5.1.2　日期格式化方法

Date 类型有几个专门用于格式化日期的方法，它们都会返回字符串：

- toDateString()显示日期中的周几、月、日、年（格式特定于实现）；
- toTimeString()显示日期中的时、分、秒和时区（格式特定于实现）；
- toLocaleDateString()显示日期中的周几、月、日、年（格式特定于实现和地区）；
- toLocaleTimeString()显示日期中的时、分、秒（格式特定于实现和地区）；
- toUTCString()显示完整的 UTC 日期（格式特定于实现）。

这些方法的输出与 toLocaleString()和 toString()一样，会因浏览器而异。因此不能用于在用户界面上一致地显示日期。

> **注意**　还有一个方法叫 toGMTString()，这个方法跟 toUTCString()是一样的，目的是为了向后兼容。不过，规范建议新代码使用 toUTCString()。

5.1.3　日期/时间组件方法

Date 类型剩下的方法（见下表）直接涉及取得或设置日期值的特定部分。注意表中“UTC 日期”，指的是没有时区偏移（将日期转换为 GMT）时的日期。

方　法	说　明
getTime()	返回日期的毫秒表示；与 valueOf()相同
setTime(*milliseconds*)	设置日期的毫秒表示，从而修改整个日期
getFullYear()	返回 4 位数年（即 2019 而不是 19）
getUTCFullYear()	返回 UTC 日期的 4 位数年
setFullYear(*year*)	设置日期的年（*year* 必须是 4 位数）
setUTCFullYear(*year*)	设置 UTC 日期的年（*year* 必须是 4 位数）
getMonth()	返回日期的月（0 表示 1 月，11 表示 12 月）
getUTCMonth()	返回 UTC 日期的月（0 表示 1 月，11 表示 12 月）
setMonth(*month*)	设置日期的月（*month* 为大于 0 的数值，大于 11 加年）
setUTCMonth(*month*)	设置 UTC 日期的月（*month* 为大于 0 的数值，大于 11 加年）
getDate()	返回日期中的日（1~31）
getUTCDate()	返回 UTC 日期中的日（1~31）
setDate(*date*)	设置日期中的日（如果 *date* 大于该月天数，则加月）
setUTCDate(*date*)	设置 UTC 日期中的日（如果 *date* 大于该月天数，则加月）
getDay()	返回日期中表示周几的数值（0 表示周日，6 表示周六）
getUTCDay()	返回 UTC 日期中表示周几的数值（0 表示周日，6 表示周六）
getHours()	返回日期中的时（0~23）
getUTCHours()	返回 UTC 日期中的时（0~23）
setHours(*hours*)	设置日期中的时（如果 *hours* 大于 23，则加日）

（续）

方　法	说　明
setUTCHours(*hours*)	设置 UTC 日期中的时（如果 *hours* 大于 23，则加日）
getMinutes()	返回日期中的分（0~59）
getUTCMinutes()	返回 UTC 日期中的分（0~59）
setMinutes(*minutes*)	设置日期中的分（如果 *minutes* 大于 59，则加时）
setUTCMinutes(*minutes*)	设置 UTC 日期中的分（如果 *minutes* 大于 59，则加时）
getSeconds()	返回日期中的秒（0~59）
getUTCSeconds()	返回 UTC 日期中的秒（0~59）
setSeconds(*seconds*)	设置日期中的秒（如果 *seconds* 大于 59，则加分）
setUTCSeconds(*seconds*)	设置 UTC 日期中的秒（如果 *seconds* 大于 59，则加分）
getMilliseconds()	返回日期中的毫秒
getUTCMilliseconds()	返回 UTC 日期中的毫秒
setMilliseconds(*milliseconds*)	设置日期中的毫秒
setUTCMilliseconds(*milliseconds*)	设置 UTC 日期中的毫秒
getTimezoneOffset()	返回以分钟计的 UTC 与本地时区的偏移量（如美国 EST 即“东部标准时间”返回 300，进入夏令时的地区可能有所差异）

5.2 RegExp

ECMAScript 通过 RegExp 类型支持正则表达式。正则表达式使用类似 Perl 的简洁语法来创建：

```
let expression = /pattern/flags;
```

这个正则表达式的 pattern（模式）可以是任何简单或复杂的正则表达式，包括字符类、限定符、分组、向前查找和反向引用。每个正则表达式可以带零个或多个 flags（标记），用于控制正则表达式的行为。下面给出了表示匹配模式的标记。

- g：全局模式，表示查找字符串的全部内容，而不是找到第一个匹配的内容就结束。
- i：不区分大小写，表示在查找匹配时忽略 pattern 和字符串的大小写。
- m：多行模式，表示查找到一行文本末尾时会继续查找。
- y：粘附模式，表示只查找从 lastIndex 开始及之后的字符串。
- u：Unicode 模式，启用 Unicode 匹配。
- s：dotAll 模式，表示元字符.匹配任何字符（包括\n 或\r）。

使用不同模式和标记可以创建出各种正则表达式，比如：

```
// 匹配字符串中的所有"at"
let pattern1 = /at/g;

// 匹配第一个"bat"或"cat"，忽略大小写
let pattern2 = /[bc]at/i;

// 匹配所有以"at"结尾的三字符组合，忽略大小写
let pattern3 = /.at/gi;
```

与其他语言中的正则表达式类似，所有**元字符**在模式中也必须转义，包括：

```
( [ { \ ^ $ | ) ] } ? * + .
```

元字符在正则表达式中都有一种或多种特殊功能，所以要匹配上面这些字符本身，就必须使用反斜杠来转义。下面是几个例子：

```
// 匹配第一个"bat"或"cat"，忽略大小写
let pattern1 = /[bc]at/i;

// 匹配第一个"[bc]at"，忽略大小写
let pattern2 = /\[bc\]at/i;

// 匹配所有以"at"结尾的三字符组合，忽略大小写
let pattern3 = /.at/gi;

// 匹配所有".at"，忽略大小写
let pattern4 = /\.at/gi;
```

这里的 pattern1 匹配"bat"或"cat"，不区分大小写。要直接匹配"[bc]at"，左右中括号都必须像 pattern2 中那样使用反斜杠转义。在 pattern3 中，点号表示"at"前面的任意字符都可以匹配。如果想匹配".at"，那么要像 pattern4 中那样对点号进行转义。

前面例子中的正则表达式都是使用字面量形式定义的。正则表达式也可以使用 RegExp 构造函数来创建，它接收两个参数：模式字符串和（可选的）标记字符串。任何使用字面量定义的正则表达式也可以通过构造函数来创建，比如：

```
// 匹配第一个"bat"或"cat"，忽略大小写
let pattern1 = /[bc]at/i;

// 跟 pattern1 一样，只不过是用构造函数创建的
let pattern2 = new RegExp("[bc]at", "i");
```

这里的 pattern1 和 pattern2 是等效的正则表达式。注意，RegExp 构造函数的两个参数都是字符串。因为 RegExp 的模式参数是字符串，所以在某些情况下需要二次转义。所有元字符都必须二次转义，包括转义字符序列，如\n（\转义后的字符串是\\，在正则表达式字符串中则要写成\\\\）。下表展示了几个正则表达式的字面量形式，以及使用 RegExp 构造函数创建时对应的模式字符串。

字面量模式	对应的字符串
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

此外，使用 RegExp 也可以基于已有的正则表达式实例，并可选择性地修改它们的标记：

```
const re1 = /cat/g;
console.log(re1);  // "/cat/g"

const re2 = new RegExp(re1);
console.log(re2);  // "/cat/g"

const re3 = new RegExp(re1, "i");
console.log(re3);  // "/cat/i"
```

5.2.1 RegExp 实例属性

每个 RegExp 实例都有下列属性，提供有关模式的各方面信息。

- global：布尔值，表示是否设置了 g 标记。
- ignoreCase：布尔值，表示是否设置了 i 标记。
- unicode：布尔值，表示是否设置了 u 标记。
- sticky：布尔值，表示是否设置了 y 标记。
- lastIndex：整数，表示在源字符串中下一次搜索的开始位置，始终从 0 开始。
- multiline：布尔值，表示是否设置了 m 标记。
- dotAll：布尔值，表示是否设置了 s 标记。
- source：正则表达式的字面量字符串（不是传给构造函数的模式字符串），没有开头和结尾的斜杠。
- flags：正则表达式的标记字符串。始终以字面量而非传入构造函数的字符串模式形式返回（没有前后斜杠）。

通过这些属性可以全面了解正则表达式的信息，不过实际开发中用得并不多，因为模式声明中包含这些信息。下面是一个例子：

```
let pattern1 = /\[bc\]at/i;

console.log(pattern1.global);      // false
console.log(pattern1.ignoreCase);  // true
console.log(pattern1.multiline);   // false
console.log(pattern1.lastIndex);   // 0
console.log(pattern1.source);      // "\[bc\]at"
console.log(pattern1.flags);       // "i"

let pattern2 = new RegExp("\\[bc\\]at", "i");

console.log(pattern2.global);      // false
console.log(pattern2.ignoreCase);  // true
console.log(pattern2.multiline);   // false
console.log(pattern2.lastIndex);   // 0
console.log(pattern2.source);      // "\[bc\]at"
console.log(pattern2.flags);       // "i"
```

注意，虽然第一个模式是通过字面量创建的，第二个模式是通过 RegExp 构造函数创建的，但两个模式的 source 和 flags 属性是相同的。source 和 flags 属性返回的是规范化之后可以在字面量中使用的形式。

5.2.2 RegExp 实例方法

RegExp 实例的主要方法是 exec()，主要用于配合捕获组使用。这个方法只接收一个参数，即要应用模式的字符串。如果找到了匹配项，则返回包含第一个匹配信息的数组；如果没找到匹配项，则返回 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。index 是字符串中匹配模式的起始位置，input 是要查找的字符串。这个数组的第一个元素是匹配整个模式的字符串，其他元素是与表达式中的捕获组匹配的字符串。如果模式中没有捕获组，则数组只包含一个元素。来看下面的例子：

```
let text = "mom and dad and baby";
let pattern = /mom( and dad( and baby)?)?/gi;

let matches = pattern.exec(text);
console.log(matches.index);   // 0
console.log(matches.input);   // "mom and dad and baby"
console.log(matches[0]);      // "mom and dad and baby"
console.log(matches[1]);      // " and dad and baby"
console.log(matches[2]);      // " and baby"
```

在这个例子中，模式包含两个捕获组：最内部的匹配项" and baby"，以及外部的匹配项" and dad" 或" and dad and baby"。调用 exec()后找到了一个匹配项。因为整个字符串匹配模式，所以 matchs 数组的 index 属性就是 0。数组的第一个元素是匹配的整个字符串，第二个元素是匹配第一个捕获组的字符串，第三个元素是匹配第二个捕获组的字符串。

如果模式设置了全局标记，则每次调用 exec()方法会返回一个匹配的信息。如果没有设置全局标记，则无论对同一个字符串调用多少次 exec()，也只会返回第一个匹配的信息。

```
let text = "cat, bat, sat, fat";
let pattern = /.at/;

let matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches[0]);         // cat
console.log(pattern.lastIndex);  // 0

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

上面例子中的模式没有设置全局标记，因此调用 exec()只返回第一个匹配项("cat")。lastIndex 在非全局模式下始终不变。

如果在这个模式上设置了 g 标记，则每次调用 exec()都会在字符串中向前搜索下一个匹配项，如下面的例子所示：

```
let text = "cat, bat, sat, fat";
let pattern = /.at/g;
let matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches[0]);         // cat
console.log(pattern.lastIndex);  // 3

matches = pattern.exec(text);
console.log(matches.index);      // 5
console.log(matches[0]);         // bat
console.log(pattern.lastIndex);  // 8

matches = pattern.exec(text);
console.log(matches.index);      // 10
console.log(matches[0]);         // sat
console.log(pattern.lastIndex);  // 13
```

这次模式设置了全局标记，因此每次调用 exec()都会返回字符串中的下一个匹配项，直到搜索到字符串末尾。注意模式的 lastIndex 属性每次都会变化。在全局匹配模式下，每次调用 exec()都会更新 lastIndex 值，以反映上次匹配的最后一个字符的索引。

如果模式设置了粘附标记 y，则每次调用 exec()就只会在 lastIndex 的位置上寻找匹配项。粘附标记覆盖全局标记。

```
let text = "cat, bat, sat, fat";
let pattern = /.at/y;

let matches = pattern.exec(text);
console.log(matches.index);        // 0
console.log(matches[0]);           // cat
console.log(pattern.lastIndex);    // 3

// 以索引 3 对应的字符开头找不到匹配项，因此 exec()返回 null
// exec()没找到匹配项，于是将 lastIndex 设置为 0
matches = pattern.exec(text);
console.log(matches);              // null
console.log(pattern.lastIndex);    // 0

// 向前设置 lastIndex 可以让粘附的模式通过 exec()找到下一个匹配项：
pattern.lastIndex = 5;
matches = pattern.exec(text);
console.log(matches.index);        // 5
console.log(matches[0]);           // bat
console.log(pattern.lastIndex);    // 8
```

正则表达式的另一个方法是 test()，接收一个字符串参数。如果输入的文本与模式匹配，则参数返回 true，否则返回 false。这个方法适用于只想测试模式是否匹配，而不需要实际匹配内容的情况。test()经常用在 if 语句中：

```
let text = "000-00-0000";
let pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)) {
  console.log("The pattern was matched.");
}
```

在这个例子中，正则表达式用于测试特定的数值序列。如果输入的文本与模式匹配，则显示匹配成功的消息。这个用法常用于验证用户输入，此时我们只在乎输入是否有效，不关心为什么无效。

无论正则表达式是怎么创建的，继承的方法 toLocaleString()和 toString()都返回正则表达式的字面量表示。比如：

```
let pattern = new RegExp("\\[bc\\]at", "gi");
console.log(pattern.toString());       // /\[bc\]at/gi
console.log(pattern.toLocaleString()); // /\[bc\]at/gi
```

这里的模式是通过 RegExp 构造函数创建的，但 toLocaleString()和 toString()返回的都是其字面量的形式。

> **注意** 正则表达式的 valueOf()方法返回正则表达式本身。

5.2.3 RegExp 构造函数属性

RegExp 构造函数本身也有几个属性。（在其他语言中，这种属性被称为静态属性。）这些属性适用于作用域中的所有正则表达式，而且会根据最后执行的正则表达式操作而变化。这些属性还有一个特点，

就是可以通过两种不同的方式访问它们。换句话说，每个属性都有一个全名和一个简写。下表列出了 RegExp 构造函数的属性。

全　名	简　写	说　明
input	$_	最后搜索的字符串（非标准特性）
lastMatch	$&	最后匹配的文本
lastParen	$+	最后匹配的捕获组（非标准特性）
leftContext	$`	input 字符串中出现在 lastMatch 前面的文本
rightContext	$'	input 字符串中出现在 lastMatch 后面的文本

通过这些属性可以提取出与 exec()和 test()执行的操作相关的信息。来看下面的例子：

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

if (pattern.test(text)) {
  console.log(RegExp.input);        // this has been a short summer
  console.log(RegExp.leftContext);  // this has been a
  console.log(RegExp.rightContext); // summer
  console.log(RegExp.lastMatch);    // short
  console.log(RegExp.lastParen);    // s
}
```

以上代码创建了一个模式，用于搜索任何后跟"hort"的字符，并把第一个字符放在了捕获组中。不同属性包含的内容如下。

- input 属性中包含原始的字符串。
- leftConext 属性包含原始字符串中"short"之前的内容，rightContext 属性包含"short"之后的内容。
- lastMatch 属性包含匹配整个正则表达式的上一个字符串，即"short"。
- lastParen 属性包含捕获组的上一次匹配，即"s"。

这些属性名也可以替换成简写形式，只不过要使用中括号语法来访问，如下面的例子所示，因为大多数简写形式都不是合法的 ECMAScript 标识符：

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

/*
 * 注意：Opera 不支持简写属性名
 * IE 不支持多行匹配
 */
if (pattern.test(text)) {
  console.log(RegExp.$_);       // this has been a short summer
  console.log(RegExp["$`"]);    // this has been a
  console.log(RegExp["$'"]);    // summer
  console.log(RegExp["$&"]);    // short
  console.log(RegExp["$+"]);    // s
}
```

RegExp 还有其他几个构造函数属性，可以存储最多 9 个捕获组的匹配项。这些属性通过 RegExp.$1~RegExp.$9 来访问，分别包含第 1~9 个捕获组的匹配项。在调用 exec()或 test()时，这些属性

就会被填充，然后就可以像下面这样使用它们：

```
let text = "this has been a short summer";
let pattern = /(..)or(.)/g;

if (pattern.test(text)) {
  console.log(RegExp.$1);  // sh
  console.log(RegExp.$2);  // t
}
```

在这个例子中，模式包含两个捕获组。调用 test()搜索字符串之后，因为找到了匹配项所以返回 true，而且可以打印出通过 RegExp 构造函数的$1 和$2 属性取得的两个捕获组匹配的内容。

> **注意** RegExp 构造函数的所有属性都没有任何 Web 标准出处，因此不要在生产环境中使用它们。

5.2.4 模式局限

虽然 ECMAScript 对正则表达式的支持有了长足的进步，但仍然缺少 Perl 语言中的一些高级特性。下列特性目前还没有得到 ECMAScript 的支持（想要了解更多信息，可以参考 Regular-Expressions.info 网站）：

- \A 和\Z 锚（分别匹配字符串的开始和末尾）
- 联合及交叉类
- 原子组
- x（忽略空格）匹配模式
- 条件式匹配
- 正则表达式注释

虽然还有这些局限，但 ECMAScript 的正则表达式已经非常强大，可以用于大多数模式匹配任务。

5.3 原始值包装类型

为了方便操作原始值，ECMAScript 提供了 3 种特殊的引用类型：Boolean、Number 和 String。这些类型具有本章介绍的其他引用类型一样的特点，但也具有与各自原始类型对应的特殊行为。每当用到某个原始值的方法或属性时，后台都会创建一个相应原始包装类型的对象，从而暴露出操作原始值的各种方法。来看下面的例子：

```
let s1 = "some text";
let s2 = s1.substring(2);
```

在这里，s1 是一个包含字符串的变量，它是一个原始值。第二行紧接着在 s1 上调用了 substring()方法，并把结果保存在 s2 中。我们知道，原始值本身不是对象，因此逻辑上不应该有方法。而实际上这个例子又确实按照预期运行了。这是因为后台进行了很多处理，从而实现了上述操作。具体来说，当第二行访问 s1 时，是以读模式访问的，也就是要从内存中读取变量保存的值。在以读模式访问字符串值的任何时候，后台都会执行以下 3 步：

(1) 创建一个 String 类型的实例；

(2) 调用实例上的特定方法；

(3) 销毁实例。

可以把这 3 步想象成执行了如下 3 行 ECMAScript 代码：

```
let s1 = new String("some text");
let s2 = s1.substring(2);
s1 = null;
```

这种行为可以让原始值拥有对象的行为。对布尔值和数值而言，以上 3 步也会在后台发生，只不过使用的是 Boolean 和 Number 包装类型而已。

引用类型与原始值包装类型的主要区别在于对象的生命周期。在通过 new 实例化引用类型后，得到的实例会在离开作用域时被销毁，而自动创建的原始值包装对象则只存在于访问它的那行代码执行期间。这意味着不能在运行时给原始值添加属性和方法。比如下面的例子：

```
let s1 = "some text";
s1.color = "red";
console.log(s1.color);  // undefined
```

这里的第二行代码尝试给字符串 s1 添加了一个 color 属性。可是，第三行代码访问 color 属性时，它却不见了。原因就是第二行代码运行时会临时创建一个 String 对象，而当第三行代码执行时，这个对象已经被销毁了。实际上，第三行代码在这里创建了自己的 String 对象，但这个对象没有 color 属性。

可以显式地使用 Boolean、Number 和 String 构造函数创建原始值包装对象。不过应该在确实必要时再这么做，否则容易让开发者疑惑，分不清它们到底是原始值还是引用值。在原始值包装类型的实例上调用 typeof 会返回"object"，所有原始值包装对象都会转换为布尔值 true。

另外，Object 构造函数作为一个工厂方法，能够根据传入值的类型返回相应原始值包装类型的实例。比如：

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

如果传给 Object 的是字符串，则会创建一个 String 的实例。如果是数值，则会创建 Number 的实例。布尔值则会得到 Boolean 的实例。

注意，使用 new 调用原始值包装类型的构造函数，与调用同名的转型函数并不一样。例如：

```
let value = "25";
let number = Number(value);     // 转型函数
console.log(typeof number);     // "number"
let obj = new Number(value);    // 构造函数
console.log(typeof obj);        // "object"
```

在这个例子中，变量 number 中保存的是一个值为 25 的原始数值，而变量 obj 中保存的是一个 Number 的实例。

虽然不推荐显式创建原始值包装类型的实例，但它们对于操作原始值的功能是很重要的。每个原始值包装类型都有相应的一套方法来方便数据操作。

5.3.1 Boolean

Boolean 是对应布尔值的引用类型。要创建一个 Boolean 对象，就使用 Boolean 构造函数并传入 true 或 false，如下例所示：

```
let booleanObject = new Boolean(true);
```

Boolean 的实例会重写 valueOf()方法，返回一个原始值 true 或 false。toString()方法被调用时也会被覆盖，返回字符串"true"或"false"。不过，Boolean 对象在 ECMAScript 中用得很少。不仅如此，它们还容易引起误会，尤其是在布尔表达式中使用 Boolean 对象时，比如：

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

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

在这段代码中，我们创建一个值为 false 的 Boolean 对象。然后，在一个布尔表达式中通过&&操作将这个对象与一个原始值 true 组合起来。在布尔算术中，false && true 等于 false。可是，这个表达式是对 falseObject 对象而不是对它表示的值（false）求值。前面刚刚说过，所有对象在布尔表达式中都会自动转换为 true，因此 falseObject 在这个表达式里实际上表示一个 true 值。那么 true && true 当然是 true。

除此之外，原始值和引用值（Boolean 对象）还有几个区别。首先，typeof 操作符对原始值返回"boolean"，但对引用值返回"object"。同样，Boolean 对象是 Boolean 类型的实例，在使用 instaceof 操作符时返回 true，但对原始值则返回 false，如下所示：

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

理解原始布尔值和 Boolean 对象之间的区别非常重要，强烈建议永远不要使用后者。

5.3.2 Number

Number 是对应数值的引用类型。要创建一个 Number 对象，就使用 Number 构造函数并传入一个数值，如下例所示：

```
let numberObject = new Number(10);
```

与 Boolean 类型一样，Number 类型重写了 valueOf()、toLocaleString()和 toString()方法。valueOf()方法返回 Number 对象表示的原始数值，另外两个方法返回数值字符串。toString()方法可选地接收一个表示基数的参数，并返回相应基数形式的数值字符串，如下所示：

```
let 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，Number 类型还提供了几个用于将数值格式化为字符串的方法。

toFixed()方法返回包含指定小数点位数的数值字符串，如：

```
let num = 10;
console.log(num.toFixed(2)); // "10.00"
```

这里的 toFixed()方法接收了参数 2，表示返回的数值字符串要包含两位小数。结果返回值为"10.00"，小数位填充了 0。如果数值本身的小数位超过了参数指定的位数，则四舍五入到最接近的