ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

字符串的扩展

本章介绍 ES6 对字符串的改造和增强，下一章介绍字符串对象的新增方法。

---

1. 字符的 Unicode 表示法

ES6 加强了对 Unicode 的支持，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"
// "𠮷"

"\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}"
// "𠮷"

"\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
```

---

2. 字符串的遍历器接口

上一章 下一章

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);
}
// "𠮷"
```

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。

---

3. 直接输入 U+2028 和 U+2029

JavaScript 字符串允许直接输入字符，以及输入字符的转义形式。举例来说，“中”的 Unicode 码点是 U+4e2d，你可以直接在字符串里面输入这个汉字，也可以输入它的转义形式 `\u4e2d`，两者是等价的。

```
'中' === '\u4e2d' // true
```

但是，JavaScript 规定有5个字符，不能在字符串里面直接使用，只能使用转义形式。

- U+005C：反斜杠（reverse solidus）
- U+000D：回车（carriage return）
- U+2028：行分隔符（line separator）
- U+2029：段分隔符（paragraph separator）
- U+000A：换行符（line feed）

举例来说，字符串里面不能直接包含反斜杠，一定要转义写成 `\\` 或者 `\u005c`。

这个规定本身没有问题，麻烦在于 JSON 格式允许字符串里面直接使用 U+2028（行分隔符）和 U+2029（段分隔符）。这样一来，服务器输出的 JSON 被 `JSON.parse` 解析，就有可能直接报错。

```
const json = '"\u2028"';
JSON.parse(json); // 可能报错
```

JSON 格式已经冻结（RFC 7159），没法修改了。为了消除这个报错，ES2019 允许 JavaScript 字符串直接输入 U+2028（行分隔符）和 U+2029（段分隔符）。

```
const PS = eval("'\u2029'");
```

根据这个提案，上面的代码不会报错。

注意，模板字符串现在就允许直接输入这两个字符。另外，正则表达式依然不允许直接输入这两个字符，这是没有问题的，因为 JSON 本来就不允许直接包含正则表达式。

---

4. JSON.stringify() 的改造

根据标准，JSON 数据必须是 UTF-8 编码。但是，现在的 `JSON.stringify()` 方法有可能返回不符合 UTF-8 标准的字符串。

具体来说，UTF-8 标准规定， `0xD800` 到 `0xDFFF` 之间的码点，不能单独使用，必须配对使用。比如， `\uD834\uDF06` 是两个码点，但是必须放在一起配对使用，代表字符 𝌆 。这是为了表示码点大于 `0xFFFF` 的字符的一种变通方法。单独使用 `\uD834` 和 `\uDF06` 这两个码点是不合法的，或者颠倒顺序也不行，因为 `\uDF06\uD834` 并没有对应的字符。

`JSON.stringify()` 的问题在于，它可能返回 `0xD800` 到 `0xDFFF` 之间的单个码点。

```
JSON.stringify('\u{D834}') // "\u{D834}"
```

为了确保返回的是合法的 UTF-8 字符，ES2019 改变了 `JSON.stringify()` 的行为。如果遇到 `0xD800` 到 `0xDFFF` 之间的单个码点，或者不存在的配对形式，它会返回转义字符串，留给应用自己决定下一步的处理。

```
JSON.stringify('\u{D834}') // ""\\uD834""
JSON.stringify('\uDF06\uD834') // ""\\udf06\\ud834""
```

---

5. 模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的（下面使用了 jQuery 的方法）。

```
$('#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，可以写成函数。

```
let func = (name) => `Hello ${name}!`;
func('Jack') // "Hello Jack!"
```

上面代码中，模板字符串写成了一个函数的返回值。执行这个函数，就相当于执行这个模板字符串了。

---

6. 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>
```

---

7. 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`hello`
// 等同于
alert(['hello'])
```

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。

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。

---

8. 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，ES2018 放松了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
```

留言