小数位：

```
let num = 10.005;
console.log(num.toFixed(2)); // "10.01"
```

toFixed()自动舍入的特点可以用于处理货币。不过要注意的是，多个浮点数值的数学计算不一定得到精确的结果。比如，0.1 + 0.2 = 0.30000000000000004。

> **注意**　toFixed()方法可以表示有 0~20 个小数位的数值。某些浏览器可能支持更大的范围，但这是通常被支持的范围。

另一个用于格式化数值的方法是 toExponential()，返回以科学记数法（也称为指数记数法）表示的数值字符串。与 toFixed()一样，toExponential()也接收一个参数，表示结果中小数的位数。来看下面的例子：

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

这段代码的输出为"1.0e+1"。一般来说，这么小的数不用表示为科学记数法形式。如果想得到数值最适当的形式，那么可以使用 toPrecision()。

toPrecision()方法会根据情况返回最合理的输出结果，可能是固定长度，也可能是科学记数法形式。这个方法接收一个参数，表示结果中数字的总位数（不包含指数）。来看几个例子：

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

在这个例子中，首先要用 1 位数字表示数值 99，得到"1e+2"，也就是 100。因为 99 不能只用 1 位数字来精确表示，所以这个方法就将它舍入为 100，这样就可以只用 1 位数字（及其科学记数法形式）来表示了。用 2 位数字表示 99 得到"99"，用 3 位数字则是"99.0"。本质上，toPrecision()方法会根据数值和精度来决定调用 toFixed()还是 toExponential()。为了以正确的小数位精确表示数值，这 3 个方法都会向上或向下舍入。

> **注意**　toPrecision()方法可以表示带 1~21 个小数位的数值。某些浏览器可能支持更大的范围，但这是通常被支持的范围。

与 Boolean 对象类似，Number 对象也为数值提供了重要能力。但是，考虑到两者存在同样的潜在问题，因此并不建议直接实例化 Number 对象。在处理原始数值和引用数值时，typeof 和 instacnceof 操作符会返回不同的结果，如下所示：

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

原始数值在调用 typeof 时始终返回"number"，而 Number 对象则返回"object"。类似地，Number 对象是 Number 类型的实例，而原始数值不是。

isInteger()方法与安全整数

ES6 新增了 Number.isInteger()方法，用于辨别一个数值是否保存为整数。有时候，小数位的 0 可能会让人误以为数值是一个浮点值：

```
console.log(Number.isInteger(1));    // true
console.log(Number.isInteger(1.00)); // true
console.log(Number.isInteger(1.01)); // false
```

IEEE 754 数值格式有一个特殊的数值范围，在这个范围内二进制值可以表示一个整数值。这个数值范围从 Number.MIN_SAFE_INTEGER（$-2^{53}+1$）到 Number.MAX_SAFE_INTEGER（$2^{53}-1$）。对超出这个范围的数值，即使尝试保存为整数，IEEE 754 编码格式也意味着二进制值可能会表示一个完全不同的数值。为了鉴别整数是否在这个范围内，可以使用 Number.isSafeInteger()方法：

```
console.log(Number.isSafeInteger(-1 * (2 ** 53)));      // false
console.log(Number.isSafeInteger(-1 * (2 ** 53) + 1));  // true

console.log(Number.isSafeInteger(2 ** 53));             // false
console.log(Number.isSafeInteger((2 ** 53) - 1));       // true
```


5.3.3 String

String 是对应字符串的引用类型。要创建一个 String 对象，使用 String 构造函数并传入一个数值，如下例所示：

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

String 对象的方法可以在所有字符串原始值上调用。3 个继承的方法 valueOf()、toLocaleString()和 toString()都返回对象的原始字符串值。

每个 String 对象都有一个 length 属性，表示字符串中字符的数量。来看下面的例子：

```
let stringValue = "hello world";
console.log(stringValue.length); // "11"
```

这个例子输出了字符串"hello world"中包含的字符数量：11。注意，即使字符串中包含双字节字符（而不是单字节的 ASCII 字符），也仍然会按单字符来计数。

String 类型提供了很多方法来解析和操作字符串。

1. JavaScript 字符

JavaScript 字符串由 16 位码元（code unit）组成。对多数字符来说，每 16 位码元对应一个字符。换句话说，字符串的 length 属性表示字符串包含多少 16 位码元：

```
let message = "abcde";

console.log(message.length); // 5
```

此外，charAt()方法返回给定索引位置的字符，由传给方法的整数参数指定。具体来说，这个方法查找指定索引位置的 16 位码元，并返回该码元对应的字符：

```
let message = "abcde";

console.log(message.charAt(2)); // "c"
```

JavaScript 字符串使用了两种 Unicode 编码混合的策略：UCS-2 和 UTF-16。对于可以采用 16 位编码的字符（U+0000~U+FFFF），这两种编码实际上是一样的。

> **注意**　要深入了解关于字符编码的内容，推荐 Joel Spolsky 写的博客文章：“The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)”。
>
> 另一个有用的资源是 Mathias Bynens 的博文：“JavaScript’s Internal Character Encoding: UCS-2 or UTF-16?”。

使用 charCodeAt()方法可以查看指定码元的字符编码。这个方法返回指定索引位置的码元值，索引以整数指定。比如：

```
let message = "abcde";

// Unicode "Latin small letter C"的编码是U+0063
console.log(message.charCodeAt(2));  // 99

// 十进制 99 等于十六进制 63
console.log(99 === 0x63);            // true
```

fromCharCode()方法用于根据给定的 UTF-16 码元创建字符串中的字符。这个方法可以接受任意多个数值，并返回将所有数值对应的字符拼接起来的字符串：

```
// Unicode "Latin small letter A"的编码是U+0061
// Unicode "Latin small letter B"的编码是U+0062
// Unicode "Latin small letter C"的编码是U+0063
// Unicode "Latin small letter D"的编码是U+0064
// Unicode "Latin small letter E"的编码是U+0065

console.log(String.fromCharCode(0x61, 0x62, 0x63, 0x64, 0x65));  // "abcde"

// 0x0061 === 97
// 0x0062 === 98
// 0x0063 === 99
// 0x0064 === 100
// 0x0065 === 101

console.log(String.fromCharCode(97, 98, 99, 100, 101));          // "abcde"
```

对于 U+0000~U+FFFF 范围内的字符，length、charAt()、charCodeAt()和 fromCharCode()返回的结果都跟预期是一样的。这是因为在这个范围内，每个字符都是用 16 位表示的，而这几个方法也都基于 16 位码元完成操作。只要字符编码大小与码元大小一一对应，这些方法就能如期工作。

这个对应关系在扩展到 Unicode 增补字符平面时就不成立了。问题很简单，即 16 位只能唯一表示 65 536 个字符。这对于大多数语言字符集是足够了，在 Unicode 中称为**基本多语言平面**（BMP）。为了表示更多的字符，Unicode 采用了一个策略，即每个字符使用另外 16 位去选择一个**增补平面**。这种每个字符使用两个 16 位码元的策略称为**代理对**。

在涉及增补平面的字符时，前面讨论的字符串方法就会出问题。比如，下面的例子中使用了一个笑脸表情符号，也就是一个使用代理对编码的字符：

```
// "smiling face with smiling eyes" 表情符号的编码是U+1F60A
// 0x1F60A === 128522
let message = "ab😊de";

console.log(message.length);          // 6
console.log(message.charAt(1));       // b
```

```
console.log(message.charAt(2));       // <?>
console.log(message.charAt(3));       // <?>
console.log(message.charAt(4));       // d

console.log(message.charCodeAt(1));   // 98
console.log(message.charCodeAt(2));   // 55357
console.log(message.charCodeAt(3));   // 56842
console.log(message.charCodeAt(4));   // 100

console.log(String.fromCodePoint(0x1F60A)); // ☺

console.log(String.fromCharCode(97, 98, 55357, 56842, 100, 101)); // ab☺de
```

这些方法仍然将 16 位码元当作一个字符，事实上索引 2 和索引 3 对应的码元应该被看成一个代理对，只对应一个字符。fromCharCode()方法仍然返回正确的结果，因为它实际上是基于提供的二进制表示直接组合成字符串。浏览器可以正确解析代理对（由两个码元构成），并正确地将其识别为一个 Unicode 笑脸字符。

为正确解析既包含单码元字符又包含代理对字符的字符串，可以使用 codePointAt()来代替 charCodeAt()。跟使用 charCodeAt()时类似，codePointAt()接收 16 位码元的索引并返回该索引位置上的码点（code point）。**码点**是 Unicode 中一个字符的完整标识。比如，"c"的码点是 0x0063，而"☺"的码点是 0x1F60A。码点可能是 16 位，也可能是 32 位，而 codePointAt()方法可以从指定码元位置识别完整的码点。

```
let message = "ab☺de";

console.log(message.codePointAt(1)); // 98
console.log(message.codePointAt(2)); // 128522
console.log(message.codePointAt(3)); // 56842
console.log(message.codePointAt(4)); // 100
```

注意，如果传入的码元索引并非代理对的开头，就会返回错误的码点。这种错误只有检测单个字符的时候才会出现，可以通过从左到右按正确的码元数遍历字符串来规避。迭代字符串可以智能地识别代理对的码点：

```
console.log([..."ab☺de"]); // ["a", "b", "☺", "d", "e"]
```

与 charCodeAt()有对应的 codePointAt()一样，fromCharCode()也有一个对应的 fromCodePoint()。这个方法接收任意数量的码点，返回对应字符拼接起来的字符串：

```
console.log(String.fromCharCode(97, 98, 55357, 56842, 100, 101));  // ab☺de
console.log(String.fromCodePoint(97, 98, 128522, 100, 101));       // ab☺de
```

2. normalize()方法

某些 Unicode 字符可以有多种编码方式。有的字符既可以通过一个 BMP 字符表示，也可以通过一个代理对表示。比如：

```
// U+00C5：上面带圆圈的大写拉丁字母 A
console.log(String.fromCharCode(0x00C5));          // Å

// U+212B：长度单位“埃”
console.log(String.fromCharCode(0x212B));          // Å

// U+004：大写拉丁字母 A
// U+030A：上面加个圆圈
console.log(String.fromCharCode(0x0041, 0x030A));  // Å
```

比较操作符不在乎字符看起来是什么样的，因此这 3 个字符互不相等。

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

console.log(a1, a2, a3); // Å, Å, Å

console.log(a1 === a2);  // false
console.log(a1 === a3);  // false
console.log(a2 === a3);  // false
```

为解决这个问题，Unicode 提供了 4 种规范化形式，可以将类似上面的字符规范化为一致的格式，无论底层字符的代码是什么。这 4 种规范化形式是：NFD（Normalization Form D）、NFC（Normalization Form C）、NFKD（Normalization Form KD）和 NFKC（Normalization Form KC）。可以使用 `normalize()` 方法对字符串应用上述规范化形式，使用时需要传入表示哪种形式的字符串：`"NFD"`、`"NFC"`、`"NFKD"` 或 `"NFKC"`。

> **注意**　这 4 种规范化形式的具体细节超出了本书范围，有兴趣的读者可以自行参考 *UAX 15#: Unicode Normalization Forms* 中的 1.2 节“Normalization Forms”。

通过比较字符串与其调用 `normalize()` 的返回值，就可以知道该字符串是否已经规范化了：

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

// U+00C5 是对 0+212B 进行 NFC/NFKC 规范化之后的结果
console.log(a1 === a1.normalize("NFD"));  // false
console.log(a1 === a1.normalize("NFC"));  // true
console.log(a1 === a1.normalize("NFKD")); // false
console.log(a1 === a1.normalize("NFKC")); // true

// U+212B 是未规范化的
console.log(a2 === a2.normalize("NFD"));  // false
console.log(a2 === a2.normalize("NFC"));  // false
console.log(a2 === a2.normalize("NFKD")); // false
console.log(a2 === a2.normalize("NFKC")); // false

// U+0041/U+030A 是对 0+212B 进行 NFD/NFKD 规范化之后的结果
console.log(a3 === a3.normalize("NFD"));  // true
console.log(a3 === a3.normalize("NFC"));  // false
console.log(a3 === a3.normalize("NFKD")); // true
console.log(a3 === a3.normalize("NFKC")); // false
```

选择同一种规范化形式可以让比较操作符返回正确的结果：

```
let a1 = String.fromCharCode(0x00C5),
    a2 = String.fromCharCode(0x212B),
    a3 = String.fromCharCode(0x0041, 0x030A);

console.log(a1.normalize("NFD") === a2.normalize("NFD"));    // true
console.log(a2.normalize("NFKC") === a3.normalize("NFKC")); // true
console.log(a1.normalize("NFC") === a3.normalize("NFC"));    // true
```

3. 字符串操作方法

本节介绍几个操作字符串值的方法。首先是 `concat()`，用于将一个或多个字符串拼接成一个新字

符串。来看下面的例子：

```
let stringValue = "hello ";
let result = stringValue.concat("world");

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

在这个例子中，对 stringValue 调用 concat()方法的结果是得到"hello world"，但 stringValue 的值保持不变。concat()方法可以接收任意多个参数，因此可以一次性拼接多个字符串，如下所示：

```
let stringValue = "hello ";
let result = stringValue.concat("world", "!");

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

这个修改后的例子将字符串"world"和"!"追加到了"hello "后面。虽然 concat()方法可以拼接字符串，但更常用的方式是使用加号操作符（+）。而且多数情况下，对于拼接多个字符串来说，使用加号更方便。

ECMAScript 提供了 3 个从字符串中提取子字符串的方法：slice()、substr()和 substring()。这 3 个方法都返回调用它们的字符串的一个子字符串，而且都接收一或两个参数。第一个参数表示子字符串开始的位置，第二个参数表示子字符串结束的位置。对 slice()和 substring()而言，第二个参数是提取结束的位置（即该位置之前的字符会被提取出来）。对 substr()而言，第二个参数表示返回的子字符串数量。任何情况下，省略第二个参数都意味着提取到字符串末尾。与 concat()方法一样，slice()、substr()和 substring()也不会修改调用它们的字符串，而只会返回提取到的原始新字符串值。来看下面的例子：

```
let 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"
```

在这个例子中，slice()、substr()和 substring()是以相同方式被调用的，而且多数情况下返回的值也相同。如果只传一个参数 3，则所有方法都将返回"lo world"，因为"hello"中"l"位置为 3。如果传入两个参数 3 和 7，则 slice()和 substring()返回"lo w"（因为"world"中"o"在位置 7，不包含），而 substr()返回"lo worl"，因为第二个参数对它而言表示返回的字符数。

当某个参数是负值时，这 3 个方法的行为又有不同。比如，slice()方法将所有负值参数都当成字符串长度加上负参数值。

而 substr()方法将第一个负参数值当成字符串长度加上该值，将第二个负参数值转换为 0。substring()方法会将所有负参数值都转换为 0。看下面的例子：

```
let 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));   // "" (empty string)
```

这个例子明确演示了 3 个方法的差异。在给 slice()和 substr()传入负参数时，它们的返回结果相同。这是因为-3 会被转换为 8（长度加上负参数），实际上调用的是 slice(8)和 substr(8)。而 substring()方法返回整个字符串，因为-3 会转换为 0。

在第二个参数是负值时，这 3 个方法各不相同。slice()方法将第二个参数转换为 7，实际上相当于调用 slice(3, 7)，因此返回"lo w"。而 substring()方法会将第二个参数转换为 0，相当于调用 substring(3, 0)，等价于 substring(0, 3)，这是因为这个方法会将较小的参数作为起点，将较大的参数作为终点。对 substr()来说，第二个参数会被转换为 0，意味着返回的字符串包含零个字符，因而会返回一个空字符串。

4. 字符串位置方法

有两个方法用于在字符串中定位子字符串：indexOf()和 lastIndexOf()。这两个方法从字符串中搜索传入的字符串，并返回位置（如果没找到，则返回-1）。两者的区别在于，indexOf()方法从字符串开头开始查找子字符串，而 lastIndexOf()方法从字符串末尾开始查找子字符串。来看下面的例子：

```
let stringValue = "hello world";
console.log(stringValue.indexOf("o"));     // 4
console.log(stringValue.lastIndexOf("o")); // 7
```

这里，字符串中第一个"o"的位置是 4，即"hello"中的"o"。最后一个"o"的位置是 7，即"world"中的"o"。如果字符串中只有一个"o"，则 indexOf()和 lastIndexOf()返回同一个位置。

这两个方法都可以接收可选的第二个参数，表示开始搜索的位置。这意味着，indexOf()会从这个参数指定的位置开始向字符串末尾搜索，忽略该位置之前的字符；lastIndexOf()则会从这个参数指定的位置开始向字符串开头搜索，忽略该位置之后直到字符串末尾的字符。下面看一个例子：

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

在传入第二个参数 6 以后，结果跟前面的例子恰好相反。这一次，indexOf()返回 7，因为它从位置 6（字符"w"）开始向后搜索字符串，在位置 7 找到了"o"。而 lastIndexOf()返回 4，因为它从位置 6 开始反向搜索至字符串开头，因此找到了"hello"中的"o"。像这样使用第二个参数并循环调用 indexOf()或 lastIndexOf()，就可以在字符串中找到所有的目标子字符串，如下所示：

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

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

console.log(positions); // [3,24,32,35,52]
```

这个例子逐步增大开始搜索的位置，通过 indexOf()遍历了整个字符串。首先取得第一个"e"的位置，然后进入循环，将上一次的位置加 1 再传给 indexOf()，确保搜索到最后一个子字符串实例之后。每个位置都保存在 positions 数组中，可供以后使用。

5. 字符串包含方法

ECMAScript 6 增加了 3 个用于判断字符串中是否包含另一个字符串的方法：startsWith()、endsWith()和 includes()。这些方法都会从字符串中搜索传入的字符串，并返回一个表示是否包含的布尔值。它们的区别在于，startsWith()检查开始于索引 0 的匹配项，endsWith()检查开始于索引(string.length - substring.length)的匹配项，而 includes()检查整个字符串：

```
let message = "foobarbaz";

console.log(message.startsWith("foo"));  // true
console.log(message.startsWith("bar"));  // false

console.log(message.endsWith("baz"));    // true
console.log(message.endsWith("bar"));    // false

console.log(message.includes("bar"));    // true
console.log(message.includes("qux"));    // false
```

startsWith()和 includes()方法接收可选的第二个参数，表示开始搜索的位置。如果传入第二个参数，则意味着这两个方法会从指定位置向着字符串末尾搜索，忽略该位置之前的所有字符。下面是一个例子：

```
let message = "foobarbaz";

console.log(message.startsWith("foo"));     // true
console.log(message.startsWith("foo", 1));  // false

console.log(message.includes("bar"));       // true
console.log(message.includes("bar", 4));    // false
```

endsWith()方法接收可选的第二个参数，表示应该当作字符串末尾的位置。如果不提供这个参数，那么默认就是字符串长度。如果提供这个参数，那么就好像字符串只有那么多字符一样：

```
let message = "foobarbaz";

console.log(message.endsWith("bar"));     // false
console.log(message.endsWith("bar", 6));  // true
```

6. trim()方法

ECMAScript 在所有字符串上都提供了 trim()方法。这个方法会创建字符串的一个副本，删除前、后所有空格符，再返回结果。比如：

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

由于 trim()返回的是字符串的副本，因此原始字符串不受影响，即原本的前、后空格符都会保留。另外，trimLeft()和 trimRight()方法分别用于从字符串开始和末尾清理空格符。

7. repeat()方法

ECMAScript 在所有字符串上都提供了 repeat()方法。这个方法接收一个整数参数，表示要将字符串复制多少次，然后返回拼接所有副本后的结果。

```
let stringValue = "na ";
console.log(stringValue.repeat(16) + "batman");
// na na na na na na na na na na na na na na na na batman
```

8. padStart()和 padEnd()方法

padStart()和 padEnd()方法会复制字符串，如果小于指定长度，则在相应一边填充字符，直至满足长度条件。这两个方法的第一个参数是长度，第二个参数是可选的填充字符串，默认为空格（U+0020）。

```
let stringValue = "foo";

console.log(stringValue.padStart(6));        // "   foo"
console.log(stringValue.padStart(9, "."));  // "......foo"

console.log(stringValue.padEnd(6));          // "foo   "
console.log(stringValue.padEnd(9, "."));    // "foo......"
```

可选的第二个参数并不限于一个字符。如果提供了多个字符的字符串，则会将其拼接并截断以匹配指定长度。此外，如果长度小于或等于字符串长度，则会返回原始字符串。

```
let stringValue = "foo";

console.log(stringValue.padStart(8, "bar")); // "barbafoo"
console.log(stringValue.padStart(2));        // "foo"

console.log(stringValue.padEnd(8, "bar"));   // "foobarba"
console.log(stringValue.padEnd(2));          // "foo"
```

9. 字符串迭代与解构

字符串的原型上暴露了一个@@iterator 方法，表示可以迭代字符串的每个字符。可以像下面这样手动使用迭代器：

```
let message = "abc";
let stringIterator = message[Symbol.iterator]();

console.log(stringIterator.next());  // {value: "a", done: false}
console.log(stringIterator.next());  // {value: "b", done: false}
console.log(stringIterator.next());  // {value: "c", done: false}
console.log(stringIterator.next());  // {value: undefined, done: true}
```

在 for-of 循环中可以通过这个迭代器按序访问每个字符：

```
for (const c of "abcde") {
  console.log(c);
}

// a
// b
// c
// d
// e
```

有了这个迭代器之后，字符串就可以通过解构操作符来解构了。比如，可以更方便地把字符串分割为字符数组：

```
let message = "abcde";

console.log([...message]); // ["a", "b", "c", "d", "e"]
```

10. 字符串大小写转换

下一组方法涉及大小写转换，包括 4 个方法：toLowerCase()、toLocaleLowerCase()、

toUpperCase()和toLocaleUpperCase()。toLowerCase()和toUpperCase()方法是原来就有的方法，与 java.lang.String 中的方法同名。toLocaleLowerCase()和 toLocaleUpperCase()方法旨在基于特定地区实现。在很多地区，地区特定的方法与通用的方法是一样的。但在少数语言中（如土耳其语），Unicode 大小写转换需应用特殊规则，要使用地区特定的方法才能实现正确转换。下面是几个例子：

```
let 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"
```

这里，toLowerCase()和 toLocaleLowerCase()都返回 hello world，而 toUpperCase()和 toLocaleUpperCase()都返回 HELLO WORLD。通常，如果不知道代码涉及什么语言，则最好使用地区特定的转换方法。

11. 字符串模式匹配方法

String 类型专门为在字符串中实现模式匹配设计了几个方法。第一个就是 match()方法，这个方法本质上跟 RegExp 对象的 exec()方法相同。match()方法接收一个参数，可以是一个正则表达式字符串，也可以是一个 RegExp 对象。来看下面的例子：

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

// 等价于 pattern.exec(text)
let matches = text.match(pattern);
console.log(matches.index);      // 0
console.log(matches[0]);         // "cat"
console.log(pattern.lastIndex);  // 0
```

match()方法返回的数组与 RegExp 对象的 exec()方法返回的数组是一样的：第一个元素是与整个模式匹配的字符串，其余元素则是与表达式中的捕获组匹配的字符串（如果有的话）。

另一个查找模式的字符串方法是 search()。这个方法唯一的参数与 match()方法一样：正则表达式字符串或 RegExp 对象。这个方法返回模式第一个匹配的位置索引，如果没找到则返回–1。search()始终从字符串开头向后匹配模式。看下面的例子：

```
let text = "cat, bat, sat, fat";
let pos = text.search(/at/);
console.log(pos);  // 1
```

这里，search(/at/)返回 1，即"at"的第一个字符在字符串中的位置。

为简化子字符串替换操作，ECMAScript 提供了 replace()方法。这个方法接收两个参数，第一个参数可以是一个 RegExp 对象或一个字符串（这个字符串不会转换为正则表达式），第二个参数可以是一个字符串或一个函数。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，第一个参数必须为正则表达式并且带全局标记，如下面的例子所示：

```
let text = "cat, bat, sat, fat";
let 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"
```

在这个例子中，字符串"at"先传给 replace()函数，而替换文本是"ond"。结果是"cat"被修改