返回-1）。这两个方法的区别在于：indexOf()方法从字符串的开头向后搜索子字符串，而lastIndexOf()方法是从字符串的末尾向前搜索子字符串。两个方法都可以接收可选的第二个参数，表示从字符串中的哪个位置开始搜索。

var stringValue = "hello world";

alert(stringValue.indexOf("o", 6)); //7

alert(stringValue.lastIndexOf("o", 6)); //4

### trim方法

删除前置及后缀的所有空格，然后返回结果。

var stringValue = " hello world ";

var trimmedStringValue = stringValue.trim();

alert(stringValue); //" hello world "

alert(trimmedStringValue); //"hello world"

### 字符串大小写转换方法

toLowerCase()、toLocaleLowerCase()、toUpperCase()和toLocaleUpperCase():toLowerCase()和toUpperCase()是两个经典的方法，借鉴自java.lang.String 中的同名方法。而toLocaleLowerCase()和toLocaleUpperCase()方法则是针对特定地区的实现。对有些地区来说，针对地区的方法与其通用方法得到的结果相同，但少数语言（如土耳其语）会为Unicode 大小写转换应用特殊的规则，这时候就必须使用针对地区的方法来保证实现正确的转换。

var stringValue = "hello world";

alert(stringValue.toLocaleUpperCase()); //"HELLO WORLD"

alert(stringValue.toUpperCase()); //"HELLO WORLD"

alert(stringValue.toLocaleLowerCase()); //"hello world"

alert(stringValue.toLowerCase()); //"hello world"

### 字符串的模式匹配方法

方法一：match()方法只接受一个参数，要么是一个正则表达式，要么是一个RegExp 对象。法返回了一个数组；如果是调用RegExp 对象的exec()方法并传递本例中的字符串作为参数，那么也会得到与此相同的数组：数组的第一项是与整个模式匹配的字符串，之后的每一项（如果有）保存着与正则表达式中的捕获组匹配的字符串。

var text = "cat, bat, sat, fat";

var pattern = /.at/;

//与pattern.exec(text)相同

var matches = text.match(pattern);

alert(matches.index); //0

alert(matches[0]); //"cat"

alert(pattern.lastIndex); //0

方法二：search（）方法的参数由字符串或RegExp 对象指定的一个正则表达式。search()方法返回字符串中第一个匹配项的索引；如果没有找到匹配项，则返回-1。而且，search()方法始终是从字符串开头向后查找模式。

var text = "cat, bat, sat, fat";

var pos = text.search(/at/);

alert(pos); //1

方法三：replace()方法。这个方法接受两个参数：第一个参数可以是一个RegExp 对象或者一个字符串（这个字符串不会被转换成正则表达式），第二个参数可以是一个字符串或者一个函数。如果第一个参数是字符串，那么只会替换第一个子字符串。要想替换所有子字符串，唯一的办法就是提供一个正则表达式，而且要指定全局（g）标志。

var text = "cat, bat, sat, fat";

var result = text.replace("at", "ond");

alert(result); //"cond, bat, sat, fat"

result = text.replace(/at/g, "ond");

aler t(result); //"cond, bond, sond, fond"

方法四：split()方法可以基于指定的分隔符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。分隔符可以是字符串，也可以是一个RegExp 对象（这个方法不会将字符串看成正则表达式）。split()方法可以接受可选的第二个参数，用于指定数组的大小，以便确保返回的数组不会超过既定大小。

var colorText = "red,blue,green,yellow";

var colors1 = colorText.split(","); //["red", "blue", "green", "yellow"]

var colors2 = colorText.split(",", 2); //["red", "blue"]

var colors3 = colorText.split(/[^\,]+/); //["", ",", ",", ",", ""]

### localeCompare()方法

这个方法比较两个字符串，并返回下列值中的一个：

* 如果字符串在字母表中应该排在字符串参数之前，则返回一个负数（大多数情况下是-1，具体的值要视实现而定）；
* 如果字符串等于字符串参数，则返回0；
* 如果字符串在字母表中应该排在字符串参数之后，则返回一个正数（大多数情况下是1，具体的值同样要视实现而定）。

下面是几个例子。

var stringValue = "yellow";

alert(stringValue.localeCompare("brick")); //1

alert(stringValue.localeCompare("yellow")); //0

alert(stringValue.localeCompare("zoo")); //-1

### fromCharCode()方法

是接收一或多个字符编码，然后将它们转换成一个字符串。从本质上来看，这个方法与实例方法charCodeAt()执行的是相反的操作。来看一个例子：

alert(String.fromCharCode(104, 101, 108, 108, 111)); //"hello"

# 单体内置对象

## Global对象

Global（全局）对象是不存在的. Global 对象在某种意义上是作为一个终极的“兜底儿对象”来定义的。换句话说，不属于任何其他对象的属性和方法，最终都是它的属性和方法。所有在全局作用域中定义的属性和函数，都是Global 对象的属性。本书前面介绍过的那些函数，诸如isNaN()、isFinite()、parseInt()以及parseFloat()，实际上全都是Global对象的方法。除Global 对象还包含其他一些方法。

### URI 编码方法

编码：Global 对象的encodeURI()和encodeURIComponent()方法可以对URI（Uniform Resource Identifiers，通用资源标识符）进行编码，以便发送给浏览器。有效的URI 中不能包含某些字符，例如空格。而这两个URI 编码方法就可以对URI 进行编码，它们用特殊的UTF-8 编码替换所有无效的字符，从而让浏览器能够接受和理解。 encodeURI()主要用于整个URI（例如，http://www.wrox.com/illegal value.htm），而encode-URIComponent()主要用于对URI 中的某一段（例如前面URI 中的illegal value.htm）进行编码。它们的主要区别在于，encodeURI()不会对本身属于URI 的特殊字符进行编码，例如冒号、正斜杠、问号和井字号；而encodeURIComponent()则会对它发现的任何非标准字符进行编码。来看下面的例子

|  |
| --- |
| var uri = "http://www.wrox.com/illegal value.htm#start";  //"http://www.wrox.com/illegal%20value.htm#start"  alert(encodeURI(uri));  //"http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start"  alert(encodeURIComponent(uri));  解码：encodeURI()和encodeURIComponent()方法对应的两个方法分别是decodeURI()和decodeURIComponent()。  var uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start";  //http%3A%2F%2Fwww.wrox.com%2Fillegal value.htm%23start  alert(decodeURI(uri));  //http://www.wrox.com/illegal value.htm#start  alert(decodeURIComponent(uri)); |

### eval()方法

现在，我们介绍最后一个——大概也是整个ECMAScript 语言中最强大的一个方法：eval()。eval()方法就像是一个完整的ECMAScript 解析器，它只接受一个参数，即要执行的ECMAScript （或JavaScript）字符串。看下面的例子：eval("alert('hi')");

通过eval()执行的代码被认为是包含该次调用的执行环境的一部分，因此被执行的代码具有与该执行环境相同的作用域链。

### Global 对象的属性

### window 对象

ECMAScript 虽然没有指出如何直接访问Global 对象，但Web 浏览器都是将这个全局对象作为window 对象的一部分加以实现的。因此，在全局作用域中声明的所有变量和函数，就都成为了window对象的属性。来看下面的例子。

var color = "red";

function sayColor(){

alert(window.color);

}

window.sayColor(); //"red"

## Math对象

### Math 对象的属性

Math.E 自然对数的底数，即常量e的值

Math.LN10 10的自然对数

Math.LN2 2的自然对数

Math.LOG2E 以2为底e的对数

Math.LOG10E 以10为底e的对数

Math.PI π的值

Math.SQRT1\_2 1/2的平方根（即2的平方根的倒数）

Math.SQRT2 2的平方根

### min()和max()方法

这两个方法都可以接收任意多个数值参数。

var max = Math.max(3, 54, 32, 16);

alert(max); //54

var min = Math.min(3, 54, 32, 16);

alert(min);

### 舍入方法

Math.ceil()执行向上舍入，即它总是将数值向上舍入为最接近的整数；

Math.floor()执行向下舍入，即它总是将数值向下舍入为最接近的整数；

Math.round()执行标准舍入，即它总是将数值四舍五入为最接近的整数

alert(Math.ceil(25.9)); //26

alert(Math.ceil(25.5)); //26

alert(Math.ceil(25.1)); //26

alert(Math.round(25.9)); //26

alert(Math.round(25.5)); //26

alert(Math.round(25.1)); //25

alert(Math.floor(25.9)); //25

alert(Math.floor(25.5)); //25

alert(Math.floor(25.1)); //25

### Math.random()方法:

返回大于等于0 小于1 的一个随机数。