《JavaScript 闯关记》之基本包装类型

javascript 劼哥stone 10月26日发布

为了便于操作基本类型值，JavaScript 还提供了3个特殊的引用类型：`Boolean`、`Number` 和 `String`。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 JavaScript 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";
console.log(s1.color);   // undefined
```

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`。

`Object` 构造函数也会像工厂方法一样，根据传入值的类型返回相应基本包装类型的实例。例如：

```
var obj = new Object("some text");
console.log(obj instanceof String);   // true
```

把字符串传给 `Object` 构造函数，就会创建 `String` 的实例；而传入数值参数会得到 `Number` 的实例，传入布尔值参数就会得到 `Boolean` 的实例。

要注意的是，使用 `new` 调用基本包装类型的构造函数，与直接调用同名的转型函数是不一样的。 例如：

```
var value = "25";
var number = Number(value);  // 转型函数
console.log(typeof number);  // "number"

var obj = new Number(value); // 构造函数
console.log(typeof obj);     // "object"
```

尽管我们不建议显式地创建基本包装类型的对象，但它们操作基本类型值的能力还是相当重要的。而每个基本包装类型都提供了操作相应值的便捷方法。

`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 JavaScript 中的用处不大，因为它经常会造成人们的误解。其中最常见的问题就是在布尔表达式中使用 `Boolean` 对象，例如：

```
var falseObject = new Boolean(false);
var result = falseObject && true;
console.log(result);  // true

var falseValue = false;
result = falseValue && true;
console.log(result);  // false
```

在这个例子中，我们使用 `false` 值创建了一个 `Boolean` 对象。然后，将这个对象与基本类型值 `true` 构成了逻辑与表达式。在布尔运算中，`false && true` 等于 `false`。可是，示例中的这行代码是对 `falseObject` 而不是对它的值 `false` 进行求值。布尔表达式中的所有对象都会被转换为 `true`，因此 `falseObject` 对象在布尔表达式中代表的是 `true`。结果，`true && true` 当然就等于 `true` 了。

基本类型与引用类型的布尔值还有两个区别。首先，`typeof` 操作符对基本类型返回 `"boolean"`，而对引用类型返回 `"object"`。其次，由于 `Boolean` 对象是 `Boolean` 类型的实例，所以使用 `instanceof` 操作符测试 `Boolean` 对象会返回 `true`，而测试基本类型的布尔值则返回 `false`。例如：

```
console.log(typeof falseObject);   // object
console.log(typeof falseValue);    // boolean
console.log(falseObject instanceof Boolean);  // true
console.log(falseValue instanceof Boolean);   // false
```

理解基本类型的布尔值与 `Boolean` 对象之间的区别非常重要，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。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;
console.log(num.toString());     // "10"
console.log(num.toString(2));    // "1010"
console.log(num.toString(8));    // "12"
console.log(num.toString(10));   // "10"
console.log(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;
console.log(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;
console.log(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。

> 但需要注意的是，不同浏览器给这个方法设定的舍入规则可能会有所不同。
>
> 在给 `toFixed()` 传入0的情况下，IE8 及之前版本不能正确舍入范围在{(-0.94,-0.5],[0.5,0.94)}之间的值。对于这个范围内的值，IE8 会返回0，而不是-1或1；其他浏览器都能返回正确的值。IE9 修复了这个问题。
>
> `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。

另外可用于格式化数值的方法是 `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。与 `toFixed()` 一样，`toExponential()` 也接收一个参数，而且该参数同样也是指定输出结果中的小数位数。看下面的例子。

```
var num = 10;
console.log(num.toExponential(1));     // "1.0e+1"
```

以上代码输出了 `"1.0e+1"`；不过，这么小的数值一般不必使用 e 表示法。如果你想得到表示某个数值的最合适的格式，就应该使用 `toPrecision()` 方法。

对于一个数值来说，`toPrecision()` 方法可能会返回固定大小（fixed）格式，也可能返回指数（exponential）格式；具体规则是看哪种格式最合适。这个方法接收一个参数，即表示数值的所有数字的位数（不包括指数部分）。请看下面的例子。

```
var num = 99;
console.log(num.toPrecision(1));     // "1e+2"
console.log(num.toPrecision(2));     // "99"
console.log(num.toPrecision(3));     // "99.0"
```

以上代码首先完成的任务是以一位数来表示 `99`，结果是 `"1e+2"`，即 `100`。因为一位数无法准确地表示 `99`，因此 `toPrecision()` 就将它向上舍入为 `100`，这样就可以使用一位数来表示它了。而接下来的用两位数表示 `99`，当然还是 `"99"`。最后，在想以三位数表示 `99` 时，`toPrecision()` 方法返回了 `"99.0"`。实际上，`toPrecision()` 会根据要处理的数值决定到底是调用 `toFixed()` 还是调用 `toExponential()`。而这三个方法都可以通过向上或向下舍入，做到以最准确的形式来表示带有正确小数位的值。

> `toPrecision()` 方法可以表现1到21位小数。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。

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。具体来讲，就是在使用 `typeof` 和 `instanceof` 操作符测试基本类型数值与引用类型数值时，得到的结果完全不同，如下面的例子所示。

```
var numberObject = new Number(10);
var numberValue = 10;
console.log(typeof numberObject);   // "object"
console.log(typeof numberValue);    // "number"
console.log(numberObject instanceof Number);  // true
console.log(numberValue instanceof Number);   // false
```

`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";
console.log(stringValue.length);     // 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。例如：

```
var stringValue = "大家好";
console.log(stringValue.length);     // 3
```

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

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（JavaScript 中没有字符类型）。例如：

```
var stringValue = "hello world";
console.log(stringValue.charAt(1));  // "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。例如：

```
var stringValue = "hello world";
console.log(stringValue.charCodeAt(1));  // 101，101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";
console.log(stringValue[1]);   // "e"
```

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");

console.log(result);        // "hello world"
console.log(stringValue);   // "hello"
```

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", "!");

console.log(result);        // "hello world!"
console.log(stringValue);   // "hello"
```

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）。

JavaScript 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";
console.log(stringValue.slice(3));              // "lo world"
console.log(stringValue.substring(3));          // "lo world"
console.log(stringValue.substr(3));             // "lo world"
console.log(stringValue.slice(3, 7));           // "lo w"
console.log(stringValue.substring(3,7));        // "lo w"
console.log(stringValue.substr(3, 7));          // "lo worl"
```

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

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。还是来看一个例子吧。

```
var stringValue = "hello world";
console.log(stringValue.indexOf("o"));          // 4
console.log(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";
console.log(stringValue.indexOf("o", 6));       // 7
console.log(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：

```
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);
}
console.log(positions);    // "3,24,32,35,52"
```

`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();
console.log(stringValue);           // "   hello world   "
console.log(trimmedStringValue);    // "hello world"
```

字符串大小写转换方法

JavaScript 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";
console.log(stringValue.toLocaleUpperCase());  // "HELLO WORLD"
console.log(stringValue.toUpperCase());        // "HELLO WORLD"
console.log(stringValue.toLocaleLowerCase());  // "hello world"
console.log(stringValue.toLowerCase());        // "hello world"
```

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。

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);
console.log(matches.index);               // 0
console.log(matches[0]);                  // "cat"
console.log(pattern.lastIndex);           // 0
```

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/);
console.log(pos);   // 1，即"at"第一次出现的位置
```

为了简化替换子字符串的操作，JavaScript 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");
console.log(result);    // "cond, bat, sat, fat"

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

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"]
```

`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";
console.log(stringValue.localeCompare("brick"));    // 1
console.log(stringValue.localeCompare("yellow"));   // 0
console.log(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){
        console.log("The string 'yellow' comes before the string '" + value + "'.");
    } else if (result > 0) {
        console.log("The string 'yellow' comes after the string '" + value + "'.");
    } else {
        console.log("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 JavaScript 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。

`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：

```
console.log(String.fromCharCode(104, 101, 108, 108, 111)); // "hello"

var s = 'hello';
for(let i=0;i<s.length;i++){
  console.log(`${s[i]}----${s[i].charCodeAt()}`);
}
/*
"h----104"
"e----101"
"l----108"
"l----108"
"o----111"
*/
```

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。

关卡

```
// 挑战一
var falseObject = new Object(false);
console.log(typeof falseObject);              // ???
console.log(falseObject instanceof Object);   // ???
console.log(falseObject instanceof Boolean); // ???
```

```
// 挑战二
var numberObject = new Object(100);
console.log(typeof numberObject);              // ???
console.log(numberObject instanceof Object);   // ???
console.log(numberObject instanceof Number);   // ???
```

```
// 挑战三
var stringObject = new Object("abcde");
console.log(typeof stringObject);              // ???
console.log(stringObject instanceof Object);  // ???
console.log(stringObject instanceof String);  // ???
```

```
// 挑战四，翻转一个字符串
// 提示：可以使用数组的 reverse() 方法
var reverse = function(str) {
    // 待实现方法体
}
console.log(reverse("hello"));  // "olleh"
```

更多

> 关注微信公众号「劼哥舍」回复「答案」，获取关卡详解。
> 关注 https://github.com/stone0090/javascript-lessons，获取最新动态。

10月26日发布　更多

你可能感兴趣的文章

javascript javascript javascript javascript　1 收藏，1k 浏览

Javascript 数组　318 浏览

Strict Mode - Javascript语法基础 - Javascript核心　8 收藏，3.5k 浏览

讨论区

本文隶属于专栏

劼哥舍

欢迎来到「劼哥舍」，您非要念成「劫个色」也行。这里坚持原创，分享随笔和学习心得，主要涉及 前端 / .NET / Java 等方面的内容。欢迎交流，欢迎提问，欢迎转载，但需注明出处。

劼哥stone
作者　关注作者

关注专栏

系列文章

《JavaScript 闯关记》之对象 6 收藏， 164 浏览

《JavaScript 闯关记》之数组 6 收藏， 232 浏览

《JavaScript 闯关记》之函数 3 收藏， 161 浏览

《JavaScript 闯关记》之正则表达式 30 收藏， 808 浏览

《JavaScript 闯关记》之单体内置对象 11 收藏， 419 浏览

《JavaScript 闯关记》之 BOM 14 收藏， 525 浏览

《JavaScript 闯关记》之 DOM（上） 3 收藏， 232 浏览

相关收藏夹

换一组

前端技术

4 个条目 | 1 人关注

mark

4 个条目 | 0 人关注

人物杂谈

5 个条目 | 1 人关注

分享扩散：