# 字符串的扩展

1. **[字符的 Unicode 表示法](http://es6.ruanyifeng.com/" \l "docs/string#字符的 Unicode 表示法)**
2. **[codePointAt()](http://es6.ruanyifeng.com/" \l "docs/string#codePointAt())**
3. **[String.fromCodePoint()](http://es6.ruanyifeng.com/" \l "docs/string#String.fromCodePoint())**
4. **[字符串的遍历器接口](http://es6.ruanyifeng.com/" \l "docs/string#字符串的遍历器接口)**
5. **[at()](http://es6.ruanyifeng.com/" \l "docs/string#at())**
6. **[normalize()](http://es6.ruanyifeng.com/" \l "docs/string#normalize())**
7. **[includes(), startsWith(), endsWith()](http://es6.ruanyifeng.com/" \l "docs/string#includes(), startsWith(), endsWith())**
8. **[repeat()](http://es6.ruanyifeng.com/" \l "docs/string#repeat())**
9. **[padStart()，padEnd()](http://es6.ruanyifeng.com/" \l "docs/string#padStart()，padEnd())**
10. **[模板字符串](http://es6.ruanyifeng.com/" \l "docs/string#模板字符串)**
11. **[实例：模板编译](http://es6.ruanyifeng.com/" \l "docs/string#实例：模板编译)**
12. **[标签模板](http://es6.ruanyifeng.com/" \l "docs/string#标签模板)**
13. **[String.raw()](http://es6.ruanyifeng.com/" \l "docs/string#String.raw())**
14. **[模板字符串的限制](http://es6.ruanyifeng.com/" \l "docs/string#模板字符串的限制)**

ES6 加强了对 Unicode 的支持，并且扩展了字符串对象。

## 字符的 Unicode 表示法

JavaScript 允许采用\uxxxx形式表示一个字符，其中xxxx表示字符的 Unicode 码点。

"\u0061"

// "a"

但是，这种表示法只限于码点在\u0000~\uFFFF之间的字符。超出这个范围的字符，必须用两个双字节的形式表示。

"\uD842\uDFB7"

// ""\u20BB7"

// " 7"

上面代码表示，如果直接在\u后面跟上超过0xFFFF的数值（比如\u20BB7），JavaScript会理解成\u20BB+7。由于\u20BB是一个不可打印字符，所以只会显示一个空格，后面跟着一个7。

ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符。

"\u{20BB7}"

// ""\u{41}\u{42}\u{43}"

// "ABC"let hello = 123;

hell\u{6F} // 123'\u{1F680}' === '\uD83D\uDE80'

// true

上面代码中，最后一个例子表明，大括号表示法与四字节的 UTF-16 编码是等价的。

有了这种表示法之后，JavaScript 共有6种方法可以表示一个字符。

'\z' === 'z' // true'\172' === 'z' // true'\x7A' === 'z' // true'\u007A' === 'z' // true'\u{7A}' === 'z' // true

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

上面代码中，汉字“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 = ';

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

s.codePointAt(2) // 97

codePointAt方法的参数，是字符在字符串中的位置（从0开始）。上面代码中，JavaScript将“20BB7）。在第二个字符（即“codePointAt方法的结果与charCodeAt方法相同。

总之，codePointAt方法会正确返回32位的UTF-16字符的码点。对于那些两个字节储存的常规字符，它的返回结果与charCodeAt方法相同。

codePointAt方法返回的是码点的十进制值，如果想要十六进制的值，可以使用toString方法转换一下。

let s = ';

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

你可能注意到了，codePointAt方法的参数，仍然是不正确的。比如，上面代码中，字符a在字符串s的正确位置序号应该是1，但是必须向codePointAt方法传入2。解决这个问题的一个办法是使用for...of循环，因为它会正确识别32位的UTF-16字符。

let s = ';for (let ch of s) {

console.log(ch.codePointAt(0).toString(16));}

// 20bb7// 61

codePointAt方法是测试一个字符由两个字节还是由四个字节组成的最简单方法。

function is32Bit(c) {

return c.codePointAt(0) > 0xFFFF;}

is32Bit(") // trueis32Bit("a") // false

## String.fromCodePoint()

ES5提供String.fromCharCode方法，用于从码点返回对应字符，但是这个方法不能识别32位的UTF-16字符（Unicode编号大于0xFFFF）。

String.fromCharCode(0x20BB7)

// "ஷ"

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codePointAt方法相反。

String.fromCodePoint(0x20BB7)

// "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。

## 字符串的遍历器接口

ES6为字符串添加了遍历器接口（详见《Iterator》一章），使得字符串可以被for...of循环遍历。

for (let codePoint of 'foo') {

console.log(codePoint)}

// "f"// "o"// "o"

除了遍历字符串，这个遍历器最大的优点是可以识别大于0xFFFF的码点，传统的for循环无法识别这样的码点。

let text = String.fromCodePoint(0x20BB7);

for (let i = 0; i < text.length; i++) {

console.log(text[i]);}

// " "// " "for (let i of text) {

console.log(i);}

// "

上面代码中，字符串text只有一个字符，但是for循环会认为它包含两个字符（都不可打印），而for...of循环会正确识别出这一个字符。

## at()

ES5 对字符串对象提供charAt方法，返回字符串给定位置的字符。该方法不能识别码点大于0xFFFF的字符。

'abc'.charAt(0) // "a"'.charAt(0) // "\uD842"

上面代码中，charAt方法返回的是UTF-16编码的第一个字节，实际上是无法显示的。

目前，有一个提案，提出字符串实例的at方法，可以识别 Unicode 编号大于0xFFFF的字符，返回正确的字符。

'abc'.at(0) // "a"'.at(0) // "

这个方法可以通过[垫片库](https://github.com/es-shims/String.prototype.at)实现。

## 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。

## 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') // trues.endsWith('!') // trues.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) // trues.endsWith('Hello', 5) // trues.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。

## 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"

## 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，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于指定的最小长度，则返回原字符串。

'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指定的最小长度，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"

## 模板字符串

传统的JavaScript语言，输出模板通常是这样写的。

$('#result').append(

'There are <b>' + basket.count + '</b> ' +

'items in your basket, ' +

'<em>' + basket.onSale +

'</em> are on sale!');

上面这种写法相当繁琐不方便，ES6引入了模板字符串解决这个问题。

$('#result').append(`

There are <b>${basket.count}</b> items

in your basket, <em>${basket.onSale}</em>

are on sale!

`);

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

// 普通字符串`In JavaScript '\n' is a line-feed.`

// 多行字符串`In JavaScript this is

not legal.`

console.log(`string text line 1

string text line 2`);

// 字符串中嵌入变量let name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

let greeting = `\`Yo\` World!`;

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

$('#list').html(`<ul>

<li>first</li>

<li>second</li></ul>

`);

上面代码中，所有模板字符串的空格和换行，都是被保留的，比如<ul>标签前面会有一个换行。如果你不想要这个换行，可以使用trim方法消除它。

$('#list').html(`<ul>

<li>first</li>

<li>second</li></ul>

`.trim());

模板字符串中嵌入变量，需要将变量名写在${}之中。

function authorize(user, action) {

if (!user.hasPrivilege(action)) {

throw new Error(

// 传统写法为 // 'User ' // + user.name // + ' is not authorized to do ' // + action // + '.' `User ${user.name} is not authorized to do ${action}.`);

}}

大括号内部可以放入任意的JavaScript表达式，可以进行运算，以及引用对象属性。

let x = 1;let y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

`${x} + ${y \* 2} = ${x + y \* 2}`

// "1 + 4 = 5"let obj = {x: 1, y: 2};

`${obj.x + obj.y}`

// "3"

模板字符串之中还能调用函数。

function fn() {

return "Hello World";}

`foo ${fn()} bar`

// foo Hello World bar

如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的toString方法。

如果模板字符串中的变量没有声明，将报错。

// 变量place没有声明let msg = `Hello, ${place}`;

// 报错

由于模板字符串的大括号内部，就是执行JavaScript代码，因此如果大括号内部是一个字符串，将会原样输出。

`Hello ${'World'}`

// "Hello World"

模板字符串甚至还能嵌套。

const tmpl = addrs => `

<table>

${addrs.map(addr => `

<tr><td>${addr.first}</td></tr>

<tr><td>${addr.last}</td></tr>

`).join('')}

</table>

`;

上面代码中，模板字符串的变量之中，又嵌入了另一个模板字符串，使用方法如下。

const data = [

{ first: '<Jane>', last: 'Bond' },

{ first: 'Lars', last: '<Croft>' },];

console.log(tmpl(data));

// <table>//// <tr><td><Jane></td></tr>// <tr><td>Bond</td></tr>//// <tr><td>Lars</td></tr>// <tr><td><Croft></td></tr>//// </table>

如果需要引用模板字符串本身，在需要时执行，可以像下面这样写。

// 写法一let str = 'return ' + '`Hello ${name}!`';let func = new Function('name', str);func('Jack') // "Hello Jack!"

// 写法二let str = '(name) => `Hello ${name}!`';let func = eval.call(null, str);func('Jack') // "Hello Jack!"

## 实例：模板编译

下面，我们来看一个通过模板字符串，生成正式模板的实例。

let template = `<ul>

<% for(let i=0; i < data.supplies.length; i++) { %>

<li><%= data.supplies[i] %></li>

<% } %></ul>

`;

上面代码在模板字符串之中，放置了一个常规模板。该模板使用<%...%>放置JavaScript代码，使用<%= ... %>输出JavaScript表达式。

怎么编译这个模板字符串呢？

一种思路是将其转换为JavaScript表达式字符串。

echo('<ul>');for(let i=0; i < data.supplies.length; i++) {

echo('<li>');

echo(data.supplies[i]);

echo('</li>');};echo('</ul>');

这个转换使用正则表达式就行了。

let evalExpr = /<%=(.+?)%>/g;let expr = /<%([\s\S]+?)%>/g;

template = template

.replace(evalExpr, '`); \n echo( $1 ); \n echo(`')

.replace(expr, '`); \n $1 \n echo(`');

template = 'echo(`' + template + '`);';

然后，将template封装在一个函数里面返回，就可以了。

let script =

`(function parse(data){

let output = "";

function echo(html){

output += html;

}

${ template }

return output;})`;

return script;

将上面的内容拼装成一个模板编译函数compile。

function compile(template){

const evalExpr = /<%=(.+?)%>/g;

const expr = /<%([\s\S]+?)%>/g;

template = template

.replace(evalExpr, '`); \n echo( $1 ); \n echo(`')

.replace(expr, '`); \n $1 \n echo(`');

template = 'echo(`' + template + '`);';

let script =

`(function parse(data){

let output = "";

function echo(html){

output += html;

}

${ template }

return output;

})`;

return script;}

compile函数的用法如下。

let parse = eval(compile(template));

div.innerHTML = parse({ supplies: [ "broom", "mop", "cleaner" ] });

// <ul>// <li>broom</li>// <li>mop</li>// <li>cleaner</li>// </ul>

## 标签模板

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。

alert`123`

// 等同于alert(123)

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

但是，如果模板字符里面有变量，就不是简单的调用了，而是会将模板字符串先处理成多个参数，再调用函数。

let a = 5;let b = 10;

tag`Hello ${ a + b } world ${ a \* b }`;

// 等同于tag(['Hello ', ' world ', ''], 15, 50);

上面代码中，模板字符串前面有一个标识名tag，它是一个函数。整个表达式的返回值，就是tag函数处理模板字符串后的返回值。

函数tag依次会接收到多个参数。

function tag(stringArr, value1, value2){

// ...}

// 等同于function tag(stringArr, ...values){

// ...}

tag函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分，也就是说，变量替换只发生在数组的第一个成员与第二个成员之间、第二个成员与第三个成员之间，以此类推。

tag函数的其他参数，都是模板字符串各个变量被替换后的值。由于本例中，模板字符串含有两个变量，因此tag会接受到value1和value2两个参数。

tag函数所有参数的实际值如下。

* 第一个参数：['Hello ', ' world ', '']
* 第二个参数: 15
* 第三个参数：50

也就是说，tag函数实际上以下面的形式调用。

tag(['Hello ', ' world ', ''], 15, 50)

我们可以按照需要编写tag函数的代码。下面是tag函数的一种写法，以及运行结果。

let a = 5;let b = 10;

function tag(s, v1, v2) {

console.log(s[0]);

console.log(s[1]);

console.log(s[2]);

console.log(v1);

console.log(v2);

return "OK";}

tag`Hello ${ a + b } world ${ a \* b}`;

// "Hello "// " world "// ""// 15// 50// "OK"

下面是一个更复杂的例子。

let total = 30;let msg = passthru`The total is ${total} (${total\*1.05} with tax)`;

function passthru(literals) {

let result = '';

let i = 0;

while (i < literals.length) {

result += literals[i++];

if (i < arguments.length) {

result += arguments[i];

}

}

return result;}

msg // "The total is 30 (31.5 with tax)"

上面这个例子展示了，如何将各个参数按照原来的位置拼合回去。

passthru函数采用 rest 参数的写法如下。

function passthru(literals, ...values) {

let output = "";

let index;

for (index = 0; index < values.length; index++) {

output += literals[index] + values[index];

}

output += literals[index]

return output;}

“标签模板”的一个重要应用，就是过滤 HTML 字符串，防止用户输入恶意内容。

let message =

SaferHTML`<p>${sender} has sent you a message.</p>`;

function SaferHTML(templateData) {

let s = templateData[0];

for (let i = 1; i < arguments.length; i++) {

let arg = String(arguments[i]);

// Escape special characters in the substitution. s += arg.replace(/&/g, "&amp;")

.replace(/</g, "&lt;")

.replace(/>/g, "&gt;");

// Don't escape special characters in the template. s += templateData[i];

}

return s;}

上面代码中，sender变量往往是用户提供的，经过SaferHTML函数处理，里面的特殊字符都会被转义。

let sender = '<script>alert("abc")</script>'; // 恶意代码let message = SaferHTML`<p>${sender} has sent you a message.</p>`;

message

// <p>&lt;script&gt;alert("abc")&lt;/script&gt; has sent you a message.</p>

标签模板的另一个应用，就是多语言转换（国际化处理）。

i18n`Welcome to ${siteName}, you are visitor number ${visitorNumber}!`

// "欢迎访问xxx，您是第xxxx位访问者！"

模板字符串本身并不能取代Mustache之类的模板库，因为没有条件判断和循环处理功能，但是通过标签函数，你可以自己添加这些功能。

// 下面的hashTemplate函数// 是一个自定义的模板处理函数let libraryHtml = hashTemplate`

<ul>

#for book in ${myBooks}

<li><i>#{book.title}</i> by #{book.author}</li>

#end

</ul>

`;

除此之外，你甚至可以使用标签模板，在JavaScript语言之中嵌入其他语言。

jsx`

<div>

<input

ref='input'

onChange='${this.handleChange}'

defaultValue='${this.state.value}' />

${this.state.value}

</div>

`

上面的代码通过jsx函数，将一个DOM字符串转为React对象。你可以在Github找到jsx函数的[具体实现](https://gist.github.com/lygaret/a68220defa69174bdec5)。

下面则是一个假想的例子，通过java函数，在JavaScript代码之中运行Java代码。

java`

class HelloWorldApp {

public static void main(String[] args) {

System.out.println(“Hello World!”); // Display the string. }}

`

HelloWorldApp.main();

模板处理函数的第一个参数（模板字符串数组），还有一个raw属性。

console.log`123`

// ["123", raw: Array[1]]

上面代码中，console.log接受的参数，实际上是一个数组。该数组有一个raw属性，保存的是转义后的原字符串。

请看下面的例子。

tag`First line\nSecond line`

function tag(strings) {

console.log(strings.raw[0]);

// strings.raw[0] 为 "First line\\nSecond line" // 打印输出 "First line\nSecond line"}

上面代码中，tag函数的第一个参数strings，有一个raw属性，也指向一个数组。该数组的成员与strings数组完全一致。比如，strings数组是["First line\nSecond line"]，那么strings.raw数组就是["First line\\nSecond line"]。两者唯一的区别，就是字符串里面的斜杠都被转义了。比如，strings.raw数组会将\n视为\\和n两个字符，而不是换行符。这是为了方便取得转义之前的原始模板而设计的。

## String.raw()

ES6还为原生的String对象，提供了一个raw方法。

String.raw方法，往往用来充当模板字符串的处理函数，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，对应于替换变量后的模板字符串。

String.raw`Hi\n${2+3}!`;

// "Hi\\n5!"

String.raw`Hi\u000A!`;

// 'Hi\\u000A!'

如果原字符串的斜杠已经转义，那么String.raw不会做任何处理。

String.raw`Hi\\n`

// "Hi\\n"

String.raw的代码基本如下。

String.raw = function (strings, ...values) {

let output = "";

for (let index = 0; index < values.length; index++) {

output += strings.raw[index] + values[index];

}

output += strings.raw[index]

return output;}

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方法也可以作为正常的函数使用。这时，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。

String.raw({ raw: 'test' }, 0, 1, 2);

// 't0e1s2t'

// 等同于String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);

## 模板字符串的限制

前面提到标签模板里面，可以内嵌其他语言。但是，模板字符串默认会将字符串转义，导致无法嵌入其他语言。

举例来说，标签模板里面可以嵌入 LaTEX 语言。

function latex(strings) {

// ...}

let document = latex`

\newcommand{\fun}{\textbf{Fun!}} // 正常工作\newcommand{\unicode}{\textbf{Unicode!}} // 报错\newcommand{\xerxes}{\textbf{King!}} // 报错

Breve over the h goes \u{h}ere // 报错`

上面代码中，变量document内嵌的模板字符串，对于 LaTEX 语言来说完全是合法的，但是 JavaScript 引擎会报错。原因就在于字符串的转义。

模板字符串会将\u00FF和\u{42}当作 Unicode 字符进行转义，所以\unicode解析时报错；而\x56会被当作十六进制字符串转义，所以\xerxes会报错。也就是说，\u和\x在 LaTEX 里面有特殊含义，但是 JavaScript 将它们转义了。

为了解决这个问题，现在有一个[提案](https://tc39.github.io/proposal-template-literal-revision/)，放松对标签模板里面的字符串转义的限制。如果遇到不合法的字符串转义，就返回undefined，而不是报错，并且从raw属性上面可以得到原始字符串。

function tag(strs) {

strs[0] === undefined

strs.raw[0] === "\\unicode and \\u{55}";}

tag`\unicode and \u{55}`

上面代码中，模板字符串原本是应该报错的，但是由于放松了对字符串转义的限制，所以不报错了，JavaScript引擎将第一个字符设置为undefined，但是raw属性依然可以得到原始字符串，因此tag函数还是可以对原字符串进行处理。

注意，这种对字符串转义的放松，只在标签模板解析字符串时生效，不是标签模板的场合，依然会报错。

let bad = `bad escape sequence: \unicode`; // 报错