# 基本包装类型

为了便于操作基本类型值，ECMAScript 还提供了 3 个特殊的引用类型： Boolean 、 Number 和String 。这些类型与其他引用类型相似，但同时也具有与各自的基本类型相应的特殊行为。

实际上，每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，从而让我们能够调用一些方法来操作这些数据。来看下面的例子。

var s1 = "some text";

var s2 = s1.substring(2);

这个例子中的变量 s1 包含一个字符串，字符串当然是基本类型值。而下一行调用了 s1 的

substring() 方法，并将返回的结果保存在了 s2 中。我们知道，基本类型值不是对象，因而从逻辑上讲它们不应该有方法（尽管如我们所愿，它们确实有方法）。其实，为了让我们实现这种直观的操作，后台已经自动完成了一系列的处理。当第二行代码访问 s1 时，访问过程处于一种读取模式，也就是要从内存中读取这个字符串的值。而在读取模式中访问字符串时，后台都会自动完成下列处理。

(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。

可以将以上三个步骤想象成是执行了下列 ECMAScript 代码。

var s1 = new String("some text");

var s2 = s1.substring(2);

s1 = null;

经过此番处理，基本的字符串值就变得跟对象一样了。而且，上面这三个步骤也分别适用于 Boolean和 Number 类型对应的布尔值和数字值。

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。这意味着我们不能在运行时为基本类型值添加属性和方法。来看下面的例子：

var s1 = "some text";

s1.color = "red";

alert(s1.color); //undefined

在此，第二行代码试图为字符串 s1 添加一个 color 属性。但是，当第三行代码再次访问 s1 时，其 color 属性不见了。问题的原因就是第二行创建的 String 对象在执行第三行代码时已经被销毁了。

第三行代码又创建自己的 String 对象，而该对象没有 color 属性。

当然，可以显式地调用 Boolean 、 Number 和 String 来创建基本包装类型的对象**。不过，应该在绝对必要的情况下再这样做，因为这种做法很容易让人分不清自己是在处理基本类型还是引用类型的值**。对基本包装类型的实例调用 typeof 会返回 "object" ，而且所有基本包装类型的对象都会被转换为布尔值 true 。

## Boolean 类型

Boolean 类型是与布尔值对应的引用类型。要创建 Boolean 对象，可以像下面这样调用 Boolean构造函数并传入 true 或 false 值。

var booleanObject = new Boolean(true);

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" 。可是， Boolean 对象在 ECMAScript 中的用处不大，因为它经常会造成人们的误解。我们的建议是永远不要使

用 Boolean 对象。

## Number 类型

Number 是与数字值对应的引用类型。要创建 Number 对象，可以在调用 Number 构造函数时向其中传递相应的数值。下面是一个例子。

var numberObject = new Number(10);

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。我们可以为 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，如下面的例子所示。

var num = 10;

alert(num.toString()); //"10"

alert(num.toString(2)); //"1010"

alert(num.toString(8)); //"12"

alert(num.toString(10)); //"10"

alert(num.toString(16)); //"a"

除了继承的方法之外， Number 类型还提供了一些用于将数值格式化为字符串的方法。

其中， toFixed() 方法会按照指定的小数位返回数值的字符串表示，例如：

var num = 10;

alert(num.toFixed(2)); //"10.00"

这里给 toFixed() 方法传入了数值 2 ，意思是显示几位小数。于是，这个方法返回了 "10.00" ，即以 0 填补了必要的小数位。如果数值本身包含的小数位比指定的还多，那么接近指定的最大小数位的值就会舍入，如下面的例子所示。

var num = 10.005;

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

能够自动舍入的特性，使得 toFixed() 方法很适合处理货币值。

与 Boolean 对象类似， Number 对象也以后台方式为数值提供了重要的功能。但与此同时，我们**仍然不建议直接实例化 Number 类型**，而原因与显式创建 Boolean 对象一样。

## 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 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。

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。

String 类型提供了很多方法，用于辅助完成对 ECMAScript 中字符串的解析和操作。

### 1. 字符方法

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"

这个例子输出的是 "101" ，也就是小写字母 "e" 的字符编码。

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"

使用方括号表示法访问个别字符的语法得到了 IE8 及 Firefox、Safari、Chrome 和 Opera 所有版本的支持。如果是在 IE7 及更早版本中使用这种语法，会返回 undefined 值（尽管根本不是特殊的undefined 值）。

### 2. 字符串操作方法

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"

在这个例子中，通过 stringValue 调用 concat() 方法返回的结果是 "hello world" ——但stringValue 的值则保持不变。实际上， 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。再看一个例子：

var stringValue = "hello ";

var result = stringValue.concat("world", "!");

alert(result); //"hello world!"

alert(stringValue); //"hello"

这个例子将 "world" 和 "!" 拼接到了 "hello" 的末尾。**虽然 concat() 是专门用来拼接字符串的方法，但实践中使用更多的还是加号操作符（+）。**而且，使用加号操作符在大多数情况下都比使用 concat()方法要简便易行（特别是在拼接多个字符串的情况下）。

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"

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。

### 3. 字符串位置方法

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。

var stringValue = "hello world";

alert(stringValue.indexOf("o")); //4

alert(stringValue.lastIndexOf("o")); //7

子字符串 "o" 第一次出现的位置是 4，即 "hello" 中的 "o" ；最后一次出现的位置是 7，即 "world" 中的"o" 。如果 "o" 在这个字符串中仅出现了一次，那么 indexOf() 和 lastIndexOf() 会返回相同的位置值。

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

在将第二个参数 6 传递给这两个方法之后，得到了与前面例子相反的结果。这一次，由于

indexOf() 是从位置 6（字母 "w" ）开始向后搜索，结果在位置 7 找到了 "o" ，因此它返回 7。而 lastIndexOf() 是从位置 6 开始向前搜索。结果找到了 "hello" 中的 "o" ，因此它返回 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：

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"

### 4. trim() 方法

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"

由于 trim() 返回的是字符串的副本，所以原始字符串中的前置及后缀空格会保持不变。

### 5. 字符串大小写转换方法

ECMAScript 中涉及字符串大小写转换的方法有 toLowerCase() 、 toUpperCase() 。

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。

### 6. 字符串的模式匹配方法

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。

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

这个例子中的 search() 方法返回 1 ，即 "at" 在字符串中第一次出现的位置。

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"

在这个例子中，首先传入 replace() 方法的是字符串 "at" 和替换用的字符串 "ond" 。替换的结果是把 "cat" 变成了 "cond" ，但字符串中的其他字符并没有受到影响。然后，通过将第一个参数修改为带有全局标志的正则表达式，就将全部 "at" 都替换成了 "ond" 。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。

### 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

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");

使用这种结构，就可以确保自己的代码在任何实现中都可以正确地运行了。

localeCompare() 方法比较与众不同的地方，就是实现所支持的地区（国家和语言）决定了这个方法的行为。比如，美国以英语作为 ECMAScript 实现的标准语言，因此 localeCompare() 就是区分大小写的，于是大写字母在字母表中排在小写字母前头就成为了一项决定性的比较规则。不过，在其他地区恐怕就不是这种情况了。

### 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"

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。