String类型

String 类型是字符串的对象包装类型，可以像下面这样使用 String 构造函数来创建。

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

String 类型的每个实例都有一个 length 属性，表示字符串中包含多个字符。来看下面的例子。

```
var stringValue = "hello world";
alert(stringValue.length);      //"11"
```

这个例子输出了字符串"hello world"中的字符数量，即"11"。应该注意的是，即使字符串中包含双字节字符（不是占一个字节的 ASCII 字符），每个字符也仍然算一个字符。

1. 字符方法

charAt()　charCodeAt()　stringValue[]

两个用于访问字符串中特定字符的方法是：charAt()和 charCodeAt()。这两个方法都接收一个参数，即基于 0 的字符位置。其中，charAt()方法以单字符字符串的形式返回给定位置的那个字符（ECMAScript 中没有字符类型）。例如：

```
var stringValue = "hello world";
alert(stringValue.charAt(1));    //"e"
```

字符串"hello world"位置 1 处的字符是"e"，因此调用 charAt(1)就返回了"e"。如果你想得到的不是字符而是字符编码，那么就要像下面这样使用 charCodeAt()了。

```
var stringValue = "hello world";
alert(stringValue.charCodeAt(1));    //输出"101"
```

ECMAScript 5 还定义了另一个访问个别字符的方法。在支持此方法的浏览器中，可以使用方括号加数字索引来访问字符串中的特定字符，如下面的例子所示。

```
var stringValue = "hello world";
alert(stringValue[1]);    //"e"
```

2. 字符串操作方法

concat()　slice()　substr()　substring()

下面介绍与操作字符串有关的几个方法。第一个就是 concat()，用于将一或多个字符串拼接起来，返回拼接得到的新字符串。先来看一个例子。

```
var stringValue = "hello ";
var result = stringValue.concat("world");
alert(result);              //"hello world"
alert(stringValue);         //"hello"
```

ECMAScript还提供了三个基于子字符串创建新字符串的方法： 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。与concat() 方法一样， slice() 、 substr() 和 substring() 也不会修改字符串本身的值——它们只是
返回一个基本类型的字符串值，对原始字符串没有任何影响。请看下面的例子。

```
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"
```

slice(起始位置，结束位置）第二个参数为负标识从最后一位往前的位置

substring(起始位置，结束位置）第二个参数为负会把第二个参数转换为零

substr(起始位置，个数）第二个参数为负会把第二个参数转换为零

这个例子比较了以相同方式调用 slice()、substr()和 substring()得到的结果，而且多数情况下的结果是相同的。在只指定一个参数 3 的情况下，这三个方法都返回"lo world"，因为"hello"中的第二个"l"处于位置 3。而在指定两个参数 3 和 7 的情况下，slice()和substring()返回"lo w"（"world"中的"o"处于位置 7，因此结果中不包含"o"），但 substr()返回"lo worl"，因为它的第二个参数指定的是要返回的字符个数。

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。下面来看例子。

```
var stringValue = "hello world";
alert(stringValue.slice(-3));             //"rld"
alert(stringValue.substring(-3));         //"hello world"
alert(stringValue.substr(-3));            //"rld"
alert(stringValue.slice(3, -4));          //"lo w"
alert(stringValue.substring(3, -4));      //"hel"
alert(stringValue.substr(3, -4));         //"" (空字符串)
```

这个例子清晰地展示了上述三个方法之间的不同行为。在给 slice()和 substr()传递一个负值参数时，它们的行为相同。这是因为-3 会被转换为 8（字符串长度加参数 11+(−3)=8），实际上相当于调用了 slice(8)和 substr(8)。但 substring()方法则返回了全部字符串，因为它将-3 转换成了 0。

IE 的 JavaScript 实现在处理向 substr()方法传递负值的情况时存在问题，它会返回原始的字符串。IE9 修复了这个问题。

当第二个参数是负值时，这三个方法的行为各不相同。slice()方法会把第二个参数转换为 7，这就相当于调用了 slice(3,7)，因此返回"lo w"。substring()方法会把第二个参数转换为 0，使调用变成了 substring(3,0)，而由于这个方法会将较小的数作为开始位置，将较大的数作为结束位置，因此最终相当于调用了 substring(0,3)。substr()也会将第二个参数转换为 0，这也就意味着返回包含零个字符的字符串，也就是一个空字符串。

3. 字符串位置方法

indexOf() lastIndexOf()

有两个可以从字符串中查找子字符串的方法：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。还是来看一个例子吧。

```
var stringValue = "hello world";
alert(stringValue.indexOf("o"));              //4
alert(stringValue.lastIndexOf("o"));          //7
```

这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。换句话说，indexOf()会从该参数指定的位置向后搜索，忽略该位置之前的所有字符；而 lastIndexOf()则会从指定的位置向前搜索，忽略该位置之后的所有字符。看下面的例子。

```
var stringValue = "hello world";
alert(stringValue.indexOf("o", 6));           //7
alert(stringValue.lastIndexOf("o", 6));       //4
```

在使用第二个参数的情况下，可以通过循环调用 indexOf() 或 lastIndexOf() 来找到所有匹配的子字符串，如下面的例子所示：

遍历整个字符串中指定字符的位置并保存在数组中

```
var stringValue = "Lorem ipsum dolor sit amet, consectetur adipisicing elit";
var positions = new Array();
var pos = stringValue.indexOf("e");

while(pos > -1){
    positions.push(pos);
    pos = stringValue.indexOf("e", pos + 1);
}

alert(positions);    //"3,24,32,35,52"
```

StringTypeLocationMethodsExample02.htm

这个例子通过不断增加 indexOf()方法开始查找的位置，遍历了一个长字符串。在循环之外，首先找到了"e"在字符串中的初始位置；而进入循环后，则每次都给 indexOf()传递上一次的位置加 1。这样，就确保了每次新搜索都从上一次找到的子字符串的后面开始。每次搜索返回的位置依次被保存在数组 positions 中，以便将来使用。

4. trim()方法

trim()

ECMAScript 5 为所有字符串定义了 trim()方法。这个方法会创建一个字符串的副本，删除前置及后缀的所有空格，然后返回结果。例如：

```
var stringValue = "   hello world   ";
var trimmedStringValue = stringValue.trim();
alert(stringValue);             //"   hello world   "
alert(trimmedStringValue);      //"hello world"
```

5. 字符串大小写转换方法

toLowerCase()、toLocaleLowerCase()、toUpperCase()和 toLocaleUpperCase()

接下来我们要介绍的是一组与大小写转换有关的方法。ECMAScript 中涉及字符串大小写转换的方法有 4 个：toLowerCase()、toLocaleLowerCase()、toUpperCase()和 toLocaleUpperCase()。其中，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。以下是几个例子。

```
var stringValue = "hello world";
alert(stringValue.toLocaleUpperCase());  //"HELLO WORLD"
alert(stringValue.toUpperCase());        //"HELLO WORLD"
alert(stringValue.toLocaleLowerCase());  //"hello world"
alert(stringValue.toLowerCase());        //"hello world"
```

StringTypeCaseMethodExample01.htm

以上代码调用的 toLocaleUpperCase()和 toUpperCase()都返回了"HELLO WORLD"，就像调用 toLocaleLowerCase()和 toLowerCase()都返回"hello world"一样。一般来说，在不知道自己的代码将在哪种语言环境中运行的情况下，还是使用针对地区的方法更稳妥一些。

6. 字符串的模式匹配方法

match()

String 类型定义了几个用于在字符串中匹配模式的方法。第一个方法就是 match()，在字符串上调用这个方法，本质上与调用 RegExp 的 exec()方法相同。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。来看下面的例子。

```
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
```

本例中的 match()方法返回了一个数组；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。

search()

另一个用于查找模式的方法是 search()。这个方法的唯一参数与 match()方法的参数相同：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。看下面的例子。

```
var text = "cat, bat, sat, fat";
var pos = text.search(/at/);
alert(pos);   //1
```

replace()

为了简化替换子字符串的操作，ECMAScript 提供了 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，如下所示。

```
var text = "cat, bat, sat, fat";
var result = text.replace("at", "ond");
alert(result);    //"cond, bat, sat, fat"

result = text.replace(/at/g, "ond");
alert(result);    //"cond, bond, sond, fond"
```

如果第二个参数是字符串，那么还可以使用一些特殊的字符序列，将正则表达式操作得到的值插入到结果字符串中。下表列出了 ECMAScript 提供的这些特殊的字符序列。

字符序列	替换文本
$$	$
$&	匹配整个模式的子字符串。与RegExp.lastMatch的值相同
$'	匹配的子字符串之前的子字符串。与RegExp.leftContext的值相同
$`	匹配的子字符串之后的子字符串。与RegExp.rightContext的值相同
$n	匹配第*n*个捕获组的子字符串，其中*n*等于0～9。例如，$1是匹配第一个捕获组的子字符串，$2是匹配第二个捕获组的子字符串，以此类推。如果正则表达式中没有定义捕获组，则使用空字符串
$nn	匹配第*nn*个捕获组的子字符串，其中*nn*等于01～99。例如，$01是匹配第一个捕获组的子字符串，$02是匹配第二个捕获组的子字符串，以此类推。如果正则表达式中没有定义捕获组，则使用空字符串

通过这些特殊的字符序列，可以使用最近一次匹配结果中的内容，如下面的例子所示。

```
var text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
alert(result);    //word (cat), word (bat), word (sat), word (fat)
```

replace()方法的第二个参数也可以是一个函数。在只有一个匹配项（即与模式匹配的字符串）的情况下，会向这个函数传递 3 个参数：模式的匹配项、模式匹配项在字符串中的位置和原始字符串。在正则表达式中定义了多个捕获组的情况下，传递给函数的参数依次是模式的匹配项、第一个捕获组的匹配项、第二个捕获组的匹配项……，但最后两个参数仍然分别是模式的匹配项在字符串中的位置和原始字符串。这个函数应该返回一个字符串，表示应该被替换的匹配项使用函数作为 replace()方法的第二个参数可以实现更加精细的替换操作，请看下面这个例子。

```
function htmlEscape(text){
    return text.replace(/[<>"&]/g, function(match, pos, originalText){
        switch(match){
            case "<":
                return "&lt;";
            case ">":
                return "&gt;";
            case "&":
                return "&amp;";
            case "\"":
                return "&quot;";
        }
    });
}

alert(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
//&lt;p class=&quot;greeting&quot;&gt;Hello world!&lt;/p&gt;
```

split()

最后一个与模式匹配有关的方法是 split()，这个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。请看下面的例子。

```
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(/[^\,]+/);     //["", ",", ",", ",", ""]
```

要了解关于 split()方法以及捕获组的跨浏览器问题的更多讨论，请参考 Steven Levithan 的文章“JavaScript split bugs：Fixed!”（http://blog.stevenlevithan.com/archives/cross-browser-split）。

7. localeCompare()方法

localeCompare()

7. localeCompare()方法

与操作字符串有关的最后一个方法是localeCompare()，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
```

StringTypeLocaleCompareExample01.htm

这个例子比较了字符串"yellow"和另外几个值："brick"、"yellow"和"zoo"。因为"brick"在字母表中排在"yellow"之前，所以localeCompare()返回了1；而"yellow"等于"yellow"，所以localeCompare()返回了0；最后，"zoo"在字母表中排在"yellow"后面，所以localeCompare()返回了-1。再强调一次，因为localeCompare()返回的数值取决于实现，所以最好是像下面例子所示的这样使用这个方法。

```
function determineOrder(value) {
    var result = stringValue.localeCompare(value);
    if (result < 0){
        alert("The string 'yellow' comes before the string '" + value + "'.");
    } else if (result > 0) {
        alert("The string 'yellow' comes after the string '" + value + "'.");
    } else {
```



```
        alert("The string 'yellow' is equal to the string '" + value + "'.");
    }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

StringTypeLocaleCompareExample01.htm

8. fromCharCode()方法

另外，String构造函数本身还有一个静态方法：fromCharCode()。这个方法的任务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"
```

StringTypeFromCharCodeExample01.htm

在这里，我们给fromCharCode()传递的是字符串"hello"中每个字母的字符编码。

9. HTML方法

早期的 Web 浏览器提供商觉察到了使用 JavaScript 动态格式化 HTML 的需求。于是，这些提供商就扩展了标准，实现了一些专门用于简化常见 HTML 格式化任务的方法。下表列出了这些 HTML 方法。不过，需要请读者注意的是，应该尽量不使用这些方法，因为它们创建的标记通常无法表达语义。

方　法	输出结果
anchor(*name*)	*string*
big()	<big>*string*</big>
bold()	*string*
fixed()	<tt>*string*</tt>
fontcolor(*color*)	*string*
fontsize(*size*)	*string*
italics()	*string*
link(url)	*string*
small()	*string*
strike()	<strike>*string*</strike>
sub()	_{*string*}
sup()	^{*string*}