字符串的新增方法

目录

- String.fromCodePoint()
- String.raw()
- 实例方法: codePointAt()
- 实例方法: normalize()
- 实例方法: includes(), startsWith(), endsWith()
- 实例方法: repeat()
- 实例方法: padStart()，padEnd()
- 实例方法: trimStart()，trimEnd()
- 实例方法: matchAll()
- 实例方法: replaceAll()
- 实例方法: at()
- 实例方法: toWellFormed()

本章介绍字符串对象的新增方法。

String.fromCodePoint()

ES5 提供 `String.fromCharCode()` 方法，用于从 Unicode 码点返回对应字符，但是这个方法不能识别码点大于 `0xFFFF` 的字符。

```
String.fromCharCode(0x20BB7)
// "𠮷"
```

上面代码中，`String.fromCharCode()` 不能识别大于 `0xFFFF` 的码点，所以 `0x20BB7` 就发生了溢出，最高位 `2` 被舍弃了，最后返回码点 `U+0BB7` 对应的字符，而不是码点 `U+20BB7` 对应的字符。

ES6 提供了 `String.fromCodePoint()` 方法，可以识别大于 `0xFFFF` 的字符，弥补了 `String.fromCharCode()` 方法的不足。在作用上，正好与下面的 `codePointAt()` 方法相反。

```
String.fromCodePoint(0x20BB7)
// "𠮷"
String.fromCodePoint(0x78, 0x1f680, 0x79) === 'x\uD83D\uDE80y'
// true
```

上面代码中，如果 `String.fromCodePoint` 方法有多个参数，则它们会被合并成一个字符串返回。

注意， `fromCodePoint` 方法定义在 `String` 对象上，而 `codePointAt` 方法定义在字符串的实例对象上。

String.raw()

ES6 还为原生的 String 对象，提供了一个 `raw()` 方法。该方法返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，往往用于模板字符串的处理方法。

```
String.raw`Hi\n${2+3}!`
// 实际返回 "Hi\\n5!"，显示的是转义后的结果 "Hi\n5!"

String.raw`Hi\u000A!`;
// 实际返回 "Hi\\u000A!"，显示的是转义后的结果 "Hi\u000A!"
```

如果原字符串的斜杠已经转义，那么 `String.raw()` 会进行再次转义。

```
String.raw`Hi\\n`
// 返回 "Hi\\\\n"

String.raw`Hi\\n` === "Hi\\\\n" // true
```

`String.raw()` 方法可以作为处理模板字符串的基本方法，它会将所有变量替换，而且对斜杠进行转义，方便下一步作为字符串来使用。

`String.raw()` 本质上是一个正常的函数，只是专用于模板字符串的标签函数。如果写成正常函数的形式，它的第一个参数，应该是一个具有 `raw` 属性的对象，且 `raw` 属性的值应该是一个数组，对应模板字符串解析后的值。

```
// `foo${1 + 2}bar`
// 等同于
String.raw({ raw: ['foo', 'bar'] }, 1 + 2) // "foo3bar"
```

上面代码中，`String.raw()`方法的第一个参数是一个对象，它的`raw`属性等同于原始的模板字符串解析后得到的数组。

作为函数，`String.raw()`的代码实现基本如下。

```
String.raw = function (strings, ...values) {
  let output = '';
  let index;
  for (index = 0; index < values.length; index++) {
    output += strings.raw[index] + values[index];
  }

  output += strings.raw[index]
  return output;
}
```

实例方法：codePointAt()

JavaScript 内部，字符以 UTF-16 的格式储存，每个字符固定为`2`个字节。对于那些需要`4`个字节储存的字符（Unicode 码点大于`0xFFFF`的字符），JavaScript 会认为它们是两个字符。

```
var s = "□□";

s.length // 2
s.charAt(0) // ''
s.charAt(1) // ''
s.charCodeAt(0) // 55362
s.charCodeAt(1) // 57271
```

上面代码中，汉字“□□”（注意，这个字不是“吉祥”的“吉”）的码点是`0x20BB7`，UTF-16 编码为`0xD842 0xDFB7`（十进制为`55362 57271`），需要`4`个字节储存。对于这种`4`个字节的字符，JavaScript 不能正确处理，字符串长度会误判为`2`，而且`charAt()`方法无法读取整个字符，`charCodeAt()`方法只能分别返回前两个字节和后两个字节的值。

ES6 提供了`codePointAt()`方法，能够正确处理`4`个字节储存的字符，返回一个字符的码点。

```
let s = '□a';

s.codePointAt(0) // 134071
s.codePointAt(1) // 57271
```

```
s.codePointAt(2) // 97
```

codePointAt() 0 JavaScript “a” codePointAt “” 134071 20BB7 “”“a” codePointAt() charCodeAt()

codePointAt() 32 UTF-16 charCodeAt()

codePointAt() toString()

```
let s = '';

s.codePointAt(0).toString(16) // "20bb7"
s.codePointAt(2).toString(16) // "61"
```

codePointAt() a s 1 codePointAt() 2 for...of 32 UTF-16

```
let s = '';
for (let ch of s) {
  console.log(ch.codePointAt(0).toString(16));
}
// 20bb7
// 61
```

...

```
let arr = [...''']; // arr.length === 2
arr.forEach(
  ch => console.log(ch.codePointAt(0).toString(16))
);
// 20bb7
// 61
```

codePointAt()

```
function is32Bit(c) {
  return c.codePointAt(0) > 0xFFFF;
```

```
}

is32Bit("□□") // true
is32Bit("a") // false
```

实例方法: normalize()

许多欧洲语言有语调符号和重音符号。为了表示它们，Unicode 提供了两种方法。一种是直接提供带重音符号的字符，比如 `Ǒ`（\u01D1）。另一种是提供合成符号（combining character），即原字符与重音符号的合成，两个字符合成一个字符，比如 `O`（\u004F）和 `ˇ`（\u030C）合成 `Ǒ`（\u004F\u030C）。

这两种表示方法，在视觉和语义上都等价，但是 JavaScript 不能识别。

```
'\u01D1'==='\u004F\u030C' //false

'\u01D1'.length // 1
'\u004F\u030C'.length // 2
```

上面代码表示，JavaScript 将合成字符视为两个字符，导致两种表示方法不相等。

ES6 提供字符串实例的 `normalize()` 方法，用来将字符的不同表示方法统一为同样的形式，这称为 Unicode 正规化。

```
'\u01D1'.normalize() === '\u004F\u030C'.normalize()
// true
```

`normalize` 方法可以接受一个参数来指定 `normalize` 的方式，参数的四个可选值如下。

- `NFC`，默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
- `NFD`，表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
- `NFKC`，表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。（这只是用来举例，`normalize` 方法不能识别中文。）
- `NFKD`，表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

```
'\u004F\u030C'.normalize('NFC').length // 1
'\u004F\u030C'.normalize('NFD').length // 2
```

上面代码表示，`NFC`参数返回字符的合成形式，`NFD`参数返回字符的分解形式。

不过，`normalize`方法目前不能识别三个或三个以上字符的合成。这种情况下，还是只能使用正则表达式，通过Unicode编号区间判断。

实例方法：includes(), startsWith(), endsWith()

传统上，JavaScript 只有`indexOf`方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6又提供了三种新方法。

- **includes()**：返回布尔值，表示是否找到了参数字符串。
- **startsWith()**：返回布尔值，表示参数字符串是否在原字符串的头部。
- **endsWith()**：返回布尔值，表示参数字符串是否在原字符串的尾部。

```
let s = 'Hello world!';

s.startsWith('Hello') // true
s.endsWith('!') // true
s.includes('o') // true
```

这三个方法都支持第二个参数，表示开始搜索的位置。

```
let s = 'Hello world!';

s.startsWith('world', 6) // true
s.endsWith('Hello', 5) // true
s.includes('Hello', 6) // false
```

上面代码表示，使用第二个参数`n`时，`endsWith`的行为与其他两个方法有所不同。它针对前`n`个字符，而其他两个方法针对从第`n`个位置直到字符串结束。

实例方法：repeat()

`repeat`方法返回一个新字符串，表示将原字符串重复`n`次。

```
'x'.repeat(3) // "xxx"
'hello'.repeat(2) // "hellohello"
'na'.repeat(0) // ""
```

参数如果是小数，会被取整。

```
'na'.repeat(2.9) // "nana"
```

如果 `repeat` 的参数是负数或者 `Infinity`，会报错。

```
'na'.repeat(Infinity)
// RangeError
'na'.repeat(-1)
// RangeError
```

但是，如果参数是 0 到 -1 之间的小数，则等同于 0，这是因为会先进行取整运算。0 到 -1 之间的小数，取整以后等于 `-0`，`repeat` 视同为 0。

```
'na'.repeat(-0.9) // ""
```

参数 `NaN` 等同于 0。

```
'na'.repeat(NaN) // ""
```

如果 `repeat` 的参数是字符串，则会先转换成数字。

```
'na'.repeat('na') // ""
'na'.repeat('3') // "nanana"
```

实例方法：padStart()，padEnd()

ES2017 引入了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。`padStart()` 用于头部补全，`padEnd()` 用于尾部补全。

```
'x'.padStart(5, 'ab') // 'ababx'
'x'.padStart(4, 'ab') // 'abax'

'x'.padEnd(5, 'ab') // 'xabab'
'x'.padEnd(4, 'ab') // 'xaba'
```

上面代码中，`padStart()`和`padEnd()`一共接受两个参数，第一个参数是字符串补全生效的最大长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串。

```
'xxx'.padStart(2, 'ab') // 'xxx'
'xxx'.padEnd(2, 'ab') // 'xxx'
```

如果用来补全的字符串与原字符串，两者的长度之和超过了最大长度，则会截去超出位数的补全字符串。

```
'abc'.padStart(10, '0123456789')
// '0123456abc'
```

如果省略第二个参数，默认使用空格补全长度。

```
'x'.padStart(4) // '   x'
'x'.padEnd(4) // 'x   '
```

`padStart()`的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。

```
'1'.padStart(10, '0') // "0000000001"
'12'.padStart(10, '0') // "0000000012"
'123456'.padStart(10, '0') // "0000123456"
```

另一个用途是提示字符串格式。

```
'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"
'09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"
```

实例方法：trimStart()，trimEnd()

ES2019 对字符串实例新增了`trimStart()`和`trimEnd()`这两个方法。它们的行为与`trim()`一致，`trimStart()`消除字符串头部的空格，`trimEnd()`消除尾部的空格。它们返回的都是新字符串，不会修改原始字符串。

```
const s = '  abc  ';

s.trim() // "abc"
s.trimStart() // "abc  "
s.trimEnd() // "  abc"
```

上面代码中，`trimStart()`只消除头部的空格，保留尾部的空格。`trimEnd()`也是类似行为。

除了空格键，这两个方法对字符串头部（或尾部）的 tab 键、换行符等不可见的空白符号也有效。

浏览器还部署了额外的两个方法，`trimLeft()`是`trimStart()`的别名，`trimRight()`是`trimEnd()`的别名。

实例方法：matchAll()

`matchAll()`方法返回一个正则表达式在当前字符串的所有匹配，详见《正则的扩展》的一章。

实例方法：replaceAll()

历史上，字符串的实例方法`replace()`只能替换第一个匹配。

```
'aabbcc'.replace('b', '_')
// 'aa_bcc'
```

上面例子中，`replace()`只将第一个`b`替换成了下划线。

如果要替换所有的匹配，不得不使用正则表达式的`g`修饰符。

```
'aabbcc'.replace(/b/g, '_')
// 'aa__cc'
```

正则表达式毕竟不是那么方便和直观，ES2021 引入了`replaceAll()`方法，可以一次性替换所有匹配。

```
'aabbcc'.replaceAll('b', '_')
// 'aa__cc'
```

它的用法与`replace()`相同，返回一个新字符串，不会改变原字符串。

```
String.prototype.replaceAll(searchValue, replacement)
```

上面代码中，`searchValue`是搜索模式，可以是一个字符串，也可以是一个全局的正则表达式（带有`g`修饰符）。

如果`searchValue`是一个不带有`g`修饰符的正则表达式，`replaceAll()`会报错。这一点跟`replace()`不同。

```
// 不报错
'aabbcc'.replace(/b/, '_')

// 报错
'aabbcc'.replaceAll(/b/, '_')
```

上面例子中，`/b/`不带有`g`修饰符，会导致`replaceAll()`报错。

`replaceAll()`的第二个参数`replacement`是一个字符串，表示替换的文本，其中可以使用一些特殊字符串。

- `$&`：匹配的字符串。
- `$`` `：匹配结果前面的文本。
- `$'`：匹配结果后面的文本。
- `$n`：匹配成功的第`n`组内容，`n`是从`1`开始的自然数。这个参数生效的前提是，第一个参数必须是正则表达式。
- `$$`：指代美元符号`$`。

下面是一些例子。

```
// $& 表示匹配的字符串，即`b`本身
// 所以返回结果与原字符串一致
'abbc'.replaceAll('b', '$&')
// 'abbc'

// $` 表示匹配结果之前的字符串
// 对于第一个`b`，$` 指代`a`
// 对于第二个`b`，$` 指代`ab`
'abbc'.replaceAll('b', '$`')
// 'aaabc'

// $' 表示匹配结果之后的字符串
// 对于第一个`b`，$' 指代`bc`
// 对于第二个`b`，$' 指代`c`
'abbc'.replaceAll('b', `$'`)
// 'abccc'

// $1 表示正则表达式的第一个组匹配，指代`ab`
// $2 表示正则表达式的第二个组匹配，指代`bc`
'abbc'.replaceAll(/(ab)(bc)/g, '$2$1')
// 'bcab'

// $$ 指代 $
```

```
'abc'.replaceAll('b', '$$')
// 'a$c'
```

`replaceAll()`的第二个参数`replacement`除了为字符串，也可以是一个函数，该函数的返回值将替换掉第一个参数`searchValue`匹配的文本。

```
'aabbcc'.replaceAll('b', () => '_')
// 'aa__cc'
```

上面例子中，`replaceAll()`的第二个参数是一个函数，该函数的返回值会替换掉所有`b`的匹配。

这个替换函数可以接受多个参数。第一个参数是捕捉到的匹配内容，第二个参数是捕捉到的组匹配（有多少个组匹配，就有多少个对应的参数）。此外，最后还可以添加两个参数，倒数第二个参数是捕捉到的内容在整个字符串中的位置，最后一个参数是原字符串。

```
const str = '123abc456';
const regex = /(\d+)([a-z]+)(\d+)/g;

function replacer(match, p1, p2, p3, offset, string) {
  return [p1, p2, p3].join(' - ');
}

str.replaceAll(regex, replacer)
// 123 - abc - 456
```

上面例子中，正则表达式有三个组匹配，所以`replacer()`函数的第一个参数`match`是捕捉到的匹配内容（即字符串`123abc456`），后面三个参数`p1`、`p2`、`p3`则依次为三个组匹配。

实例方法：at()

`at()`方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

```
const str = 'hello';
str.at(1) // "e"
str.at(-1) // "o"
```

如果参数位置超出了字符串范围，`at()`返回`undefined`。

该方法来自数组添加的`at()`方法，目前还是一个第三阶段的提案，可以参考《数组》一章的介绍。

实例方法: toWellFormed()

ES2024 引入了新的字符串方法 `toWellFormed()`，用来处理 Unicode 的代理字符对问题（surrogates）。

JavaScript 语言内部使用 UTF-16 格式，表示每个字符。UTF-16 只有 16位，只能表示码点在 `U+0000` 到 `U+FFFF` 之间的 Unicode 字符。对于码点大于 `U+FFFF` 的 Unicode 字符（即码点大于 16位的字符，`U+10000` 到 `U+10FFFF`），解决办法是使用代理字符对，即用两个 UTF-16 字符组合表示。

具体来说，UTF-16 规定，`U+D800` 至 `U+DFFF` 是空字符段，专门留给代理字符对使用。只要遇到这个范围内的码点，就知道它是代理字符对，本身没有意义，必须两个字符结合在一起解读。其中，前一个字符的范围规定为 `0xD800` 到 `0xDBFF` 之间，后一个字符的范围规定为 `0xDC00` 到 `0xDFFF` 之间。举例来说，码点 `U+1D306` 对应的字符为 `𝌆`，它写成 UTF-16 就是 `0xD834 0xDF06`。

但是，字符串里面可能会出现单个代理字符对，即 `U+D800` 至 `U+DFFF` 里面的字符，它没有配对的另一个字符，无法进行解读，导致出现各种状况。

`.toWellFormed()` 就是为了解决这个问题，不改变原始字符串，返回一个新的字符串，将原始字符串里面的单个代理字符对，都替换为 `U+FFFD`，从而可以在任何正常处理字符串的函数里面使用。

```
"ab\uD800".toWellFormed() // 'ab�'
```

上面示例中，`\uD800` 是单个的代理字符对，单独使用时没有意义。`toWellFormed()` 将这个字符转为 `\uFFFD`。

再看下面的例子，`encodeURI()` 遇到单个的代理字符对，会报错。

```
const illFormed = "https://example.com/search?q=\uD800";

encodeURI(illFormed) // 报错
```

`toWellFormed()` 将其转换格式后，再使用 `encodeURI()` 就不会报错了。

```
const illFormed = "https://example.com/search?q=\uD800";

encodeURI(illFormed.toWellFormed()) // 正确
```

版本号 #2
由 壳壳虫 创建于 2020-04-16 10:29:52 UTC
由 壳壳虫 更新于 2024-12-31 16:24:06 UTC