**第4章 字符串的扩展**

ES6加强了对Unicode的支持，并且扩展了字符串对象。

4.1 字符的Unicode表示法

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

1. “\u0061”

2. // “a”

3.

但是，这种表示法只限于\u0000~\uffff之间的字符。超出这个范围的字符，必须用2个双字节的形式表达。

1. “\uD842\uDFB7”

2. // “𠮷”

3.

1.       "\u20bb7";

2.       // " 7"

3.

上面的代码表示，如果直接在\u后面跟上超过0xffff的数值（比如\u20bb7），JavaScript就会理解成\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。

1. "\u{20BB7}"

2. // 𠮷

3.

4. "\u{41}\u{42}\u{43}"

5. // ABC

6.

7. let hello = 123;

8. hell\u{6F} // 123

9.

10. "\u{1F680}" === "\uD83D\uDE80"

11. // true

12.

上面的代码中，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公有6种方法可以表示一个字符。

1. '\z' === 'z'  // true

2. '\172' === 'z' // true

3. '\x7A' === 'z' // true

4. '\u007A' === 'z' // true

5. '\u{7A}' === 'z' // true

6.

4.2 codePointAt()

JavaScript内部，字符以UTF-16的格式存储，每个字符固定为2字节。对于那些需要4个字节存储的字符（Unicode码点大于0xFFFF的字符），JavaScript会认为它们是2个字符。

1. var s = '𠮷';

2.

3. s.length // 2

4. s.charAt(0); // ''

5. s.charAt(1); // ''

6. s.charCodeAt(0); // 55362

7. s.charCodeAt(1); // 57271

8.

上面的代码中，汉字“𠮷”的码点是0x20BB7，UTF-16编码为0xD842 0xDFB7（十进制为55362 57271），需要4个字节存储。对于这种4个字节的字符，JavaScript不能正确处理，字符串长度会被误判为2，而且charAt方法无法读取整个字符，charCodeAt方法只能分别返回前2个字节和后2个字节的值。

ES6提供了codePointAt方法，能够正确处理4个字节存储的字符，返回一个字符的码点。

1. var s = '𠮷a';

2.

3. s.codePointAt(0); // 134071

4. s.codePointAt(1); // 57271

5.

6. s.codePointAt(2); // 97

7.

codePointAt方法的参数是字符在字符串中的位置（从0开始）。上面的代码中，JavaScript将“𠮷a”视为3个字符。codePointAt方法在第一个字符上正确识别了“𠮷”，返回了它的十进制码点134071（即十六进制的20BB7）。在第二个字符（即“𠮷”的后2个字节）和第三个字符“a”上，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2个字节存储的常规字符，它返回的结果与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。

1. var s = '𠮷a';

2.

3. s.codePointAt(0).toString(16); // 20BB7

4. s.charCodeAt(2).toString(16); // 61

5.

你可能注意到了，codePointAt方法的参数仍然是不正确的。比如，上面的代码中，字符a在字符串s中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。

1. var s = '𠮷a';

2.

3. for (let ch of s) {

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

5. }

6.

7. // 20bb7

8. // 61

9.

codePointAt方法是测试一个字符由2个字节还是4个字节组成的最简单方法。

1. var s = '𠮷a';

2.

3. function is32Bit(c) {

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

5. }

6.

7. console.log(is32Bit("𠮷"));

8. console.log(is32Bit("a"));

9.

4.3 String.fromCodePoint()

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

1. String. fromCharCode(0x20BB7);

2. // ஷ

3.

上面的代码中，String.fromCharCode不能识别大于0xFFFF的码点，所以0x20BB7就发生了溢出，最高为2被舍弃，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的码点，弥补了String.fromCharCode方法的不足。在作用上，正好与codePointAt方法相反。

1. String.fromCodePoint(0x20BB7);

2. // 𠮷

3.

4. String.fromCodePoint(0x78, 0x1f680, 0x79) === "x\uD83D\uDE80y"

5. // true

6.

上面的代码中，如果String.fromCharCode方法有多个参数，则它们会被合成一个字符串返回。

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。

4.4 字符串的遍历器接口

ES6为字符串添加了遍历接口（详见14章），使字符串可以由for…of循环遍历。

1. for (let codePoint of "foo") {

2.     console.log(codePoint);

3. }

4.

5. // "f", "o", "o"

6.

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。

1. var text = String.fromCodePoint(0x20bb7);

2.

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

4.     console.log(text[i]);

5. }

6. // " "

7. // " "

8.

9. for (let i of text) {

10.     console.log(i);

11. }

12. // "𠮷"

13.

上面的代码中，字符串text只有一个字符，但是for循环会认为它包含2个字符（都不可打印），而for…of循环会正确识别出这个字符。

4.5 at()

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

1. "abc".charAt(0); // "a"

2. "𠮷".charAt(0); // "\uD842"

3.

上面的代码中，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。

ES7为字符串实例提供了at方法，可以识别Unicode编号大于0xFFFF的字符，返回正确的字符。

Chrome浏览器已经支持该方法。

1. "abc".at(0); // "a"

2. "𠮷".at(0); // "𠮷"

3.

4.6 normalize()

为了表示语调和重音符号，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合成为一个字符，比如O（\u004f）和Ǒ（\u030c）合成Ǒ（\u004f\u030c）。

这两种表示法，在视觉和语义上都等价，但是JavaScript无法识别。

1. "\u01d1" === "\u004f\u030c" // false

2. "\u01d1".length // 1

3. "\u004f\u030c".length // 2

4.

上面的代码表示，Javascript将合成字符视为两个字符，导致两种表示法不等价。

ES6为字符串实例提供了normalize方法，用来将字符的不同表示法统一为同样的形式，这称为Unicode正规化。

1. "\u01d1".normalize() === "\u004f\u030c".normalize()

2. // true

3.

normalize方法可以接受4个参数。

* 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，返回合成字符分解出的多个简单字符。
* 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”指的是语义上等价，但视觉上不等价，比如“囍”和“双喜”。（这只是举例，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，返回合成字符分解出的多个简单字符。

1. "\u004f\u030c".normalize("NFC").length // 1

2. "\u004f\u030c".normalize("NFD").length // 2

3.

上面的代码表示，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3个或3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。

4.7 includes()，startsWith()，endsWith()

传统上，JavaScript中只有indexOf方法可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源字符串的头部。
* endsWith()：返回布尔值，表示参数字符串是否在源字符串的尾部。

1. var s = "Hello World!";

2.

3. s.startsWith("Hello");  // true

4. s.endsWith("!");    // true

5. s.includes("o");    // true

6.

这3个方法都支持第2个参数，表示开始搜索的位置。

1. var s = "Hello World!";

2.

3. s.startsWith("World", 6);  // true

4. s.endsWith("Hello", 5);    // true

5. s.includes("Hello", 6);    // false

6.

上面的代码表示，使用第2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的字符。

4.8 repeat()

repeat方法返回一个新字符串，表示将原字符串重复n次。

1. "x".repeat(3); // "xxx"

2. "hello".repeat(2); // "hellohello"

3. "na".repeat(0); // ""

4.

参数如果是小数，会被取整。

1. "na".repeat(2.9); // "nana"

2.

如果repeat的参数是负数或者Infinity，会报错。

1. "na".repeat(Infinity);

2. //RangeError

3. "na".repeat(-1);

4. //RangeError

5.

但如果参数是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。

1. "na".repeat(-0.9); // ""

2.

参数NaN等同于0.

1. "na".repeat(NaN); // ""

2.

如果repeat的参数是字符串，则会先转换为数字。

1. "na".repeat(""); // ""

2. "na".repeat("2"); // "nana"

3.

4.9 padStart()，padEnd()

ES7推出了字符串补全长度的功能。如果某个字符串未达指定长度，会在头部或尾部补全。

padStart用于头部补全，padEnd用于尾部补全。

1. "x".padStart(5, "ab"); // "ababx"

2. "x".padStart(4, "ab"); // "abax"

3.

4. "x".padEnd(5, "ab"); // "xabab"

5. "x".padEnd(4, "ab"); // "xaba"

6.

上面的代码中，padStart和padEnd分别接受两个参数，第一个参数用来指定字符串的最小长度，第二个参数则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。

1. "xxx".padStart(2, "ab"); // "xxx"

2. "xxx".padEnd(2, "ab"); // "xxx"

3.

如果省略第二个参数，则会用空格来补全。

1. "x".padStart(2); // " x"

2. "x".padEnd(2); // " x"

3.

4.10 模板字符串

传统的JavaScript输出模板通常是这样写的。

1. $("#result").append(

2.     "There are <b>" + basket.count + "</b> " +

3.         "itens in your basket，" +

4.         "<em>" + basket.onSale +

5.         "</em> are on sale!"

6. );

7.

上面这种写法相当烦琐而不方便，ES6引入了模板字符串来解决这个问题。

1. $("#result").append(

2.     `There are <b>${basket.count}</b>  +

3.         itens in your basket， +

4.         <em>${basket.onSale}</em> are on sale!`

5. );

6.

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。

1. // 普通字符串

2. `In JavaScript '\n' is a line-feed.`

3.

4. // 多行字符串

5. `In JavaScript this is

6.  not legal.`

7.

8. // 字符串中嵌入变量

9. var name = "Bob", time = "today";

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

11.

以上代码中的字符串都使用了反引号。如果在模板字符串中需要使用反引号，则在其前面要用反斜杠转义。

1. var greeting = `\`yo\` World!`;

2.

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中。

1. $("#warning").html(`

2.     <h1>Watch out!</h1>

3.     <p>Unauthoried hockeying can result in penalties

4.     of up to ${maxPenalty} minutes.</p>

5. `);

6.

在模板字符串中嵌入变量，需要将变量名写在${}中。

1. function authorize(user, action) {

2.     if (!user.hasPrivilege(action)) {

3.         throw new Error(

4.             // 传统写法为

5.             // 'User '

6.             // + user.name

7.             // + ' is not authorized to do '

8.             // + action

9.             // + "."

10.             `User ${user.name} is not authorized to do ${action}.`

11.         );

12.     }

13. }

14.

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

1. var x = 1;

2. var y = 2;

3.

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

5. // "1 + 2 = 3"

6.

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

8. // "1 + 4 = 5"

9.

10. var obj  = {

11.     x: 1,

12.     y: 2

13. }

14.

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

16. // 3

18.

模板字符串中还能调用函数。

1. function fn() {

2.     return "Hello World!";

3. }

4.

5. `foo ${fn()} bar`

6. // foo Hello World bar

7.

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。

1. // 变量place没有声明

2. var msg = `Hello, ${place}`;

3. // 报错

4.

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

1. `Hello ${'World'}`

2. // Hello World

3.

如果需要引用模板字符串本身，可以像下面这样写。

1. // 写法1

2. let str = "return " + "`Hello ${name}!`";

3. let func = new Function("name", str);

4. func("Jack") // "Hello Jack"

5.

6. // 写法2

7. let str = "(name) => `Hello ${name}`";

8. let func = eval.call(null, str);

9. func("Jack") // "Hello Jack"

10.

4.11 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。

1. var template = `

2.     <ul>

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

4.             <li><%= data.supplies %></li>

5.         <% } %>

6.     </ul>

7. `;

8.

上面的代码在模板字符串中放置了一个常规模板。该模板使用<% … %>放置JavaScript代码，使用<%= … %>放置输出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。

1. echo("<ul>")

2. for (var i = 0; i < data.supplies.length; i++) {

3.     echo("<li>");

4.     echo(data.supplies[i]);

5.     echo("</li>");

6. }

7. echo("</ul>")

8.

这个转换使用正则表达式即可。

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

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

3.

4. template = template

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

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

7.

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

9.

然后，将template封装在一个函数里返回即可。

1. var script = `

2.     (function parse(data) {

3.         var output = "";

4.         function ehco(html) {

5.             output += html;

6.         }

7.

8.         ${ template }

9.

10.         return output;

11.     })

12. `;

13.

14. return script;

15.

将上面内容拼装成一个模板编译函数compile。

1. function compile(template) {

2.

3.     var evalExpr = /<%=(.+?)%>/g;

4.     var expr = /<%([\s\S]+?)%>/g;

5.

6.     template = template

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

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

9.

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

11.

12.     var script = `

13.         (function parse(data) {

14.             var output = "";

15.             function echo (html) {

16.                 output += html;

17.             }

18.

19.             ${ template }

20.

21.             return output;

22.         })

23.     `;

24.

25.     return script;

26. }

compile函数的用法如下。

1. let parse = eval(compile(template));

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

3. // <ul>

4. // <li>broom</li>

5. // <li>mop</li>

6. // <li>cleaner</li>

7. // </ul>

8.

4.12 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）。

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。

1. var a = 5;

2. var b = 5;

3.

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

5.

上面的代码中，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会依次接受到多个参数。

1. function tag(stringArr, value1, value2) {

2.     // ...

3. }

4.

5. // 等同于

6.

7. function tag(stringArr, ...values) {

8.     // ...

9. }

10.

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函数的其它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会接收到value1和value2两个参数。

* 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被调用。

1. tag(["Hello ", " world ", ""], 15, 50);

2.

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

1. function tag(stringArr, ...values) {

2.     let result = "";

3.

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

5.         result += (stringArr[i] + values[i])

6.     }

7.     var lastStr = stringArr[stringArr.length - 1];

8.     if (lastStr !== "") {

9.         result += lastStr;

10.     }

11.     return result;

12. }

13.

14. var a = 30;

15. var b = 30;

16.

17. console.log(tag`Hello ${a + b} world ${a \* b}`);

19.

下面是一个更复杂的例子。

1. var total = 30;

2. var msg = passthru`The total is ${total} (${total \* 1.05} with tax)`;

3.

4. function passthru(literals) {

5.     var result = "";

6.     var i = 0;

7.      while (i < literals.length) {

8.         result += literals[i++];

9.         if (i < arguments.length) {

10.             result += arguments[i];

11.         }

12.      }

13.      return result;

14. }

15.

16. console.log(msg);

17.

上面这个例子展示了如何将各个参数按照原来的位置拼回去。

passthru函数采用rest参数的写法如下。

1. function passthru(literals, ...values) {

2.     var output = "";

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

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

5.     }

6.

7.     output += literals[index];

8.     return output;

9. }

10.

“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。

1. var sender = "<h1>xiesy</h1>";

2.

3. var message = SafeHTML`<p>${sender} has sent you a message.</p>`;

4.

5. function SafeHTML(templateData) {

6.

7.     var s = templateData[0];

8.     console.log(s);

9.     for (var i = 1; i < templateData.length; i++) {

10.

11.         // Escape special characters in the substitution.

12.         var arg = String(arguments[i]);

13.

14.         s += arg.replace(/&/g, "&")

15.             .replace(/</g, "<")

16.             .replace(/>/g, ">");

17.

18.             // Don't escape special characters in the template.

19.         s += templateData[i];

20.     }

21.     return s;

22. }

23.

24. console.log(message);

25.

上面的代码中，经过SafeHTML函数的处理，HTML字符串中的特殊字符都会被转义。

标签模板的另一个应用是多语言转换（国国际化处理）。

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

2. // 欢饮访问xxxx，你是第xxx位访问者!

3.

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。

1. // 下面的hashTemplate函数是一个自定义的模板处理函数

2. var libraryHtml = hashTemplate`

3.     <ul>

4.         #for book in ${myBooks}

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

6.         #end

7.     </ul>

8. `;

9.

除此自外，你甚至可以使用标签模板，在JavaScript语言中嵌入其他语言。

1. jsx`

2.     <div>

3.         <input

4.             ref='input'

5.             onChange='${this.handleChange}'

6.             defaultValue='${this.state.value}'

7.         />

8.         ${this.state.value}

9.     </div>

10. `;

11.

上面的代码通过jsx函数将一个DOM字符串转成了React对象。你可以在Github上找到jsx函数的具体实现（<https://gist.github.com/lygaret/a68220defa69174bdec5>）。

下面是一个假象的例子，通过java函数在JavaScript代码中运行Java代码。

1. java`

2.     class HelloWorldApp {

3.         public static void main(String[] args) {

4.

5.             System.out.println("Hello World!"); // Display the string

6.         }

7.     }

8. `;

9.  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。

1. tag`First line\nSecond line`;

2.

3. function tag(strings) {

4.     console.log(strings.raw[0]);

5.     // "First line\\nSecond line"

6. }

7.

上面的代码中，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数组几乎完全一致。比如，strings数组是[“First line\nSecond lint”]，那么strings.raw数组就是[“First line\\nSecond line”]。两者唯一的区别，就是字符串里面的反斜线都被转义了。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。

4.13 String.raw()

ES6还为原生的String对象提供了raw方法。

String.raw方法往往用来充当模板字符串的处理函数，返回一个反斜线都被转义（即反斜线前面再加上一个反斜线）的字符串，对应于替换变量后的模板字符串。

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

2. // "Hi\\n5!"

3.

4. String.raw`Hi\u000A!`;

5. // "Hi\\u000A!"

6.

如果原字符串中的反斜线已经转义，那么String.raw不会做任何处理。

1. String.raw`Hi\\n`;

2. // "Hi\\n"

4.

String.raw的代码基本如下。

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

2.     var output = "";

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

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

5.     }

6.     output += strings.raw[index];

7.     return output;

8. };

9.

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，并对反斜线进行转义，方便下一步作为字符串使用。

String.raw方法也可以作为正常的函数使用。这时，其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。

1. String.raw({raw: "test"}, 1, 2, 3);

2. // t1e2s3t

3.

4. // 等同于

5. String.raw({raw: ["t", "e", "s", "t"]}, 1, 2, 3);

6.