第 2 章　HTML中的JavaScript

本章内容

- 使用<script>元素
- 行内脚本与外部脚本的比较
- 文档模式对JavaScript有什么影响
- 确保JavaScript不可用时的用户体验

将JavaScript引入网页，首先要解决它与网页的主导语言HTML的关系问题。在JavaScript早期，网景公司的工作人员希望在将JavaScript引入HTML页面的同时，不会导致页面在其他浏览器中渲染出问题。通过反复试错和讨论，他们最终做出了一些决定，并达成了向网页中引入通用脚本能力的共识。当初他们的很多工作得到了保留，并且最终形成了HTML规范。

2.1　<script>元素

将JavaScript插入HTML的主要方法是使用<script>元素。这个元素是由网景公司创造出来，并最早在Netscape Navigator 2中实现的。后来，这个元素被正式加入到HTML规范。<script>元素有下列8个属性。

- async：可选。表示应该立即开始下载脚本，但不能阻止其他页面动作，比如下载资源或等待其他脚本加载。只对外部脚本文件有效。
- charset：可选。使用src属性指定的代码字符集。这个属性很少使用，因为大多数浏览器不在乎它的值。
- crossorigin：可选。配置相关请求的CORS（跨源资源共享）设置。默认不使用CORS。crossorigin="anonymous"配置文件请求不必设置凭据标志。crossorigin="use-credentials"设置凭据标志，意味着出站请求会包含凭据。
- defer：可选。表示在文档解析和显示完成后再执行脚本是没有问题的。只对外部脚本文件有效。在IE7及更早的版本中，对行内脚本也可以指定这个属性。
- integrity：可选。允许比对接收到的资源和指定的加密签名以验证子资源完整性（SRI，Subresource Intergrity）。如果接收到的资源的签名与这个属性指定的签名不匹配，则页面会报错，脚本不会执行。这个属性可以用于确保内容分发网络（CDN，Content Delivery Network）不会提供恶意内容。
- language：废弃。最初用于表示代码块中的脚本语言（如"JavaScript"、"JavaScript 1.2"或"VBScript"）。大多数浏览器都会忽略这个属性，不应该再使用它。
- src：可选。表示包含要执行的代码的外部文件。
- type：可选。代替language，表示代码块中脚本语言的内容类型（也称MIME类型）。按照惯例，这个值始终都是"text/javascript"，尽管"text/javascript"和"text/ecmascript"都已经废弃了。JavaScript文件的MIME类型通常是"application/x-javascript"，不过给type属性这个值有可能导致脚本被忽略。在非IE的浏览器中有效的其他值还有"application/javascript"和"application/ecmascript"。如果这个值是module，则代码会被当成ES6模块，而且只有这时候代码中才能出现import和export关键字。

使用<script>的方式有两种：通过它直接在网页中嵌入JavaScript代码，以及通过它在网页中包含外部JavaScript文件。

要嵌入行内JavaScript代码，直接把代码放在<script>元素中就行：

```
<script>
  function sayHi() {
    console.log("Hi!");
  }
</script>
```

包含在<script>内的代码会被从上到下解释。在上面的例子中，被解释的是一个函数定义，并且该函数会被保存在解释器环境中。在<script>元素中的代码被计算完成之前，页面的其余内容不会被加载，也不会被显示。

在使用行内JavaScript代码时，要注意代码中不能出现字符串</script>。比如，下面的代码会导致浏览器报错：

```
<script>
  function sayScript() {
    console.log("</script>");
  }
</script>
```

浏览器解析行内脚本的方式决定了它在看到字符串</script>时，会将其当成结束的</script>标签。想避免这个问题，只需要转义字符“\”**1**即可：

1此处的转义字符指在JavaScript中使用反斜杠“\”来向文本字符串添加特殊字符。——编者注

```
<script>
  function sayScript() {
    console.log("<\/script>");
  }
</script>
```

这样修改之后，代码就可以被浏览器完全解释，不会导致任何错误。

要包含外部文件中的JavaScript，就必须使用src属性。这个属性的值是一个URL，指向包含JavaScript代码的文件，比如：

```
<script src="example.js"></script>
```

这个例子在页面中加载了一个名为example.js的外部文件。文件本身只需包含要放在<script>的起始及结束标签中间的JavaScript代码。与解释行内JavaScript一样，在解释外部JavaScript文件时，页面也会阻塞。（阻塞时间也包含下载文件的时间。）在XHTML文档中，可以忽略结束标签，比如：

```
<script src="example.js"/>
```

以上语法不能在HTML文件中使用，因为它是无效的HTML，有些浏览器不能正常处理，比如IE。

> **注意** 按照惯例，外部JavaScript文件的扩展名是.js。这不是必需的，因为浏览器不会检查所包含JavaScript文件的扩展名。这就为使用服务器端脚本语言动态生成JavaScript代码，或者在浏览器中将JavaScript扩展语言（如TypeScript，或React的JSX）转译为JavaScript提供了可能性。不过要注意，服务器经常会根据文件扩展来确定响应的正确MIME类型。如果不打算使用.js扩展名，一定要确保服务器能返回正确的MIME类型。

另外，使用了src属性的<script>元素不应该再在<script>和</script>标签中再包含其他JavaScript代码。如果两者都提供的话，则浏览器只会下载并执行脚本文件，从而忽略行内代码。

浏览器在解析这个资源时，会向`src`属性指定的路径发送一个`GET`请求，以取得相应资源，假定是一个JavaScript文件。这个初始的请求不受浏览器同源策略限制，但返回并被执行的JavaScript则受限制。当然，这个请求仍然受父页面HTTP/HTTPS协议的限制。

来自外部域的代码会被当成加载它的页面的一部分来加载和解释。这个能力可以让我们通过不同的域分发JavaScript。不过，引用了放在别人服务器上的JavaScript文件时要格外小心，因为恶意的程序员随时可能替换这个文件。在包含外部域的JavaScript文件时，要确保该域是自己所有的，或者该域是一个可信的来源。`<script>`标签的`integrity`属性是防范这种问题的一个武器，但这个属性也不是所有浏览器都支持。

不管包含的是什么代码，浏览器都会按照`<script>`在页面中出现的顺序依次解释它们，前提是它们没有使用`defer`和`async`属性。第二个`<script>`元素的代码必须在第一个`<script>`元素的代码解释完毕才能开始解释，第三个则必须等第二个解释完，以此类推。

2.1.1 标签占位符

过去，所有`<script>`元素都被放在页面的`<head>`标签内，如下面的例子所示：

```

```

这种做法的主要目的是把外部的CSS和JavaScript文件都集中放到一起。不过，把所有JavaScript文件都放在<head>里，也就意味着必须把所有JavaScript代码都下载、解析和解释完成后，才能开始渲染页面（页面在浏览器解析到<body>的起始标签时开始渲染）。对于需要很多JavaScript的页面，这会导致页面渲染的明显延迟，在此期间浏览器窗口完全空白。为解决这个问题，现代Web应用程序通常将所有JavaScript引用放在<body>元素中的页面内容后面，如下面的例子所示：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  <script src="example1.js"></script>
  <script src="example2.js"></script>
```

```
  </body>
</html>
```

这样一来，页面会在处理JavaScript代码之前完全渲染页面。用户会感觉页面加载更快了，因为浏览器显示空白页面的时间短了。

2.1.2 推迟执行脚本

HTML 4.01为<script>元素定义了一个叫defer的属性。这个属性表示脚本在执行的时候不会改变页面的结构。因此，这个脚本完全可以在整个页面解析完之后再运行。在<script>元素上设置defer属性，会告诉浏览器应该立即开始下载，但执行应该推迟：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script defer src="example1.js"></script>
  <script defer src="example2.js"></script>
  </head>
  <body>
  <!-- 这里是页面内容 -->
  </body>
</html>
```

虽然这个例子中的<script>元素包含在页面的<head>中，但它们会在浏览器解析到结束的</html>标签后才会执行。HTML5规范要求脚本应该按照它们出现的顺序执行，因此第一个推迟的脚本会在第二个推迟的脚本之前执行，而且两者都会在DOMContentLoaded事件之前执行（关于事件，请参考第17章）。不过在实际当中，推迟执行的脚本不一定总会按顺序执行或者在DOMContentLoaded事件之前执行，因此最好只包含一个这样的脚本。

如前所述，defer属性只对外部脚本文件才有效。这是HTML5中明确规定的，因此支持HTML5的浏览器会忽略行内脚本的defer属性。IE4~7展示出的都是旧的行为，IE8及更高版本则支持HTML5定义的行为。

对defer属性的支持是从IE4、Firefox 3.5、Safari 5和Chrome 7开始的。其他所有浏览器则会忽略这个属性，按照通常的做法来处理脚本。考虑到这一点，还是把要推迟执行的脚本放在页面底部比较好。

> **注意** 对于XHTML文档，指定defer属性时应该写成defer="defer"。

2.1.3 异步执行脚本

HTML5为<script>元素定义了async属性。从改变脚本处理方式上看，async属性与defer类似。当然，它们两者也都只适用于外部脚本，都会告诉浏览器立即开始下载。不过，与defer不同的是，标记为async的脚本并不保证能按照它们出现的次序执行，比如：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
```

```
    <script async src="example1.js"></script>
    <script async src="example2.js"></script>
    </head>
    <body>
    <!-- 这里是页面内容 -->
    </body>
  </html>
```

在这个例子中，第二个脚本可能先于第一个脚本执行。因此，重点在于它们之间没有依赖关系。给脚本添加async属性的目的是告诉浏览器，不必等脚本下载和执行完后再加载页面，同样也不必等到该异步脚本下载和执行后再加载其他脚本。正因为如此，异步脚本不应该在加载期间修改DOM。

异步脚本保证会在页面的load事件前执行，但可能会在DOMContentLoaded（参见第17章）之前或之后。Firefox 3.6、Safari 5和Chrome 7支持异步脚本。使用async也会告诉页面你不会使用document.write，不过好的Web开发实践根本就不推荐使用这个方法。

> **注意** 对于XHTML文档，指定async属性时应该写成async="async"。

2.1.4 动态加载脚本

除了<script>标签，还有其他方式可以加载脚本。因为JavaScript可以使用DOM API，所以通过向DOM中动态添加script元素同样可以加载指定的脚本。只要创建一个script元素并将其添加到DOM即可。

```
let script = document.createElement('script');
script.src = 'gibberish.js';
document.head.appendChild(script);
```

当然，在把HTMLElement元素添加到DOM且执行到这段代码之前不会发送请求。默认情况下，以这种方式创建的<script>元素是以异步方式加载的，相当于添加了async属性。不过这样做可能会有问题，因为所有浏览器都支持createElement()方法，但不是所有浏览器都支持async属性。因此，如果要统一动态脚本的加载行为，可以明确将其设置为同步加载：

```
let script = document.createElement('script');
script.src = 'gibberish.js';
script.async = false;
document.head.appendChild(script);
```

以这种方式获取的资源对浏览器预加载器是不可见的。这会严重影响它们在资源获取队列中的优先级。根据应用程序的工作方式以及怎么使用，这种方式可能会严重影响性能。要想让预加载器知道这些动态请求文件的存在，可以在文档头部显式声明它们：

```
<link rel="preload" href="gibberish.js">
```

2.1.5 XHTML中的变化

可扩展超文本标记语言（XHTML，Extensible HyperText Markup Language）是将HTML作为XML的应用重新包装的结果。与HTML不同，在XHTML中使用JavaScript必须指定`type`属性且值为`text/javascript`，HTML中则可以没有这个属性。XHTML虽然已经退出历史舞台，但实践中偶尔可能也会遇到遗留代码，为此本节稍作介绍。

在XHTML中编写代码的规则比HTML中严格，这会影响使用`<script>`元素嵌入JavaScript代码。下面的代码块虽然在HTML中有效，但在XHML中是无效的。

```
<script type="text/javascript">
  function compare(a, b) {
    if (a < b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
</script>
```

在HTML中，解析`<script>`元素会应用特殊规则。XHTML中则没有这些规则。这意味着`a < b`语句中的小于号（`<`）会被解释成一个标签的开始，并且由于作为标签开始的小于号后面不能有空格，这会导致语法错误。

避免XHTML中这种语法错误的方法有两种。第一种是把所有小于号（`<`）都替换成对应的HTML实体形式（`<`）。结果代码就是这样的：

```
<script type="text/javascript">
  function compare(a, b) {
    if (a &lt; b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
</script>
```

这样代码就可以在XHTML页面中运行了。不过，缺点是会影响阅读。好在还有另一种方法。

第二种方法是把所有代码都包含到一个CDATA块中。在XHTML（及XML）中，CDATA块表示文档中可以包含任意文本的区块，其内容不作为标签来解析，因此可以在其中包含任意字符，包括小于号，并且不会引发语法错误。使用CDATA的格式如下：

```
<script type="text/javascript"><![CDATA[
  function compare(a, b) {
    if (a < b) {
```

```
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
]]></script>
```

在兼容XHTML的浏览器中，这样能解决问题。但在不支持CDATA块的非XHTML兼容浏览器中则不行。为此，CDATA标记必须使用JavaScript注释来抵消：

```
<script type="text/javascript">
//<![CDATA[
  function compare(a, b) {
    if (a < b) {
      console.log("A is less than B");
    } else if (a > b) {
      console.log("A is greater than B");
    } else {
      console.log("A is equal to B");
    }
  }
//]]>
</script>
```

这种格式适用于所有现代浏览器。虽然有点黑科技的味道，但它可以通过XHTML验证，而且对XHTML之前的浏览器也能优雅地降级。

> **注意** XHTML模式会在页面的MIME类型被指定为`"application/xhtml+xml"`时触发。并不是所有浏览器都支持以这种方式送达的XHTML。

2.1.6 废弃的语法

自1995年Netscape 2发布以来，所有浏览器都将JavaScript作为默认的编程语言。`type`属性使用一个MIME类型字符串来标识`<script>`的内容，但MIME类型并没有跨浏览器标准化。即使浏览器默认使用JavaScript，在某些情况下某个无效或无法识别的MIME类型也可能导致浏览器跳过（不执行）相关代码。因此，除非你使用XHML或`<script>`标签要求或包含非JavaScript代码，最佳做法是不指定`type`属性。

在最初采用`script`元素时，它标志着开始走向与传统HTML解析不同的流程。对这个元素需要应用特殊的解析规则，而这在不支持JavaScript的浏览器（特别是Mosaic）中会导致问题。不支持的浏览器会把`<script>`元素的内容输出到页面上，从而破坏页面的外观。

Netscape联合Mosaic拿出了一个解决方案，对不支持JavaScript的浏览器隐藏嵌入的JavaScript代码。最终方案是把脚本代码包含在一个HTML注释中，像这样：

```
<script><!--
  function sayHi(){
```

```
    console.log("Hi!");
  }
//--></script>
```

使用这种格式，Mosaic等浏览器就可以忽略<script>标签中的内容，而支持JavaScript的浏览器则必须识别这种模式，将其中的内容作为JavaScript来解析。

虽然这种格式仍然可以被所有浏览器识别和解析，但已经不再必要，而且不应该再使用了。在XHTML模式下，这种格式也会导致脚本被忽略，因为代码处于有效的XML注释当中。

2.2 行内代码与外部文件

虽然可以直接在HTML文件中嵌入JavaScript代码，但通常认为最佳实践是尽可能将JavaScript代码放在外部文件中。不过这个最佳实践并不是明确的强制性规则。推荐使用外部文件的理由如下。

- **可维护性**。JavaScript代码如果分散到很多HTML页面，会导致维护困难。而用一个目录保存所有JavaScript文件，则更容易维护，这样开发者就可以独立于使用它们的HTML页面来编辑代码。
- **缓存**。浏览器会根据特定的设置缓存所有外部链接的JavaScript文件，这意味着如果两个页面都用到同一个文件，则该文件只需下载一次。这最终意味着页面加载更快。
- **适应未来**。通过把JavaScript放到外部文件中，就不必考虑用XHTML或前面提到的注释黑科技。包含外部JavaScript文件的语法在HTML和XHTML中是一样的。

在配置浏览器请求外部文件时，要重点考虑的一点是它们会占用多少带宽。在SPDY/HTTP2中，预请求的消耗已显著降低，以轻量、独立JavaScript组件形式向客户端送达脚本更具优势。

比如，第一个页面包含如下脚本：

```
<script src="mainA.js"></script>
<script src="component1.js"></script>
<script src="component2.js"></script>
<script src="component3.js"></script>
...
```

后续页面可能包含如下脚本：

```
<script src="mainB.js"></script>
<script src="component3.js"></script>
<script src="component4.js"></script>
<script src="component5.js"></script>
...
```

在初次请求时，如果浏览器支持SPDY/HTTP2，就可以从同一个地方取得一批文件，并将它们逐个放到浏览器缓存中。从浏览器角度看，通过SPDY/HTTP2获取所有这些独立的资源与获取一个大JavaScript文件的延迟差不多。

在第二个页面请求时，由于你已经把应用程序切割成了轻量可缓存的文件，第二个页面也依赖的某些组件此时已经存在于浏览器缓存中了。

当然，这里假设浏览器支持SPDY/HTTP2，只有比较新的浏览器才满足。如果你还想支持那些比较老的浏览器，可能还是用一个大文件更合适。

2.3 文档模式

IE5.5发明了文档模式的概念，即可以使用doctype切换文档模式。最初的文档模式有两种：**混杂模式**（quirks mode）和**标准模式**（standards mode）。前者让IE像IE5一样（支持一些非标准的特性），后者让IE具有兼容标准的行为。虽然这两种模式的主要区别只体现在通过CSS渲染的内容方面，但对JavaScript也有一些关联影响，或称为副作用。本书会经常提到这些副作用。

IE初次支持文档模式切换以后，其他浏览器也跟着实现了。随着浏览器的普遍实现，又出现了第三种文档模式：**准标准模式**（almost standards mode）。这种模式下的浏览器支持很多标准的特性，但是没有标准规定得那么严格。主要区别在于如何对待图片元素周围的空白（在表格中使用图片时最明显）。

混杂模式在所有浏览器中都以省略文档开头的doctype声明作为开关。这种约定并不合理，因为混杂模式在不同浏览器中的差异非常大，不使用黑科技基本上就没有浏览器一致性可言。

标准模式通过下列几种文档类型声明开启：

```
<!-- HTML 4.01 Strict -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!-- XHTML 1.0 Strict -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<!-- HTML5 -->
<!DOCTYPE html>
```

准标准模式通过过渡性文档类型（Transitional）和框架集文档类型（Frameset）来触发：

```
<!-- HTML 4.01 Transitional -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<!-- HTML 4.01 Frameset -->
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">

<!-- XHTML 1.0 Transitional -->
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- XHTML 1.0 Frameset -->
<!DOCTYPE html PUBLIC
```

```
  "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

准标准模式与标准模式非常接近，很少需要区分。人们在说到“标准模式”时，可能指其中任何一个。而对文档模式的检测（本书后面会讨论）也不会区分它们。本书后面所说的**标准模式**，指的就是除混杂模式以外的模式。

2.4 <noscript>元素

针对早期浏览器不支持JavaScript的问题，需要一个页面优雅降级的处理方案。最终，<noscript>元素出现，被用于给不支持JavaScript的浏览器提供替代内容。虽然如今的浏览器已经100%支持JavaScript，但对于禁用JavaScript的浏览器来说，这个元素仍然有它的用处。

<noscript>元素可以包含任何可以出现在<body>中的HTML元素，<script>除外。在下列两种情况下，浏览器将显示包含在<noscript>中的内容：

- 浏览器不支持脚本；
- 浏览器对脚本的支持被关闭。

任何一个条件被满足，包含在<noscript>中的内容就会被渲染。否则，浏览器不会渲染<noscript>中的内容。

下面是一个例子：

```
<!DOCTYPE html>
<html>
  <head>
  <title>Example HTML Page</title>
  <script ""defer="defer" src="example1.js"></script>
  <script ""defer="defer" src="example2.js"></script>
  </head>
  <body>
  <noscript>
    <p>This page requires a JavaScript-enabled browser.</p>
  </noscript>
  </body>
</html>
```

这个例子是在脚本不可用时让浏览器显示一段话。如果浏览器支持脚本，则用户永远不会看到它。

2.5 小结

JavaScript是通过<script>元素插入到HTML页面中的。这个元素可用于把JavaScript代码嵌入到HTML页面中，跟其他标记混合在一起，也可用于引入保存在外部文件中的JavaScript。本章的重点可以总结如下。

- 要包含外部JavaScript文件，必须将src属性设置为要包含文件的URL。文件可以跟网页在同一台服务器上，也可以位于完全不同的域。
- 所有<script>元素会依照它们在网页中出现的次序被解释。在不使用defer和async属性的情况下，包含在<script>元素中的代码必须严格按次序解释。

- 对不推迟执行的脚本，浏览器必须解释完位于<script>元素中的代码，然后才能继续渲染页面的剩余部分。为此，通常应该把<script>元素放到页面末尾，介于主内容之后及</body>标签之前。
- 可以使用defer属性把脚本推迟到文档渲染完毕后再执行。推迟的脚本总是按照它们被列出的次序执行。
- 可以使用async属性表示脚本不需要等待其他脚本，同时也不阻塞文档渲染，即异步加载。异步脚本不能保证按照它们在页面中出现的次序执行。
- 通过使用<noscript>元素，可以指定在浏览器不支持脚本时显示的内容。如果浏览器支持并启用脚本，则<noscript>元素中的任何内容都不会被渲染。

第 3 章　语言基础

> **本章内容**
>
> - 语法
> - 数据类型
> - 流控制语句
> - 理解函数

任何语言的核心所描述的都是这门语言在最基本的层面上如何工作，涉及语法、操作符、数据类型以及内置功能，在此基础之上才可以构建复杂的解决方案。如前所述，ECMA-262以一个名为ECMAScript的伪语言的形式，定义了JavaScript的所有这些方面。

ECMA-262第5版（ES5）定义的ECMAScript，是目前为止实现得最为广泛（即受浏览器支持最好）的一个版本。第6版（ES6）在浏览器中的实现（即受支持）程度次之。到2017年底，大多数主流浏览器几乎或全部实现了这一版的规范。为此，本章接下来的内容主要基于ECMAScript第6版。

3.1　语法

ECMAScript的语法很大程度上借鉴了C语言和其他类C语言，如Java和Perl。熟悉这些语言的开发者，应该很容易理解ECMAScript宽松的语法。

3.1.1　区分大小写

首先要知道的是，ECMAScript中一切都区分大小写。无论是变量、函数名还是操作符，都区分大小写。换句话说，变量test和变量Test是两个不同的变量。类似地，typeof不能作为函数名，因为它是一个关键字（后面会介绍）。但Typeof是一个完全有效的函数名。

3.1.2　标识符

所谓**标识符**，就是变量、函数、属性或函数参数的名称。标识符可以由一或多个下列字符组成：

- 第一个字符必须是一个字母、下划线（_）或美元符号（$）；
- 剩下的其他字符可以是字母、下划线、美元符号或数字。

标识符中的字母可以是扩展ASCII（Extended ASCII）中的字母，也可以是Unicode的字母字符，如À和Æ（但不推荐使用）。

按照惯例，ECMAScript标识符使用驼峰大小写形式，即第一个单词的首字母小写，后面每个单词的首字母大写，如：

```
firstSecond
myCar
doSomethingImportant
```

虽然这种写法并不是强制性的，但因为这种形式跟ECMAScript内置函数和对象的命名方式一致，所以算是最佳实践。

> **注意** 关键字、保留字、`true`、`false`和`null`不能作为标识符。具体内容请参考3.2节。

3.1.3 注释

ECMAScript采用C语言风格的注释，包括单行注释和块注释。单行注释以两个斜杠字符开头，如：

```
// 单行注释
```

块注释以一个斜杠和一个星号（`/*`）开头，以它们的反向组合（`*/`）结尾，如：

```
/* 这是多行
注释 */
```

3.1.4 严格模式

ECMAScript 5增加了严格模式（strict mode）的概念。严格模式是一种不同的JavaScript解析和执行模型，ECMAScript 3的一些不规范写法在这种模式下会被处理，对于不安全的活动将抛出错误。要对整个脚本启用严格模式，在脚本开头加上这一行：

```
"use strict";
```

虽然看起来像个没有赋值给任何变量的字符串，但它其实是一个预处理指令。任何支持的JavaScript引擎看到它都会切换到严格模式。选择这种语法形式的目的是不破坏ECMAScript 3语法。

也可以单独指定一个函数在严格模式下执行，只要把这个预处理指令放到函数体开头即可：

```
function doSomething() {
  "use strict";
  // 函数体
}
```

严格模式会影响JavaScript执行的很多方面，因此本书在用到它时会明确指出来。所有现代浏览器都支持严格模式。

3.1.5 语句

ECMAScript中的语句以分号结尾。省略分号意味着由解析器确定语句在哪里结尾，如下面的例子所示：

```
let sum = a + b      // 没有分号也有效，但不推荐
let diff = a - b;    // 加分号有效，推荐
```

即使语句末尾的分号不是必需的，也应该加上。记着加分号有助于防止省略造成的问题，比如可以避免输入内容不完整。此外，加分号也便于开发者通过删除空行来压缩代码（如果没有结尾的分号，只删除空行，则会导致语法错误）。加分号也有助于在某些情况下提升性能，因为解析器会尝试在合适的位置补上分号以纠正语法错误。

多条语句可以合并到一个C语言风格的代码块中。代码块由一个左花括号（{）标识开始，一个右花括号（}）标识结束：

```
if (test) {
  test = false;
  console.log(test);
}
```

if之类的控制语句只在执行多条语句时要求必须有代码块。不过，最佳实践是始终在控制语句中使用代码块，即使要执行的只有一条语句，如下例所示：

```
// 有效，但容易导致错误，应该避免
if (test)
  console.log(test);

// 推荐
if (test) {  console.log(test);
}
```

在控制语句中使用代码块可以让内容更清晰，在需要修改代码时也可以减少出错的可能性。

3.2 关键字与保留字

ECMA-262描述了一组保留的**关键字**，这些关键字有特殊用途，比如表示控制语句的开始和结束，或者执行特定的操作。按照规定，保留的关键字不能用作标识符或属性名。ECMA-262第6版规定的所有关键字如下：

```
break       do          in            typeof
case        else        instanceof    var
catch       export      new           void
class       extends     return        while
const       finally     super         with
continue    for         switch        yield
debugger    function    this
default     if          throw
delete      import      try
```

规范中也描述了一组**未来的保留字**，同样不能用作标识符或属性名。虽然保留字在语言中没有特定用途，但它们是保留给将来做关键字用的。

以下是ECMA-262第6版为将来保留的所有词汇。

```
始终保留：

enum


严格模式下保留：

implements  package     public
interface   protected   static
let         private


模块代码中保留：

await
```

这些词汇不能用作标识符，但现在还可以用作对象的属性名。一般来说，最好还是不要使用关键字和保留字作为标识符和属性名，以确保兼容过去和未来的ECMAScript版本。

3.3 变量

ECMAScript变量是松散类型的，意思是变量可以用于保存任何类型的数据。每个变量只不过是一个用于保存任意值的命名占位符。有3个关键字可以声明变量：`var`、`const`和`let`。其中，`var`在ECMAScript的所有版本中都可以使用，而`const`和`let`只能在ECMAScript 6及更晚的版本中使用。

3.3.1 `var`关键字

要定义变量，可以使用`var`操作符（注意`var`是一个关键字），后跟变量名（即标识符，如前所述）：

```
var message;
```

这行代码定义了一个名为`message`的变量，可以用它保存任何类型的值。（不初始化的情况下，变量会保存一个特殊值`undefined`，下一节讨论数据类型时会谈到。）ECMAScript实现变量初始化，因此可以同时定义变量并设置它的值：

```
var message = "hi";
```

这里，`message`被定义为一个保存字符串值`hi`的变量。像这样初始化变量不会将它标识为字符串类型，只是一个简单的赋值而已。随后，不仅可以改变保存的值，也可以改变值的类型：

```
var message = "hi";
message = 100;  // 合法，但不推荐
```

在这个例子中，变量message首先被定义为一个保存字符串值hi的变量，然后又被重写为保存了数值100。虽然不推荐改变变量保存值的类型，但这在ECMAScript中是完全有效的。

1. **var声明作用域**

   关键的问题在于，使用var操作符定义的变量会成为包含它的函数的局部变量。比如，使用var在一个函数内部定义一个变量，就意味着该变量将在函数退出时被销毁：

   ```
   function test() {
     var message = "hi"; // 局部变量
   }
   test();
   console.log(message); // 出错!
   ```

   这里，message变量是在函数内部使用var定义的。函数叫test()，调用它会创建这个变量并给它赋值。调用之后变量随即被销毁，因此示例中的最后一行会导致错误。不过，在函数内定义变量时省略var操作符，可以创建一个全局变量：

   ```
   function test() {
     message = "hi";     // 全局变量
   }
   test();
   console.log(message); // "hi"
   ```

   去掉之前的var操作符之后，message就变成了全局变量。只要调用一次函数test()，就会定义这个变量，并且可以在函数外部访问到。

   > **注意** 虽然可以通过省略var操作符定义全局变量，但不推荐这么做。在局部作用域中定义的全局变量很难维护，也会造成困惑。这是因为不能一下子断定省略var是不是有意而为之。在严格模式下，如果像这样给未声明的变量赋值，则会导致抛出ReferenceError。

   如果需要定义多个变量，可以在一条语句中用逗号分隔每个变量（及可选的初始化）：

   ```
   var message = "hi",
       found = false,
       age = 29;
   ```

   这里定义并初始化了3个变量。因为ECMAScript是松散类型的，所以使用不同数据类型初始化的变量可以用一条语句来声明。插入换行和空格缩进并不是必需的，但这样有利于阅读理解。

   在严格模式下，不能定义名为eval和arguments的变量，否则会导致语法错误。

2. **var声明提升**

使用var时，下面的代码不会报错。这是因为使用这个关键字声明的变量会自动提升到函数作用域顶部：

```
function foo() {
  console.log(age);
  var age = 26;
}
foo();  // undefined
```

之所以不会报错，是因为ECMAScript运行时把它看成等价于如下代码：

```
function foo() {
  var age;
  console.log(age);
  age = 26;
}
foo();  // undefined
```

这就是所谓的“提升”（hoist），也就是把所有变量声明都拉到函数作用域的顶部。此外，反复多次使用var声明同一个变量也没有问题：

```
function foo() {
  var age = 16;
  var age = 26;
  var age = 36;
  console.log(age);
}
foo();  // 36
```

3.3.2 let声明

let跟var的作用差不多，但有着非常重要的区别。最明显的区别是，let声明的范围是块作用域，而var声明的范围是函数作用域。

```
if (true) {
  var name = 'Matt';
  console.log(name); // Matt
}
console.log(name);   // Matt

if (true) {
  let age = 26;
  console.log(age);   // 26
}
console.log(age);     // ReferenceError: age没有定义
```

在这里，age变量之所以不能在if块外部被引用，是因为它的作用域仅限于该块内部。块作用域是函数作用域的子集，因此适用于var的作用域限制同样也适用于let。

let也不允许同一个块作用域中出现冗余声明。这样会导致报错：

```
var name;
var name;

let age;
let age;  // SyntaxError；标识符age已经声明过了
```

当然，JavaScript引擎会记录用于变量声明的标识符及其所在的块作用域，因此嵌套使用相同的标识符不会报错，而这是因为同一个块中没有重复声明：

```
var name = 'Nicholas';
console.log(name);    // 'Nicholas'
if (true) {
  var name = 'Matt';
  console.log(name);  // 'Matt'
}

let age = 30;
console.log(age);    // 30
if (true) {
  let age = 26;
  console.log(age);  // 26
}
```

对声明冗余报错不会因混用let和var而受影响。这两个关键字声明的并不是不同类型的变量，它们只是指出变量在相关作用域如何存在。

```
var name;
let name; // SyntaxError

let age;
var age; // SyntaxError
```

1. **暂时性死区**

   let与var的另一个重要的区别，就是let声明的变量不会在作用域中被提升。

   ```
   // name会被提升
   console.log(name); // undefined
   var name = 'Matt';
   ```

```
// age不会被提升
console.log(age); // ReferenceError：age没有定义
let age = 26;
```

在解析代码时，JavaScript引擎也会注意出现在块后面的let声明，只不过在此之前不能以任何方式来引用未声明的变量。在let声明之前的执行瞬间被称为“暂时性死区”（temporal dead zone），在此阶段引用任何后面才声明的变量都会抛出ReferenceError。

2. **全局声明**

与var关键字不同，使用let在全局作用域中声明的变量不会成为window对象的属性（var声明的变量则会）。

```
var name = 'Matt';
console.log(window.name); // 'Matt'

let age = 26;
console.log(window.age);  // undefined
```

不过，let声明仍然是在全局作用域中发生的，相应变量会在页面的生命周期内存续。因此，为了避免SyntaxError，必须确保页面不会重复声明同一个变量。

3. **条件声明**

在使用var声明变量时，由于声明会被提升，JavaScript引擎会自动将多余的声明在作用域顶部合并为一个声明。因为let的作用域是块，所以不可能检查前面是否已经使用let声明过同名变量，同时也就不可能在没有声明的情况下声明它。

```
<script>
  var name = 'Nicholas';
  let age = 26;
</script>

<script>
  // 假设脚本不确定页面中是否已经声明了同名变量
  // 那它可以假设还没有声明过

  var name = 'Matt';
  // 这里没问题，因为可以被作为一个提升声明来处理
  // 不需要检查之前是否声明过同名变量

  let age = 36;
  // 如果age之前声明过，这里会报错
</script>
```

使用try/catch语句或typeof操作符也不能解决，因为条件块中let声明的作用域仅限于该块。

```
<script>
  let name = 'Nicholas';
  let age = 36;
</script>

<script>
  // 假设脚本不确定页面中是否已经声明了同名变量
  // 那它可以假设还没有声明过

  if (typeof name === 'undefined') {
    let name;
  }
  // name被限制在if {} 块的作用域内
  // 因此这个赋值形同全局赋值
  name = 'Matt';

  try (age) {
    // 如果age没有声明过，则会报错
  }
  catch(error) {
    let age;
  }
  // age被限制在catch {}块的作用域内
  // 因此这个赋值形同全局赋值
  age = 26;
</script>
```

为此，对于let这个新的ES6声明关键字，不能依赖条件声明模式。

> **注意** 不能使用let进行条件式声明是件好事，因为条件声明是一种反模式，它让程序变得更难理解。如果你发现自己在使用这个模式，那一定有更好的替代方式。

4. **for循环中的let声明**

在let出现之前，for循环定义的迭代变量会渗透到循环体外部：

```
for (var i = 0; i < 5; ++i) {
  // 循环逻辑
}
console.log(i); // 5
```

改成使用let之后，这个问题就消失了，因为迭代变量的作用域仅限于for循环块内部：

```
for (let i = 0; i < 5; ++i) {
  // 循环逻辑
}
console.log(i); // ReferenceError: i没有定义
```

在使用var的时候，最常见的问题就是对迭代变量的奇特声明和修改：

```
for (var i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// 你可能以为会输出0、1、2、3、4
// 实际上会输出5、5、5、5、5
```

之所以会这样，是因为在退出循环时，迭代变量保存的是导致循环退出的值：5。在之后执行超时逻辑时，所有的i都是同一个变量，因而输出的都是同一个最终值。

而在使用let声明迭代变量时，JavaScript引擎在后台会为每个迭代循环声明一个新的迭代变量。每个setTimeout引用的都是不同的变量实例，所以console.log输出的是我们期望的值，也就是循环执行过程中每个迭代变量的值。

```
for (let i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// 会输出0、1、2、3、4
```

这种每次迭代声明一个独立变量实例的行为适用于所有风格的for循环，包括for-in和for-of循环。

3.3.3 const声明

const的行为与let基本相同，唯一一个重要的区别是用它声明变量时必须同时初始化变量，且尝试修改const声明的变量会导致运行时错误。

```
const age = 26;
age = 36; // TypeError: 给常量赋值
// const也不允许重复声明
const name = 'Matt';
const name = 'Nicholas'; // SyntaxError

// const声明的作用域也是块
const name = 'Matt';
if (true) {
  const name = 'Nicholas';
}
console.log(name); // Matt
```

const声明的限制只适用于它指向的变量的引用。换句话说，如果const变量引用的是一个对象，那么修改这个对象内部的属性并不违反const的限制。

```
const person = {};
person.name = 'Matt';  // ok
```

即使JavaScript引擎会为for循环中的let声明分别创建独立的变量实例，而且const变量跟let变量很相似，也不能用const来声明迭代变量（因为迭代变量会自增）：

```
for (const i = 0; i < 10; ++i) {} // TypeError: 给常量赋值
```

不过，如果你只想用const声明一个不会被修改的for循环变量，那也是可以的。也就是说，每次迭代只是创建一个新变量。这对for-of和for-in循环特别有意义：

```
let i = 0;
for (const j = 7; i < 5; ++i) {
  console.log(j);
}
// 7, 7, 7, 7, 7

for (const key in {a: 1, b: 2}) {
  console.log(key);
}
// a, b

for (const value of [1,2,3,4,5]) {
  console.log(value);
}
// 1, 2, 3, 4, 5
```

3.3.4 声明风格及最佳实践

ECMAScript 6增加let和const从客观上为这门语言更精确地声明作用域和语义提供了更好的支持。行为怪异的var所造成的各种问题，已经让JavaScript社区为之苦恼了很多年。随着这两个新关键字的出现，新的有助于提升代码质量的最佳实践也逐渐显现。

1. **不使用var**

   有了let和const，大多数开发者会发现自己不再需要var了。限制自己只使用let和const有助于提升代码质量，因为变量有了明确的作用域、声明位置，以及不变的值。

2. **const优先，let次之**

   使用const声明可以让浏览器运行时强制保持变量不变，也可以让静态代码分析工具提前发现不合法的赋值操作。因此，很多开发者认为应该优先使用const来声明变量，只在提前知道未来会有修改时，再使用let。这样可以让开发者更有信心地推断某些变量的值永远不会变，同时也能迅速发现因意外赋值导致的非预期行为。

3.4 数据类型

ECMAScript有6种简单数据类型（也称为**原始类型**）：Undefined、Null、Boolean、Number、String和Symbol。Symbol（符号）是ECMAScript 6新增的。还有一种复杂数据类型叫Object（对象）。Object是一种无序名值对的集合。因为在ECMAScript中不能定义自己的数据类型，所有值都可以用上述7种数据类型之一来

表示。只有7种数据类型似乎不足以表示全部数据。但ECMAScript的数据类型很灵活，一种数据类型可以当作多种数据类型来使用。

3.4.1 typeof操作符

因为ECMAScript的类型系统是松散的，所以需要一种手段来确定任意变量的数据类型。typeof操作符就是为此而生的。对一个值使用typeof操作符会返回下列字符串之一：

- "undefined"表示值未定义；
- "boolean"表示值为布尔值；
- "string"表示值为字符串；
- "number"表示值为数值；
- "object"表示值为对象（而不是函数）或null；
- "function"表示值为函数；
- "symbol"表示值为符号。

下面是使用typeof操作符的例子：

```
let message = "some string";
console.log(typeof message);    // "string"
console.log(typeof(message));   // "string"
console.log(typeof 95);         // "number"
```

在这个例子中，我们把一个变量（message）和一个数值字面量传给了typeof操作符。注意，因为typeof是一个操作符而不是函数，所以不需要参数（但可以使用参数）。

注意typeof在某些情况下返回的结果可能会让人费解，但技术上讲还是正确的。比如，调用typeof null返回的是"object"。这是因为特殊值null被认为是一个对空对象的引用。

> **注意** 严格来讲，函数在ECMAScript中被认为是对象，并不代表一种数据类型。可是，函数也有自己特殊的属性。为此，就有必要通过typeof操作符来区分函数和其他对象。

3.4.2 Undefined类型

Undefined类型只有一个值，就是特殊值undefined。当使用var或let声明了变量但没有初始化时，就相当于给变量赋予了undefined值：

```
let message;
console.log(message == undefined); // true
```

在这个例子中，变量message在声明的时候并未初始化。而在比较它和undefined的字面值时，两者是相等的。这个例子等同于如下示例：

```
let message = undefined;
console.log(message == undefined); // true
```

这里，变量message显式地以undefined来初始化。但这是不必要的，因为默认情况下，任何未经初始化的变量都会取得undefined值。

> **注意** 一般来说，永远不用显式地给某个变量设置undefined值。字面值undefined主要用于比较，而且在ECMA-262第3版之前是不存在的。增加这个特殊值的目的就是为了正式明确空对象指针（null）和未初始化变量的区别。

注意，包含undefined值的变量跟未定义变量是有区别的。请看下面的例子：

```
let message;    // 这个变量被声明了，只是值为undefined

// 确保没有声明过这个变量
// let age

console.log(message); // "undefined"
console.log(age);     // 报错
```

在上面的例子中，第一个console.log会指出变量message的值，即"undefined"。而第二个console.log要输出一个未声明的变量age的值，因此会导致报错。对未声明的变量，只能执行一个有用的操作，就是对它调用typeof。（对未声明的变量调用delete也不会报错，但这个操作没什么用，实际上在严格模式下会抛出错误。）

在对未初始化的变量调用typeof时，返回的结果是"undefined"，但对未声明的变量调用它时，返回的结果还是"undefined"，这就有点让人看不懂了。比如下面的例子：

```
let message; // 这个变量被声明了，只是值为undefined

// make sure this variable isn't declared
// let age

console.log(typeof message); // "undefined"
console.log(typeof age);     // "undefined"
```

无论是声明还是未声明，typeof返回的都是字符串"undefined"。逻辑上讲这是对的，因为虽然严格来讲这两个变量存在根本性差异，但它对任何一个变量都不可能执行什么真正的操作。

> **注意** 即使未初始化的变量会被自动赋予undefined值，但我们仍然建议在声明变量的同时进行初始化。这样，当typeof返回"undefined"时，你就会知道那是因为给定的变量尚未声明，而不是声明了但未初始化。

undefined是一个假值。因此，如果需要，可以用更简洁的方式检测它。不过要记住，也有很多其他可能的值同样是假值。所以一定要明确自己想检测的就是undefined这个字面值，而不仅仅是假值。

```
let message; // 这个变量被声明了，只是值为undefined
// age没有声明

if (message) {
```

```
  // 这个块不会执行
}

if (!message) {
  // 这个块会执行
}

if (age) {
  // 这里会报错
}
```

3.4.3 Null类型

Null类型同样只有一个值，即特殊值null。逻辑上讲，null值表示一个空对象指针，这也是给typeof传一个null会返回"object"的原因：

```
let car = null;
console.log(typeof car);  // "object"
```

在定义将来要保存对象值的变量时，建议使用null来初始化，不要使用其他值。这样，只要检查这个变量的值是不是null就可以知道这个变量是否在后来被重新赋予了一个对象的引用，比如：

```
if (car != null) {
  // car是一个对象的引用
}
```

undefined值是由null值派生而来的，因此ECMA-262将它们定义为表面上相等，如下面的例子所示：

```
console.log(null == undefined);  // true
```

用等于操作符（==）比较null和undefined始终返回true。但要注意，这个操作符会为了比较而转换它的操作数（本章后面将详细介绍）。

即使null和undefined有关系，它们的用途也是完全不一样的。如前所述，永远不必显式地将变量值设置为undefined。但null不是这样的。任何时候，只要变量要保存对象，而当时又没有那个对象可保存，就要用null来填充该变量。这样就可以保持null是空对象指针的语义，并进一步将其与undefined区分开来。

null是一个假值。因此，如果需要，可以用更简洁的方式检测它。不过要记住，也有很多其他可能的值同样是假值。所以一定要明确自己想检测的就是null这个字面值，而不仅仅是假值。

```
let message = null;
let age;

if (message) {
```

```
  // 这个块不会执行
}

if (!message) {
  // 这个块会执行
}

if (age) {
  // 这个块不会执行
}

if (!age) {
  // 这个块会执行
}
```

3.4.4 Boolean类型

Boolean（布尔值）类型是ECMAScript中使用最频繁的类型之一，有两个字面值：true和false。这两个布尔值不同于数值，因此true不等于1，false不等于0。下面是给变量赋布尔值的例子：

```
let found = true;
let lost = false;
```

注意，布尔值字面量true和false是区分大小写的，因此True和False（及其他大小混写形式）是有效的标识符，但不是布尔值。

虽然布尔值只有两个，但所有其他ECMAScript类型的值都有相应布尔值的等价形式。要将一个其他类型的值转换为布尔值，可以调用特定的Boolean()转型函数：

```
let message = "Hello world!";
let messageAsBoolean = Boolean(message);
```

在这个例子中，字符串message会被转换为布尔值并保存在变量messageAsBoolean中。Boolean()转型函数可以在任意类型的数据上调用，而且始终返回一个布尔值。什么值能转换为true或false的规则取决于数据类型和实际的值。下表总结了不同类型与布尔值之间的转换规则。

数据类型	转换为true的值	转换为false的值
Boolean	true	false
String	非空字符串	""（空字符串）
Number	非零数值（包括无穷值）	0、NaN（参见后面的相关内容）
Object	任意对象	null
Undefined	N/A（不存在）	undefined

理解以上转换非常重要，因为像if等流控制语句会自动执行其他类型值到布尔值的转换，例如：

```
let message = "Hello world!";
if (message) {
  console.log("Value is true");
}
```

在这个例子中，console.log会输出字符串"Value is true"，因为字符串message会被自动转换为等价的布尔值true。由于存在这种自动转换，理解流控制语句中使用的是什么变量就非常重要。错误地使用对象而不是布尔值会明显改变应用程序的执行流。

3.4.5 Number类型

ECMAScript中最有意思的数据类型或许就是Number了。Number类型使用IEEE 754格式表示整数和浮点值（在某些语言中也叫双精度值）。不同的数值类型相应地也有不同的数值字面量格式。

最基本的数值字面量格式是十进制整数，直接写出来即可：

```
let intNum = 55;  // 整数
```

整数也可以用八进制（以8为基数）或十六进制（以16为基数）字面量表示。对于八进制字面量，第一个数字必须是零（0），然后是相应的八进制数字（数值0~7）。如果字面量中包含的数字超出了应有的范围，就会忽略前缀的零，后面的数字序列会被当成十进制数，如下所示：

```
let octalNum1 = 070;  // 八进制的56
let octalNum2 = 079;  // 无效的八进制值，当成79处理
let octalNum3 = 08;   // 无效的八进制值，当成8处理
```

八进制字面量在严格模式下是无效的，会导致JavaScript引擎抛出语法错误。**1**

1ECMAScript 2015或ES6中的八进制值通过前缀0o来表示；严格模式下，前缀0会被视为语法错误，如果要表示八进制值，应该使用前缀0o。——译者注

要创建十六进制字面量，必须让真正的数值前缀0x（区分大小写），然后是十六进制数字（0~9以及A~F）。十六进制数字中的字母大小写均可。下面是几个例子：

```
let hexNum1 = 0xA;   // 十六进制10
let hexNum2 = 0x1f;  // 十六进制31
```

使用八进制和十六进制格式创建的数值在所有数学操作中都被视为十进制数值。

> **注意** 由于JavaScript保存数值的方式，实际中可能存在正零（+0）和负零（-0）。正零和负零在所有情况下都被认为是等同的，这里特地说明一下。

1. **浮点值**

要定义浮点值，数值中必须包含小数点，而且小数点后面必须至少有一个数字。虽然小数点前面不是必须有整数，但推荐加上。下面是几个例子：

```
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum3 = .1;   // 有效，但不推荐
```

因为存储浮点值使用的内存空间是存储整数值的两倍，所以ECMAScript总是想方设法把值转换为整数。在小数点后面没有数字的情况下，数值就会变成整数。类似地，如果数值本身就是整数，只是小数点后面跟着0（如1.0），那它也会被转换为整数，如下例所示：

```
let floatNum1 = 1.;   // 小数点后面没有数字，当成整数1处理
let floatNum2 = 10.0; // 小数点后面是零，当成整数10处理
```

对于非常大或非常小的数值，浮点值可以用科学记数法来表示。科学记数法用于表示一个应该乘以10的给定次幂的数值。ECMAScript中科学记数法的格式要求是一个数值（整数或浮点数）后跟一个大写或小写的字母e，再加上一个要乘的10的多少次幂。比如：

```
let floatNum = 3.125e7; // 等于31250000
```

在这个例子中，`floatNum`等于31 250 000，只不过科学记数法显得更简洁。这种表示法实际上相当于说：“以3.125作为系数，乘以10的7次幂。”

科学记数法也可以用于表示非常小的数值，例如0.000 000 000 000 000 03。这个数值用科学记数法可以表示为3e-17。默认情况下，ECMAScript会将小数点后至少包含6个零的浮点值转换为科学记数法（例如，0.000 000 3会被转换为3e-7）。

浮点值的精确度最高可达17位小数，但在算术计算中远不如整数精确。例如，0.1加0.2得到的不是0.3，而是0.300 000 000 000 000 04。由于这种微小的舍入错误，导致很难测试特定的浮点值。比如下面的例子：

```
if (a + b == 0.3) {       // 别这么干!
  console.log("You got 0.3.");
}
```

这里检测两个数值之和是否等于0.3。如果两个数值分别是0.05和0.25，或者0.15和0.15，那没问题。但如果是0.1和0.2，如前所述，测试将失败。因此永远不要测试某个特定的浮点值。

> **注意** 之所以存在这种舍入错误，是因为使用了IEEE 754数值，这种错误并非ECMAScript所独有。其他使用相同格式的语言也有这个问题。

2. **值的范围**

由于内存的限制，ECMAScript并不支持表示这个世界上的所有数值。ECMAScript可以表示的最小数值保存在Number.MIN_VALUE中，这个值在多数浏览器中是5e-324；可以表示的最大数值保存在Number.MAX_VALUE中，这个值在多数浏览器中是1.797 693 134 862 315 7e+308。如果某个计算得到的数值结果超出了JavaScript可以表示的范围，那么这个数值会被自动转换为一个特殊的Infinity（无穷）值。任何无法表示的负数以-Infinity（负无穷大）表示，任何无法表示的正数以Infinity（正无穷大）表示。

如果计算返回正Infinity或负Infinity，则该值将不能再进一步用于任何计算。这是因为Infinity没有可用于计算的数值表示形式。要确定一个值是不是有限大（即介于JavaScript能表示的最小值和最大值之间），可以使用isFinite()函数，如下所示：

```
let result = Number.MAX_VALUE + Number.MAX_VALUE;
console.log(isFinite(result));  // false
```

虽然超出有限数值范围的计算并不多见，但总归还是有可能的。因此在计算非常大或非常小的数值时，有必要监测一下计算结果是否超出范围。

> **注意** 使用Number.NEGATIVE_INFINITY和Number.POSITIVE_INFINITY也可以获取正、负Infinity。没错，这两个属性包含的值分别就是-Infinity和Infinity。

3. **NaN**

有一个特殊的数值叫NaN，意思是“不是数值”（Not a Number），用于表示本来要返回数值的操作失败了（而不是抛出错误）。比如，用0除任意数值在其他语言中通常都会导致错误，从而中止代码执行。但在ECMAScript中，0、+0或-0相除会返回NaN：

```
console.log(0/0);    // NaN
console.log(-0/+0);  // NaN
```

如果分子是非0值，分母是有符号0或无符号0，则会返回Infinity或-Infinity：

```
console.log(5/0);   // Infinity
console.log(5/-0);  // -Infinity
```

NaN有几个独特的属性。首先，任何涉及NaN的操作始终返回NaN（如NaN/10），在连续多步计算时这可能是个问题。其次，NaN不等于包括NaN在内的任何值。例如，下面的比较操作会返回false：

```
console.log(NaN == NaN); // false
```

为此，ECMAScript提供了isNaN()函数。该函数接收一个参数，可以是任意数据类型，然后判断这个参数是否“不是数值”。把一个值传给isNaN()后，该函数会尝试把它转换为数值。某些非数值的值可以直接转换成数值，如字符串"10"或布尔值。任何不能转换为数值的值都会导致这个函数返回true。举例如下：

```
console.log(isNaN(NaN));      // true
console.log(isNaN(10));       // false，10是数值
console.log(isNaN("10"));     // false，可以转换为数值10
console.log(isNaN("blue"));   // true，不可以转换为数值
console.log(isNaN(true));     // false，可以转换为数值1
```

上述的例子测试了5个不同的值。首先测试的是NaN本身，显然会返回true。接着测试了数值10和字符串"10"，都返回false，因为它们的数值都是10。字符串"blue"不能转换为数值，因此函数返回true。布尔值true可以转换为数值1，因此返回false。

> **注意** 虽然不常见，但isNaN()可以用于测试对象。此时，首先会调用对象的valueOf()方法，然后再确定返回的值是否可以转换为数值。如果不能，再调用toString()方法，并测试其返回值。这通常是ECMAScript内置函数和操作符的工作方式，本章后面会讨论。

4. **数值转换**

有3个函数可以将非数值转换为数值：Number()、parseInt()和parseFloat()。Number()是转型函数，可用于任何数据类型。后两个函数主要用于将字符串转换为数值。对于同样的参数，这3个函数执行的操作也不同。

Number()函数基于如下规则执行转换。

- 布尔值，true转换为1，false转换为0。
- 数值，直接返回。
- null，返回0。
- undefined，返回NaN。
- 字符串，应用以下规则。
  - 如果字符串包含数值字符，包括数值字符前面带加、减号的情况，则转换为一个十进制数值。因此，Number("1")返回1，Number("123")返回123，Number("011")返回11（忽略前面的零）。
  - 如果字符串包含有效的浮点值格式如"1.1"，则会转换为相应的浮点值（同样，忽略前面的零）。
  - 如果字符串包含有效的十六进制格式如"0xf"，则会转换为与该十六进制值对应的十进制整数值。
  - 如果是空字符串（不包含字符），则返回0。
  - 如果字符串包含除上述情况之外的其他字符，则返回NaN。
- 对象，调用valueOf()方法，并按照上述规则转换返回的值。如果转换结果是NaN，则调用toString()方法，再按照转换字符串的规则转换。

从不同数据类型到数值的转换有时候会比较复杂，看一看Number()的转换规则就知道了。下面是几个具体的例子：

```
let num1 = Number("Hello world!");  // NaN
let num2 = Number("");              // 0
let num3 = Number("000011");        // 11
let num4 = Number(true);            // 1
```

可以看到，字符串"Hello world"转换之后是NaN，因为它找不到对应的数值。空字符串转换后是0。字符串000011转换后是11，因为前面的零被忽略了。最后，true转换为1。

> **注意** 本章后面会讨论到的一元加操作符与Number()函数遵循相同的转换规则。

考虑到用Number()函数转换字符串时相对复杂且有点反常规，通常在需要得到整数时可以优先使用parseInt()函数。parseInt()函数更专注于字符串是否包含数值模式。字符串最前面的空格会被忽略，从第一个非空格字符开始转换。如果第一个字符不是数值字符、加号或减号，parseInt()立即返回NaN。这意味着空字符串也会返回NaN（这一点跟Number()不一样，它返回0）。如果第一个字符是数值字符、加号或减号，则继续依次检测每个字符，直到字符串末尾，或碰到非数值字符。比如，"1234blue"会被转换为1234，因为"blue"会被完全忽略。类似地，"22.5"会被转换为22，因为小数点不是有效的整数字符。

假设字符串中的第一个字符是数值字符，parseInt()函数也能识别不同的整数格式（十进制、八进制、十六进制）。换句话说，如果字符串以"0x"开头，就会被解释为十六进制整数。如果字符串以"0"开头，且紧跟着数值字符，就会被解释为八进制整数。

下面几个转换示例有助于理解上述规则：

```
let num1 = parseInt("1234blue");  // 1234
let num2 = parseInt("");          // NaN
let num3 = parseInt("0xA");       // 10，解释为十六进制整数
let num4 = parseInt(22.5);        // 22
let num5 = parseInt("70");        // 70，解释为十进制值
let num6 = parseInt("0xf");       // 15，解释为十六进制整数
```

不同的数值格式很容易混淆，因此parseInt()也接收第二个参数，用于指定底数（进制数）。如果知道要解析的值是十六进制，那么可以传入16作为第二个参数，以便正确解析：

```
let num = parseInt("0xAF", 16); // 175
```

事实上，如果提供了十六进制参数，那么字符串前面的"0x"可以省掉：

```
let num1 = parseInt("AF", 16);  // 175
let num2 = parseInt("AF");      // NaN
```

在这个例子中，第一个转换是正确的，而第二个转换失败了。区别在于第一次传入了进制数作为参数，告诉parseInt()要解析的是一个十六进制字符串。而第二个转换检测到第一个字符就是非数值字符，随即自动停止并返回NaN。

通过第二个参数，可以极大扩展转换后获得的结果类型。比如：

```
let num1 = parseInt("10", 2);   // 2，按二进制解析
let num2 = parseInt("10", 8);   // 8，按八进制解析
```

```
let num3 = parseInt("10", 10);  // 10，按十进制解析
let num4 = parseInt("10", 16);  // 16，按十六进制解析
```

因为不传底数参数相当于让parseInt()自己决定如何解析，所以为避免解析出错，建议始终传给它第二个参数。

> **注意** 多数情况下解析的应该都是十进制数，此时第二个参数就要传入10。

parseFloat()函数的工作方式跟parseInt()函数类似，都是从位置0开始检测每个字符。同样，它也是解析到字符串末尾或者解析到一个无效的浮点数值字符为止。这意味着第一次出现的小数点是有效的，但第二次出现的小数点就无效了，此时字符串的剩余字符都会被忽略。因此，"22.34.5"将转换成22.34。

parseFloat()函数的另一个不同之处在于，它始终忽略字符串开头的零。这个函数能识别前面讨论的所有浮点格式，以及十进制格式（开头的零始终被忽略）。十六进制数值始终会返回0。因为parseFloat()只解析十进制值，因此不能指定底数。最后，如果字符串表示整数（没有小数点或者小数点后面只有一个零），则parseFloat()返回整数。下面是几个示例：

```
let num1 = parseFloat("1234blue");  // 1234，按整数解析
let num2 = parseFloat("0xA");       // 0
let num3 = parseFloat("22.5");      // 22.5
let num4 = parseFloat("22.34.5");   // 22.34
let num5 = parseFloat("0908.5");    // 908.5
let num6 = parseFloat("3.125e7");   // 31250000
```

3.4.6 String类型

String（字符串）数据类型表示零或多个16位Unicode字符序列。字符串可以使用双引号（"）、单引号（'）或反引号（`）标示，因此下面的代码都是合法的：

```
let firstName = "John";
let lastName = 'Jacob';
let lastName = `Jingleheimerschmidt`
```

跟某些语言中使用不同的引号会改变对字符串的解释方式不同，ECMAScript语法中表示字符串的引号没有区别。不过要注意的是，以某种引号作为字符串开头，必须仍然以该种引号作为字符串结尾。比如，下面的写法会导致语法错误：

```
let firstName = 'Nicholas"; // 语法错误：开头和结尾的引号必须是同一种
```

1. **字符字面量**

   字符串数据类型包含一些字符字面量，用于表示非打印字符或有其他用途的字符，如下表所示：

字面量	含义
\n	换行
\t	制表
\b	退格
\r	回车
\f	换页
\\\\	反斜杠（\）
\'	单引号（'），在字符串以单引号标示时使用，例如'He said, \'hey.\''
\"	双引号（"），在字符串以双引号标示时使用，例如"He said, \"hey.\""
\\\`` | 反引号（`），在字符串以反引号标示时使用，例如`He said, \`hey.\```	
\x*nn*	以十六进制编码*nn*表示的字符（其中*n*是十六进制数字0~F），例如\x41等于"A"
\u*nnnn*	以十六进制编码*nnnn*表示的Unicode字符（其中*n*是十六进制数字0~F），例如\u03a3等于希腊字符"Σ"

这些字符字面量可以出现在字符串中的任意位置，且可以作为单个字符被解释：

```
let text = "This is the letter sigma: \u03a3.";
```

在这个例子中，即使包含6个字符长的转义序列，变量text仍然是28个字符长。因为转义序列表示一个字符，所以只算一个字符。

字符串的长度可以通过其length属性获取：

```
console.log(text.length); // 28
```

这个属性返回字符串中16位字符的个数。

> **注意** 如果字符串中包含双字节字符，那么length属性返回的值可能不是准确的字符数。第5章将具体讨论如何解决这个问题。

2. **字符串的特点**

ECMAScript中的字符串是不可变的（immutable），意思是一旦创建，它们的值就不能变了。要修改某个变量中的字符串值，必须先销毁原始的字符串，然后将包含新值的另一个字符串保存到该变量，如下所示：

```
let lang = "Java";
lang = lang + "Script";
```

这里，变量lang一开始包含字符串"Java"。紧接着，lang被重新定义为包含"Java"和"Script"的组合，也就是"JavaScript"。整个过程首先会分配一个足够容纳10个字符的空间，然后填充上"Java"和"Script"。最后销毁原始的字符串"Java"和字符串"Script"，因为这两个字符串都没有用了。所有处理都是在后台发生的，而这也是一些早期的浏览器（如Firefox 1.0之前的版本和IE6.0）在拼接字符串时非常慢的原因。这些浏览器在后来的版本中都有针对性地解决了这个问题。

3. **转换为字符串**

有两种方式把一个值转换为字符串。首先是使用几乎所有值都有的toString()方法。这个方法唯一的用途就是返回当前值的字符串等价物。比如：

```
let age = 11;
let ageAsString = age.toString();       // 字符串"11"
let found = true;
let foundAsString = found.toString();  // 字符串"true"
```

toString()方法可见于数值、布尔值、对象和字符串值。（没错，字符串值也有toString()方法，该方法只是简单地返回自身的一个副本。）null和undefined值没有toString()方法。

多数情况下，toString()不接收任何参数。不过，在对数值调用这个方法时，toString()可以接收一个底数参数，即以什么底数来输出数值的字符串表示。默认情况下，toString()返回数值的十进制字符串表示。而通过传入参数，可以得到数值的二进制、八进制、十六进制，或者其他任何有效基数的字符串表示，比如：

```
let num = 10;
console.log(num.toString());     // "10"
console.log(num.toString(2));    // "1010"
console.log(num.toString(8));    // "12"
console.log(num.toString(10));   // "10"
console.log(num.toString(16));   // "a"
```

这个例子展示了传入底数参数时，toString()输出的字符串值也会随之改变。数值10可以输出为任意数值格式。注意，默认情况下（不传参数）的输出与传入参数10得到的结果相同。

如果你不确定一个值是不是null或undefined，可以使用String()转型函数，它始终会返回表示相应类型值的字符串。String()函数遵循如下规则。

- 如果值有toString()方法，则调用该方法（不传参数）并返回结果。
- 如果值是null，返回"null"。
- 如果值是undefined，返回"undefined"。

下面看几个例子：

```
let value1 = 10;
let value2 = true;
let value3 = null;
let value4;

console.log(String(value1));  // "10"
console.log(String(value2));  // "true"
console.log(String(value3));  // "null"
console.log(String(value4));  // "undefined"
```

这里展示了将4个值转换为字符串的情况：一个数值、一个布尔值、一个null和一个undefined。数值和布尔值的转换结果与调用toString()相同。因为null和undefined没有toString()方法，所以String()方法就直接返回了这两个值的字面量文本。

> **注意** 用加号操作符给一个值加上一个空字符串""也可以将其转换为字符串（加号操作符本章后面会介绍）。

4. **模板字面量**

ECMAScript 6新增了使用模板字面量定义字符串的能力。与使用单引号或双引号不同，模板字面量保留换行字符，可以跨行定义字符串：

```
let myMultiLineString = 'first line\nsecond line';
let myMultiLineTemplateLiteral = `first line
second line`;

console.log(myMultiLineString);
// first line
// second line"

console.log(myMultiLineTemplateLiteral);
// first line
// second line

console.log(myMultiLineString === myMultiLinetemplateLiteral); // true
```

顾名思义，模板字面量在定义模板时特别有用，比如下面这个HTML模板：

```
let pageHTML = `
<div>
  <a href="#">
    <span>Jake</span>
  </a>
</div>`;
```

由于模板字面量会保持反引号内部的空格，因此在使用时要格外注意。格式正确的模板字符串可能会看起来缩进不当：

```
// 这个模板字面量在换行符之后有25个空格符
let myTemplateLiteral = `first line
                         second line`;
console.log(myTemplateLiteral.length);  // 47

// 这个模板字面量以一个换行符开头
let secondTemplateLiteral = `
first line
second line`;
console.log(secondTemplateLiteral[0] === '\n'); // true

// 这个模板字面量没有意料之外的字符
let thirdTemplateLiteral = `first line
second line`;
console.log(thirdTemplateLiteral[0]);
// first line
// second line
```

5. **字符串插值**

模板字面量最常用的一个特性是支持字符串插值，也就是可以在一个连续定义中插入一个或多个值。技术上讲，模板字面量不是字符串，而是一种特殊的JavaScript句法表达式，只不过求值后得到的是字符串。模板字面量在定义时立即求值并转换为字符串实例，任何插入的变量也会从它们最接近的作用域中取值。

字符串插值通过在`${}`中使用一个JavaScript表达式实现：

```
let value = 5;
let exponent = 'second';
// 以前，字符串插值是这样实现的：
let interpolatedString =
  value + ' to the ' + exponent + ' power is ' + (value * value);

// 现在，可以用模板字面量这样实现：
let interpolatedTemplateLiteral =
  `${ value } to the ${ exponent } power is ${ value * value }`;

console.log(interpolatedString);           // 5 to the second power is 25
console.log(interpolatedTemplateLiteral);  // 5 to the second power is 25
```

所有插入的值都会使用`toString()`强制转型为字符串，而且任何JavaScript表达式都可以用于插值。嵌套的模板字符串无须转义：

```
console.log(`Hello, ${ `World` }!`);  // Hello, World!
```

将表达式转换为字符串时会调用`toString()`：

```
let foo = { toString: () => 'World' };
console.log(`Hello, ${ foo }!`);      // Hello, World!
```

在插值表达式中可以调用函数和方法：

```
function capitalize(word) {
  return `${ word[0].toUpperCase() }${ word.slice(1) }`;
}
console.log(`${ capitalize('hello') }, ${ capitalize('world') }!`); //
Hello, World!
```

此外，模板也可以插入自己之前的值：

```
let value = '';
function append() {
  value = `${value}abc`
  console.log(value);
}
append();  // abc
append();  // abcabc
append();  // abcabcabc
```

6. **模板字面量标签函数**

模板字面量也支持定义**标签函数**（tag function），而通过标签函数可以自定义插值行为。标签函数会接收被插值记号分隔后的模板和对每个表达式求值的结果。

标签函数本身是一个常规函数，通过前缀到模板字面量来应用自定义行为，如下例所示。标签函数接收到的参数依次是原始字符串数组和对每个表达式求值的结果。这个函数的返回值是对模板字面量求值得到的字符串。

最好通过一个例子来理解：

```
let a = 6;
let b = 9;

function simpleTag(strings, aValExpression, bValExpression, sumExpression) {
  console.log(strings);
  console.log(aValExpression);
  console.log(bValExpression);
  console.log(sumExpression);

  return 'foobar';
}

let untaggedResult = `${ a } + ${ b } = ${ a + b }`;
```

```
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
// 15

console.log(untaggedResult);   // "6 + 9 = 15"
console.log(taggedResult);     // "foobar"
```

因为表达式参数的数量是可变的，所以通常应该使用剩余操作符（rest operator）将它们收集到一个数组中：

```
let a = 6;
let b = 9;

function simpleTag(strings, ...expressions) {
  console.log(strings);
  for(const expression of expressions) {
    console.log(expression);
  }

  return 'foobar';
}
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
// 15

console.log(taggedResult);  // "foobar"
```

对于有n个插值的模板字面量，传给标签函数的表达式参数的个数始终是n，而传给标签函数的第一个参数所包含的字符串个数则始终是$n + 1$。因此，如果你想把这些字符串和对表达式求值的结果拼接起来作为默认返回的字符串，可以这样做：

```
let a = 6;
let b = 9;

function zipTag(strings, ...expressions) {
  return strings[0] +
         expressions.map((e, i) => `${e}${strings[i + 1]}`)
                    .join('');
}

let untaggedResult =    `${ a } + ${ b } = ${ a + b }`;
let taggedResult = zipTag`${ a } + ${ b } = ${ a + b }`;

console.log(untaggedResult);  // "6 + 9 = 15"
console.log(taggedResult);    // "6 + 9 = 15"
```

7. **原始字符串**

使用模板字面量也可以直接获取原始的模板字面量内容（如换行符或Unicode字符），而不是被转换后的字符表示。为此，可以使用默认的`String.raw`标签函数：

```
// Unicode示例
// \u00A9是版权符号
console.log(`\u00A9`);            // ©
console.log(String.raw`\u00A9`);  // \u00A9

// 换行符示例
console.log(`first line\nsecond line`);
// first line
// second line

console.log(String.raw`first line\nsecond line`); // "first line\nsecond
line"

// 对实际的换行符来说是不行的
// 它们不会被转换成转义序列的形式
console.log(`first line
second line`);
// first line
// second line

console.log(String.raw`first line
second line`);
// first line
// second line
```

另外，也可以通过标签函数的第一个参数，即字符串数组的`.raw`属性取得每个字符串的原始内容：

```
function printRaw(strings) {
  console.log('Actual characters:');
  for (const string of strings) {
    console.log(string);
  }

  console.log('Escaped characters;');
  for (const rawString of strings.raw) {
    console.log(rawString);
  }
}

printRaw`\u00A9${ 'and' }\n`;
// Actual characters:
// ©
// (换行符)
// Escaped characters:
```

```
// \u00A9
// \n
```

3.4.7 Symbol类型

Symbol（符号）是ECMAScript 6新增的数据类型。符号是原始值，且符号实例是唯一、不可变的。符号的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险。

尽管听起来跟私有属性有点类似，但符号并不是为了提供私有属性的行为才增加的（尤其是因为Object API提供了方法，可以更方便地发现符号属性）。相反，符号就是用来创建唯一记号，进而用作非字符串形式的对象属性。

1. **符号的基本用法**

   符号需要使用Symbol()函数初始化。因为符号本身是原始类型，所以typeof操作符对符号返回symbol。

```
let sym = Symbol();
console.log(typeof sym); // symbol
```

   调用Symbol()函数时，也可以传入一个字符串参数作为对符号的描述（description），将来可以通过这个字符串来调试代码。但是，这个字符串参数与符号定义或标识完全无关：

```
let genericSymbol = Symbol();
let otherGenericSymbol = Symbol();

let fooSymbol = Symbol('foo');
let otherFooSymbol = Symbol('foo');

console.log(genericSymbol == otherGenericSymbol);  // false
console.log(fooSymbol == otherFooSymbol);          // false
```

   符号没有字面量语法，这也是它们发挥作用的关键。按照规范，你只要创建Symbol()实例并将其用作对象的新属性，就可以保证它不会覆盖已有的对象属性，无论是符号属性还是字符串属性。

```
let genericSymbol = Symbol();
console.log(genericSymbol);  // Symbol()

let fooSymbol = Symbol('foo');
console.log(fooSymbol);      // Symbol(foo);
```

   最重要的是，Symbol()函数不能用作构造函数，与new关键字一起使用。这样做是为了避免创建符号包装对象，像使用Boolean、String或Number那样，它们都支持构造函数且可用于初始化包含原始值的包装对象：

```
let myBoolean = new Boolean();
console.log(typeof myBoolean); // "object"

let myString = new String();
console.log(typeof myString);  // "object"

let myNumber = new Number();
console.log(typeof myNumber);  // "object"

let mySymbol = new Symbol(); // TypeError: Symbol is not a constructor
```

如果你确实想使用符号包装对象，可以借用`Object()`函数：

```
let mySymbol = Symbol();
let myWrappedSymbol = Object(mySymbol);
console.log(typeof myWrappedSymbol);   // "object"
```

2. **使用全局符号注册表**

如果运行时的不同部分需要共享和重用符号实例，那么可以用一个字符串作为键，在全局符号注册表中创建并重用符号。

为此，需要使用`Symbol.for()`方法：

```
let fooGlobalSymbol = Symbol.for('foo');
console.log(typeof fooGlobalSymbol); // symbol
```

`Symbol.for()`对每个字符串键都执行幂等操作。第一次使用某个字符串调用时，它会检查全局运行时注册表，发现不存在对应的符号，于是就会生成一个新符号实例并添加到注册表中。后续使用相同字符串的调用同样会检查注册表，发现存在与该字符串对应的符号，然后就会返回该符号实例。

```
let fooGlobalSymbol = Symbol.for('foo');       // 创建新符号
let otherFooGlobalSymbol = Symbol.for('foo');  // 重用已有符号

console.log(fooGlobalSymbol === otherFooGlobalSymbol);  // true
```

即使采用相同的符号描述，在全局注册表中定义的符号跟使用`Symbol()`定义的符号也并不等同：

```
let localSymbol = Symbol('foo');
let globalSymbol = Symbol.for('foo');

console.log(localSymbol === globalSymbol); // false
```

全局注册表中的符号必须使用字符串键来创建，因此作为参数传给`Symbol.for()`的任何值都会被转换为字符串。此外，注册表中使用的键同时也会被用作符号描述。

```
let emptyGlobalSymbol = Symbol.for();
console.log(emptyGlobalSymbol);    // Symbol(undefined)
```

还可以使用`Symbol.keyFor()`来查询全局注册表，这个方法接收符号，返回该全局符号对应的字符串键。如果查询的不是全局符号，则返回`undefined`。

```
// 创建全局符号
let s = Symbol.for('foo');
console.log(Symbol.keyFor(s));   // foo

// 创建普通符号
let s2 = Symbol('bar');
console.log(Symbol.keyFor(s2));  // undefined
```

如果传给`Symbol.keyFor()`的不是符号，则该方法抛出`TypeError`：

```
Symbol.keyFor(123); // TypeError: 123 is not a symbol
```

3. **使用符号作为属性**

凡是可以使用字符串或数值作为属性的地方，都可以使用符号。这就包括了对象字面量属性和`Object.defineProperty()`/`Object.definedProperties()`定义的属性。对象字面量只能在计算属性语法中使用符号作为属性。

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar'),
    s3 = Symbol('baz'),
    s4 = Symbol('qux');

let o = {
  [s1]: 'foo val'
};
// 这样也可以：o[s1] = 'foo val';

console.log(o);
// {Symbol(foo): foo val}

Object.defineProperty(o, s2, {value: 'bar val'});

console.log(o);
// {Symbol(foo): foo val, Symbol(bar): bar val}

Object.defineProperties(o, {
```

```
  [s3]: {value: 'baz val'},
  [s4]: {value: 'qux val'}
});

console.log(o);
// {Symbol(foo): foo val, Symbol(bar): bar val,
//  Symbol(baz): baz val, Symbol(qux): qux val}
```

类似于`Object.getOwnPropertyNames()`返回对象实例的常规属性数组，`Object.getOwnPropertySymbols()`返回对象实例的符号属性数组。这两个方法的返回值彼此互斥。`Object.getOwnPropertyDescriptors()`会返回同时包含常规和符号属性描述符的对象。`Reflect.ownKeys()`会返回两种类型的键：

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar');

let o = {
  [s1]: 'foo val',
  [s2]: 'bar val',
  baz: 'baz val',
  qux: 'qux val'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(foo), Symbol(bar)]

console.log(Object.getOwnPropertyNames(o));
// ["baz", "qux"]

console.log(Object.getOwnPropertyDescriptors(o));
// {baz: {...}, qux: {...}, Symbol(foo): {...}, Symbol(bar): {...}}

console.log(Reflect.ownKeys(o));
// ["baz", "qux", Symbol(foo), Symbol(bar)]
```

因为符号属性是对内存中符号的一个引用，所以直接创建并用作属性的符号不会丢失。但是，如果没有显式地保存对这些属性的引用，那么必须遍历对象的所有符号属性才能找到相应的属性键：

```
let o = {
  [Symbol('foo')]: 'foo val',
  [Symbol('bar')]: 'bar val'
};

console.log(o);
// {Symbol(foo): "foo val", Symbol(bar): "bar val"}

let barSymbol = Object.getOwnPropertySymbols(o)
              .find((symbol) => symbol.toString().match(/bar/));
```

```
console.log(barSymbol);
// Symbol(bar)
```

4. **常用内置符号**

ECMAScript 6也引入了一批**常用内置符号**（well-known symbol），用于暴露语言内部行为，开发者可以直接访问、重写或模拟这些行为。这些内置符号都以`Symbol`工厂函数字符串属性的形式存在。

这些内置符号最重要的用途之一是重新定义它们，从而改变原生结构的行为。比如，我们知道`for-of`循环会在相关对象上使用`Symbol.iterator`属性，那么就可以通过在自定义对象上重新定义`Symbol.iterator`的值，来改变`for-of`在迭代该对象时的行为。

这些内置符号也没有什么特别之处，它们就是全局函数`Symbol`的普通字符串属性，指向一个符号的实例。所有内置符号属性都是不可写、不可枚举、不可配置的。

> **注意** 在提到ECMAScript规范时，经常会引用符号在规范中的名称，前缀为`@@`。比如，`@@iterator`指的就是`Symbol.iterator`。

5. **`Symbol.asyncIterator`**

根据ECMAScript规范，这个符号作为一个属性表示“一个方法，该方法返回对象默认的`AsyncIterator`。由`for-await-of`语句使用”。换句话说，这个符号表示实现异步迭代器API的函数。

`for-await-of`循环会利用这个函数执行异步迭代操作。循环时，它们会调用以`Symbol.asyncIterator`为键的函数，并期望这个函数会返回一个实现迭代器API的对象。很多时候，返回的对象是实现该API的`AsyncGenerator`：

```
class Foo {
  async *[Symbol.asyncIterator]() {}
}

let f = new Foo();

console.log(f[Symbol.asyncIterator]());
// AsyncGenerator {<suspended>}
```

技术上，这个由`Symbol.asyncIterator`函数生成的对象应该通过其`next()`方法陆续返回`Promise`实例。可以通过显式地调用`next()`方法返回，也可以隐式地通过异步生成器函数返回：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => resolve(this.asyncIdx++));
    }
```

```
  }
}

async function asyncCount() {
  let emitter = new Emitter(5);

  for await(const x of emitter) {
    console.log(x);
  }
}

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

> **注意** Symbol.asyncIterator是ES2018规范定义的，因此只有版本非常新的浏览器支持它。关于异步迭代和for-await-of循环的细节，参见附录A。

6. Symbol.hasInstance

根据ECMAScript规范，这个符号作为一个属性表示"一个方法，该方法决定一个构造器对象是否认可一个对象是它的实例。由instanceof操作符使用"。instanceof操作符可以用来确定一个对象实例的原型链上是否有原型。instanceof的典型使用场景如下：

```
function Foo() {}
let f = new Foo();
console.log(f instanceof Foo); // true

class Bar {}
let b = new Bar();
console.log(b instanceof Bar); // true
```

在ES6中，instanceof操作符会使用Symbol.hasInstance函数来确定关系。以Symbol.hasInstance为键的函数会执行同样的操作，只是操作数对调了一下：

```
function Foo() {}
let f = new Foo();
console.log(Foo[Symbol.hasInstance](f)); // true

class Bar {}
let b = new Bar();
console.log(Bar[Symbol.hasInstance](b)); // true
```

这个属性定义在Function的原型上，因此默认在所有函数和类上都可以调用。由于instanceof操作符会在原型链上寻找这个属性定义，就跟在原型链上寻找其他属性一样，因此可以在继承的类上通过静态

方法重新定义这个函数：

```
class Bar {}
class Baz extends Bar {
  static [Symbol.hasInstance]() {
    return false;
  }
}

let b = new Baz();
console.log(Bar[Symbol.hasInstance](b)); // true
console.log(b instanceof Bar);           // true
console.log(Baz[Symbol.hasInstance](b)); // false
console.log(b instanceof Baz);           // false
```

7. `Symbol.isConcatSpreadable`

根据ECMAScript规范，这个符号作为一个属性表示"一个布尔值，如果是`true`，则意味着对象应该用`Array.prototype.concat()`打平其数组元素"。ES6中的`Array.prototype.concat()`方法会根据接收到的对象类型选择如何将一个类数组对象拼接成数组实例。覆盖`Symbol.isConcatSpreadable`的值可以修改这个行为。

数组对象默认情况下会被打平到已有的数组，`false`或假值会导致整个对象被追加到数组末尾。类数组对象默认情况下会被追加到数组末尾，`true`或真值会导致这个类数组对象被打平到数组实例。其他不是类数组对象的对象在`Symbol.isConcatSpreadable`被设置为`true`的情况下将被忽略。

```
let initial = ['foo'];

let array = ['bar'];
console.log(array[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(array));             // ['foo', 'bar']
array[Symbol.isConcatSpreadable] = false;
console.log(initial.concat(array));             // ['foo', Array(1)]

let arrayLikeObject = { length: 1, 0: 'baz' };
console.log(arrayLikeObject[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(arrayLikeObject));             // ['foo', {...}]
arrayLikeObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(arrayLikeObject));             // ['foo', 'baz']

let otherObject = new Set().add('qux');
console.log(otherObject[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(otherObject));             // ['foo', Set(1)]
otherObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(otherObject));             // ['foo']
```

8. `Symbol.iterator`

根据ECMAScript规范，这个符号作为一个属性表示“一个方法，该方法返回对象默认的迭代器。由for-of语句使用”。换句话说，这个符号表示实现迭代器API的函数。

for-of循环这样的语言结构会利用这个函数执行迭代操作。循环时，它们会调用以Symbol.iterator为键的函数，并默认这个函数会返回一个实现迭代器API的对象。很多时候，返回的对象是实现该API的Generator：

```
class Foo {
  *[Symbol.iterator]() {}
}

let f = new Foo();

console.log(f[Symbol.iterator]());
// Generator {<suspended>}
```

技术上，这个由Symbol.iterator函数生成的对象应该通过其next()方法陆续返回值。可以通过显式地调用next()方法返回，也可以隐式地通过生成器函数返回：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.idx = 0;
  }

  *[Symbol.iterator]() {
    while(this.idx < this.max) {
      yield this.idx++;
    }
  }
}

function count() {
  let emitter = new Emitter(5);

  for (const x of emitter) {
    console.log(x);
  }
}

count();
// 0
// 1
// 2
// 3
// 4
```

> **注意** 迭代器的相关内容将在第7章详细介绍。

9. `Symbol.match`

根据ECMAScript规范，这个符号作为一个属性表示“一个正则表达式方法，该方法用正则表达式去匹配字符串。由`String.prototype.match()`方法使用”。`String.prototype.match()`方法会使用以`Symbol.match`为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个`String`方法的有效参数：

```
console.log(RegExp.prototype[Symbol.match]);
// f [Symbol.match]() { [native code] }

console.log('foobar'.match(/bar/));
// ["bar", index: 3, input: "foobar", groups: undefined]
```

给这个方法传入非正则表达式值会导致该值被转换为`RegExp`对象。如果想改变这种行为，让方法直接使用参数，则可以重新定义`Symbol.match`函数以取代默认对正则表达式求值的行为，从而让`match()`方法使用非正则表达式实例。`Symbol.match`函数接收一个参数，就是调用`match()`方法的字符串实例。返回的值没有限制：

```
class FooMatcher {
  static [Symbol.match](target) {
    return target.includes('foo');
  }
}

console.log('foobar'.match(FooMatcher)); // true
console.log('barbaz'.match(FooMatcher)); // false

class StringMatcher {
  constructor(str) {
    this.str = str;
  }

  [Symbol.match](target) {
    return target.includes(this.str);
  }
}

console.log('foobar'.match(new StringMatcher('foo'))); // true
console.log('barbaz'.match(new StringMatcher('qux'))); // false
```

10. `Symbol.replace`

根据ECMAScript规范，这个符号作为一个属性表示“一个正则表达式方法，该方法替换一个字符串中匹配的子串。由`String.prototype.replace()`方法使用”。`String.prototype.replace()`方法会使用以`Symbol.replace`为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个`String`方法的有效参数：

```
console.log(RegExp.prototype[Symbol.replace]);
// f [Symbol.replace]() { [native code] }

console.log('foobarbaz'.replace(/bar/, 'qux'));
// 'fooquxbaz'
```

给这个方法传入非正则表达式值会导致该值被转换为RegExp对象。如果想改变这种行为，让方法直接使用参数，可以重新定义Symbol.replace函数以取代默认对正则表达式求值的行为，从而让replace()方法使用非正则表达式实例。Symbol.replace函数接收两个参数，即调用replace()方法的字符串实例和替换字符串。返回的值没有限制：

```
class FooReplacer {
  static [Symbol.replace](target, replacement) {
    return target.split('foo').join(replacement);
  }
}

console.log('barfoobaz'.replace(FooReplacer, 'qux'));
// "barquxbaz"

class StringReplacer {
  constructor(str) {
    this.str = str;
  }

  [Symbol.replace](target, replacement) {
    return target.split(this.str).join(replacement);
  }
}

console.log('barfoobaz'.replace(new StringReplacer('foo'), 'qux'));
// "barquxbaz"
```

11. **Symbol.search**

根据ECMAScript规范，这个符号作为一个属性表示“一个正则表达式方法，该方法返回字符串中匹配正则表达式的索引。由String.prototype.search()方法使用”。String.prototype.search()方法会使用以Symbol.search为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个String方法的有效参数：

```
console.log(RegExp.prototype[Symbol.search]);
// f [Symbol.search]() { [native code] }

console.log('foobar'.search(/bar/));
// 3
```

给这个方法传入非正则表达式值会导致该值被转换为RegExp对象。如果想改变这种行为，让方法直接使用参数，可以重新定义Symbol.search函数以取代默认对正则表达式求值的行为，从而让search()方法使用非正则表达式实例。Symbol.search函数接收一个参数，就是调用match()方法的字符串实例。返回的值没有限制：

```
class FooSearcher {
  static [Symbol.search](target) {
    return target.indexOf('foo');
  }
}

console.log('foobar'.search(FooSearcher)); // 0
console.log('barfoo'.search(FooSearcher)); // 3
console.log('barbaz'.search(FooSearcher)); // -1

class StringSearcher {
  constructor(str) {
    this.str = str;
  }

  [Symbol.search](target) {
    return target.indexOf(this.str);
  }
}

console.log('foobar'.search(new StringSearcher('foo'))); // 0
console.log('barfoo'.search(new StringSearcher('foo'))); // 3
console.log('barbaz'.search(new StringSearcher('qux'))); // -1
```

12. **Symbol.species**

根据ECMAScript规范，这个符号作为一个属性表示“一个函数值，该函数作为创建派生对象的构造函数”。这个属性在内置类型中最常用，用于对内置类型实例方法的返回值暴露实例化派生对象的方法。用Symbol.species定义静态的获取器（getter）方法，可以覆盖新创建实例的原型定义：

```
class Bar extends Array {}
class Baz extends Array {
  static get [Symbol.species]() {
    return Array;
  }
}

let bar = new Bar();
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true
bar = bar.concat('bar');
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true

let baz = new Baz();
```

```
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // true
baz = baz.concat('baz');
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // false
```

13. **Symbol.split**

根据ECMAScript规范，这个符号作为一个属性表示“一个正则表达式方法，该方法在匹配正则表达式的索引位置拆分字符串。由String.prototype.split()方法使用”。String.prototype.split()方法会使用以Symbol.split为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个String方法的有效参数：

```
console.log(RegExp.prototype[Symbol.split]);
// f [Symbol.split]() { [native code] }

console.log('foobarbaz'.split(/bar/));
// ['foo', 'baz']
```

给这个方法传入非正则表达式值会导致该值被转换为RegExp对象。如果想改变这种行为，让方法直接使用参数，可以重新定义Symbol.split函数以取代默认对正则表达式求值的行为，从而让split()方法使用非正则表达式实例。Symbol.split函数接收一个参数，就是调用match()方法的字符串实例。返回的值没有限制：

```
class FooSplitter {
  static [Symbol.split](target) {
    return target.split('foo');
  }
}

console.log('barfoobaz'.split(FooSplitter));
// ["bar", "baz"]

class StringSplitter {
  constructor(str) {
    this.str = str;
  }

  [Symbol.split](target) {
    return target.split(this.str);
  }
}

console.log('barfoobaz'.split(new StringSplitter('foo')));
// ["bar", "baz"]
```

14. **Symbol.toPrimitive**

根据ECMAScript规范，这个符号作为一个属性表示“一个方法，该方法将对象转换为相应的原始值。由ToPrimitive抽象操作使用”。很多内置操作都会尝试强制将对象转换为原始值，包括字符串、数值和未指定的原始类型。对于一个自定义对象实例，通过在这个实例的Symbol.toPrimitive属性上定义一个函数可以改变默认行为。

根据提供给这个函数的参数（string、number或default），可以控制返回的原始值：

```
class Foo {}
let foo = new Foo();

console.log(3 + foo);       // "3[object Object]"
console.log(3 - foo);       // NaN
console.log(String(foo));   // "[object Object]"

class Bar {
  constructor() {
    this[Symbol.toPrimitive] = function(hint) {
      switch (hint) {
        case 'number':
          return 3;
        case 'string':
          return 'string bar';
        case 'default':
        default:
          return 'default bar';
      }
    }
  }
}
let bar = new Bar();

console.log(3 + bar);     // "3default bar"
console.log(3 - bar);     // 0
console.log(String(bar)); // "string bar"
```

15. **Symbol.toStringTag**

根据ECMAScript规范，这个符号作为一个属性表示“一个字符串，该字符串用于创建对象的默认字符串描述。由内置方法Object.prototype.toString()使用”。

通过toString()方法获取对象标识时，会检索由Symbol.toStringTag指定的实例标识符，默认为"Object"。内置类型已经指定了这个值，但自定义类实例还需要明确定义：

```
let s = new Set();

console.log(s);                      // Set(0) {}
console.log(s.toString());           // [object Set]
console.log(s[Symbol.toStringTag]);  // Set

class Foo {}
```

```
let foo = new Foo();

console.log(foo);                       // Foo {}
console.log(foo.toString());            // [object Object]
console.log(foo[Symbol.toStringTag]);   // undefined

class Bar {
  constructor() {
    this[Symbol.toStringTag] = 'Bar';
  }
}
let bar = new Bar();

console.log(bar);                       // Bar {}
console.log(bar.toString());            // [object Bar]
console.log(bar[Symbol.toStringTag]);   // Bar
```

16. `Symbol.unscopables`

根据ECMAScript规范，这个符号作为一个属性表示“一个对象，该对象所有的以及继承的属性，都会从关联对象的`with`环境绑定中排除”。设置这个符号并让其映射对应属性的键值为`true`，就可以阻止该属性出现在`with`环境绑定中，如下例所示：

```
let o = { foo: 'bar' };

with (o) {
  console.log(foo); // bar
}

o[Symbol.unscopables] = {
  foo: true
};

with (o) {
  console.log(foo); // ReferenceError
}
```

> **注意** 不推荐使用`with`，因此也不推荐使用`Symbol.unscopables`。

3.4.8 `Object`类型

ECMAScript中的对象其实就是一组数据和功能的集合。对象通过`new`操作符后跟对象类型的名称来创建。开发者可以通过创建`Object`类型的实例来创建自己的对象，然后再给对象添加属性和方法：

```
let o = new Object();
```

这个语法类似Java，但ECMAScript只要求在给构造函数提供参数时使用括号。如果没有参数，如上面的例子所示，那么完全可以省略括号（不推荐）：

```
let o = new Object;  // 合法，但不推荐
```

Object的实例本身并不是很有用，但理解与它相关的概念非常重要。类似Java中的java.lang.Object，ECMAScript中的Object也是派生其他对象的基类。Object类型的所有属性和方法在派生的对象上同样存在。

每个Object实例都有如下属性和方法。

- constructor：用于创建当前对象的函数。在前面的例子中，这个属性的值就是Object()函数。
- hasOwnProperty(*propertyName*)：用于判断当前对象实例（不是原型）上是否存在给定的属性。要检查的属性名必须是字符串（如o.hasOwnProperty("name")）。
- isPrototypeof(*object*)：用于判断当前对象是否为另一个对象的原型。（第5章将详细介绍原型。）
- propertyIsEnumerable(*propertyName*)：用于判断给定的属性是否可以使用（本章稍后讨论的）for-in语句枚举。与hasOwnProperty()一样，属性名必须是字符串。
- toLocaleString()：返回对象的字符串表示，该字符串反映对象所在的本地化执行环境。
- toString()：返回对象的字符串表示。
- valueOf()：返回对象对应的字符串、数值或布尔值表示。通常与toString()的返回值相同。

因为在ECMAScript中Object是所有对象的基类，所以任何对象都有这些属性和方法。第5章和第6章将介绍对象间的继承机制。

> **注意** 严格来讲，ECMA-262中对象的行为不一定适合JavaScript中的其他对象。比如浏览器环境中的BOM和DOM对象，都是由宿主环境定义和提供的宿主对象。而宿主对象不受ECMA-262约束，所以它们可能会也可能不会继承Object。

3.5 操作符

ECMA-262描述了一组可用于操作数据值的**操作符**，包括数学操作符（如加、减）、位操作符、关系操作符和相等操作符等。ECMAScript中的操作符是独特的，因为它们可用于各种值，包括字符串、数值、布尔值，甚至还有对象。在应用给对象时，操作符通常会调用valueOf()和/或toString()方法来取得可以计算的值。

3.5.1 一元操作符

只操作一个值的操作符叫**一元操作符**（unary operator）。一元操作符是ECMAScript中最简单的操作符。

1. **递增/递减操作符**

   递增和递减操作符直接照搬自C语言，但有两个版本：前缀版和后缀版。顾名思义，前缀版就是位于要操作的变量前头，后缀版就是位于要操作的变量后头。前缀递增操作符会给数值加1，把两个加号（++）放到变量前头即可：

   ```
   let age = 29;
   ++age;
   ```

   在这个例子中，前缀递增操作符把age的值变成了30（给之前的值29加1）。因此，它实际上等于如下表达式：

```
let age = 29;
age = age + 1;
```

前缀递减操作符也类似，只不过是从一个数值减1。使用前缀递减操作符，只要把两个减号（--）放到变量前头即可：

```
let age = 29;
--age;
```

执行操作后，变量age的值变成了28（从29减1）。

无论使用前缀递增还是前缀递减操作符，变量的值都会在语句被求值之前改变。（在计算机科学中，这通常被称为具有**副作用**。）请看下面的例子：

```
let age = 29;
let anotherAge = --age + 2;

console.log(age);         // 28
console.log(anotherAge);  // 30
```

在这个例子中，变量anotherAge以age减1后的值再加2进行初始化。因为递减操作先发生，所以age的值先变成28，然后再加2，结果是30。

前缀递增和递减在语句中的优先级是相等的，因此会从左到右依次求值。比如：

```
let num1 = 2;
let num2 = 20;
let num3 = --num1 + num2;
let num4 = num1 + num2;
console.log(num3);  // 21
console.log(num4);  // 21
```

这里，num3等于21是因为num1先减1之后才加num2。变量num4也是21，那是因为加法使用的也是递减后的值。

递增和递减的后缀版语法一样（分别是++和--），只不过要放在变量后面。后缀版与前缀版的主要区别在于，后缀版递增和递减在语句被求值后才发生。在某些情况下，这种差异没什么影响，比如：

```
let age = 29;
age++;
```

把递增操作符放到变量后面不会改变语句执行的结果，因为递增是唯一的操作。可是，在跟其他操作混合时，差异就会变明显，比如：

```
let num1 = 2;
let num2 = 20;
let num3 = num1-- + num2;
let num4 = num1 + num2;
console.log(num3);  // 22
console.log(num4);  // 21
```

这个例子跟前面的那个一样，只是把前缀递减改成了后缀递减，区别很明显。在使用前缀版的例子中，num3和num4的值都是21。而在这个例子中，num3的值是22，num4的值是21。这里的不同之处在于，计算num3时使用的是num1的原始值（2），而计算num4时使用的是num1递减后的值（1）。

这4个操作符可以作用于任何值，意思是不限于整数——字符串、布尔值、浮点值，甚至对象都可以。递增和递减操作符遵循如下规则。

- 对于字符串，如果是有效的数值形式，则转换为数值再应用改变。变量类型从字符串变成数值。
- 对于字符串，如果不是有效的数值形式，则将变量的值设置为NaN。变量类型从字符串变成数值。
- 对于布尔值，如果是false，则转换为0再应用改变。变量类型从布尔值变成数值。
- 对于布尔值，如果是true，则转换为1再应用改变。变量类型从布尔值变成数值。
- 对于浮点值，加1或减1。
- 如果是对象，则调用其（第5章会详细介绍的）valueOf()方法取得可以操作的值。对得到的值应用上述规则。如果是NaN，则调用toString()并再次应用其他规则。变量类型从对象变成数值。

下面的例子演示了这些规则：

```
let s1 = "2";
let s2 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
};

s1++;  // 值变成数值3
s2++;  // 值变成NaN
b++;   // 值变成数值1
f--;   // 值变成0.10000000000000009（因为浮点数不精确）
o--;   // 值变成-2
```

2. **一元加和减**

一元加和减操作符对大多数开发者来说并不陌生，它们在ECMAScript中跟在高中数学中的用途一样。一元加由一个加号（+）表示，放在变量前头，对数值没有任何影响：

```
let num = 25;
num = +num;
console.log(num); // 25
```

如果将一元减应用到非数值，则会执行与使用Number()转型函数一样的类型转换：布尔值false和true转换为0和1，字符串根据特殊规则进行解析，对象会调用它们的valueOf()和/或toString()方法以得到可以转换的值。

下面的例子演示了一元加在应用到不同数据类型时的行为：

```
let s1 = "01";
let s2 = "1.1";
let s3 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
};

s1 = +s1;  // 值变成数值1
s2 = +s2;  // 值变成数值1.1
s3 = +s3;  // 值变成NaN
b = +b;    // 值变成数值0
f = +f;    // 不变，还是1.1
o = +o;    // 值变成数值-1
```

一元减由一个减号（-）表示，放在变量前头，主要用于把数值变成负值，如把1转换为-1。示例如下：

```
let num = 25;
num = -num;
console.log(num);  // -25
```

对数值使用一元减会将其变成相应的负值（如上面的例子所示）。在应用到非数值时，一元减会遵循与一元加同样的规则，先对它们进行转换，然后再取负值：

```
let s1 = "01";
let s2 = "1.1";
let s3 = "z";
let b = false;
let f = 1.1;
let o = {
  valueOf() {
    return -1;
  }
```

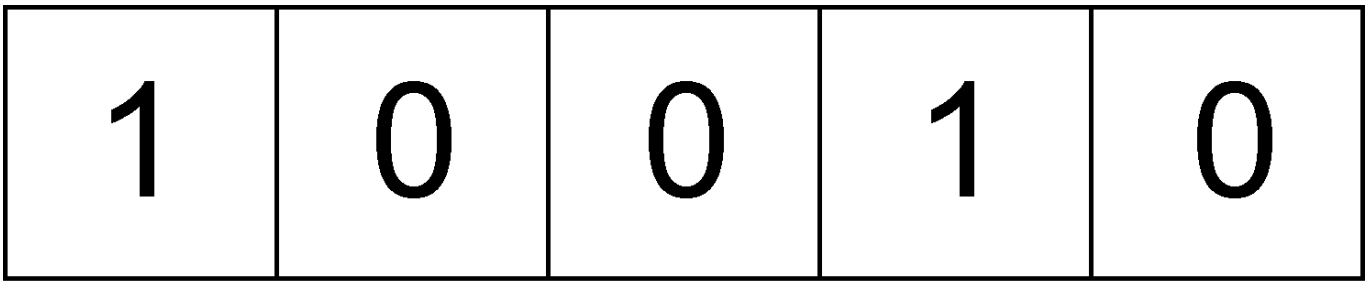
```
};

s1 = -s1;  // 值变成数值-1
s2 = -s2;  // 值变成数值-1.1
s3 = -s3;  // 值变成NaN
b = -b;    // 值变成数值0
f = -f;    // 变成-1.1
o = -o;    // 值变成数值1
```

一元加和减操作符主要用于基本的算术，但也可以像上面的例子那样，用于数据类型转换。

3.5.2 位操作符

接下来要介绍的操作符用于数值的底层操作，也就是操作内存中表示数据的比特（位）。ECMAScript中的所有数值都以IEEE 754 64位格式存储，但位操作并不直接应用到64位表示，而是先把值转换为32位整数，再进行位操作，之后再把结果转换为64位。对开发者而言，就好像只有32位整数一样，因为64位整数存储格式是不可见的。既然知道了这些，就只需要考虑32位整数即可。

有符号整数使用32位的前31位表示整数值。第32位表示数值的符号，如0表示正，1表示负。这一位称为**符号位**（sign bit），它的值决定了数值其余部分的格式。正值以真正的二进制格式存储，即31位中的每一位都代表2的幂。第一位（称为第0位）表示20，第二位表示21，依此类推。如果一个位是空的，则以0填充，相当于忽略不计。比如，数值18的二进制格式为00000000000000000000000000010010，或更精简的10010。后者是用到的5个有效位，决定了实际的值（如图3-1所示）。



$$(2^4 \times 1) + (2^3 \times 0) + (2^2 \times 0) + (2^1 \times 1) + (2^0 \times 0)$$

$$16 + 0 + 0 + 2 + 0$$

$$18$$

图 3-1

负值以一种称为**二补数**的二进制编码存储。一个数值的二补数通过如下3个步骤计算得到：

(1) 确定绝对值的二进制表示（如，对于-18，先确定18的二进制表示）；

(2) 找到数值的一补数，换句话说，就是每个0都变成1，每个1都变成0；

(3) 给结果加1。

基于上述步骤确定-18的二进制表示，首先从18的二进制表示开始：

```
0000  0000  0000  0000  0000  0000  0001  0010
```

然后，计算一补数，即反转每一位的二进制值：

```
1111  1111  1111  1111  1111  1111  1110  1101
```

最后，给一补数加1：

```
1111  1111  1111  1111  1111  1111  1110  1101
                      1
-----------------------------------------------
1111  1111  1111  1111  1111  1111  1110  1110
```

那么，-18的二进制表示就是11111111111111111111111111101110。要注意的是，在处理有符号整数时，我们无法访问第31位。

ECMAScript会帮我们记录这些信息。在把负值输出为一个二进制字符串时，我们会得到一个前面加了减号的绝对值，如下所示：

```
let num = -18;
console.log(num.toString(2)); // "-10010"
```

在将-18转换为二进制字符串时，结果得到-10010。转换过程会求得二补数，然后再以更符合逻辑的形式表示出来。

> **注意** 默认情况下，ECMAScript中的所有整数都表示为有符号数。不过，确实存在无符号整数。对无符号整数来说，第32位不表示符号，因为只有正值。无符号整数比有符号整数的范围更大，因为符号位被用来表示数值了。

在对ECMAScript中的数值应用位操作符时，后台会发生转换：64位数值会转换为32位数值，然后执行位操作，最后再把结果从32位转换为64位存储起来。整个过程就像处理32位数值一样，这让二进制操作变得与其他语言中类似。但这个转换也导致了一个奇特的副作用，即特殊值`NaN`和`Infinity`在位操作中都会被当成0处理。

如果将位操作符应用到非数值，那么首先会使用`Number()`函数将该值转换为数值（这个过程是自动的），然后再应用位操作。最终结果是数值。

1. **按位非**

   按位非操作符用波浪符（~）表示，它的作用是返回数值的一补数。按位非是ECMAScript中为数不多的几个二进制数学操作符之一。看下面的例子：

```
let num1 = 25;      // 二进制00000000000000000000000000011001
let num2 = ~num1;   // 二进制11111111111111111111111111100110
console.log(num2);  // -26
```

这里，按位非操作符作用到了数值25，得到的结果是-26。由此可以看出，按位非的最终效果是对数值取反并减1，就像执行如下操作的结果一样：

```
let num1 = 25;
let num2 = -num1 - 1;
console.log(num2);   // "-26"
```

实际上，尽管两者返回的结果一样，但位操作的速度快得多。这是因为位操作是在数值的底层表示上完成的。

2. **按位与**

按位与操作符用和号（&）表示，有两个操作数。本质上，按位与就是将两个数的每一个位对齐，然后基于真值表中的规则，对每一位执行相应的与操作。

第一个数值的位	第二个数值的位	结果
1	1	1
1	0	0
0	1	0
0	0	0

按位与操作在两个位都是1时返回1，在任何一位是0时返回0。

下面看一个例子，我们对数值25和3求与操作，如下所示：

```
let result = 25 & 3;
console.log(result); // 1
```

25和3的按位与操作的结果是1。为什么呢？看下面的二进制计算过程：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
-----------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

如上所示，25和3的二进制表示中，只有第0位上的两个数都是1。于是结果数值的所有其他位都会以0填充，因此结果就是1。

3. **按位或**

按位或操作符用管道符（|）表示，同样有两个操作数。按位或遵循如下真值表：

第一个数值的位	第二个数值的位	结果
1	1	1
1	0	1
0	1	1
0	0	0

按位或操作在至少一位是1时返回1，两位都是0时返回0。

仍然用按位与的示例，如果对25和3执行按位或，代码如下所示：

```
let result = 25 | 3;
console.log(result); // 27
```

可见25和3的按位或操作的结果是27：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
-------------------------------------------
 OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

在参与计算的两个数中，有4位都是1，因此它们直接对应到结果上。二进制码11011等于27。

4. **按位异或**

按位异或用脱字符（^）表示，同样有两个操作数。下面是按位异或的真值表：

第一个数的位	第二个数的位	结果
1	1	0
1	0	1
0	1	1
0	0	0

按位异或与按位或的区别是，它只在一位上是1的时候返回1（两位都是1或0，则返回0）。

对数值25和3执行按位异或操作：

```
let result = 25 ^ 3;
console.log(result); // 26
```

可见，25和3的按位异或操作结果为26，如下所示：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
-----------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

两个数在4位上都是1，但两个数的第0位都是1，因此那一位在结果中就变成了0。其余位上的1在另一个数上没有对应的1，因此会直接传递到结果中。二进制码11010等于26。（注意，这比对同样两个值执行按位或操作得到的结果小1。）

5. **左移**

左移操作符用两个小于号（<<）表示，会按照指定的位数将数值的所有位向左移动。比如，如果数值2（二进制10）向左移5位，就会得到64（二进制1000000），如下所示：

```
let oldValue = 2;               // 等于二进制10
let newValue = oldValue << 5;  // 等于二进制1000000，即十进制64
```

注意在移位后，数值右端会空出5位。左移会以0填充这些空位，让结果是完整的32位数值（见图3-2）。

图 3-2

注意，左移会保留它所操作数值的符号。比如，如果-2左移5位，将得到-64，而不是正64。

6. **有符号右移**

有符号右移由两个大于号（>>）表示，会将数值的所有32位都向右移，同时保留符号（正或负）。有符号右移实际上是左移的逆运算。比如，如果将64右移5位，那就是2：

```
let oldValue = 64;              // 等于二进制1000000
let newValue = oldValue >> 5;  // 等于二进制10，即十进制2
```

同样，移位后就会出现空位。不过，右移后空位会出现在左侧，且在符号位之后（见图3-3）。ECMAScript会用符号位的值来填充这些空位，以得到完整的数值。

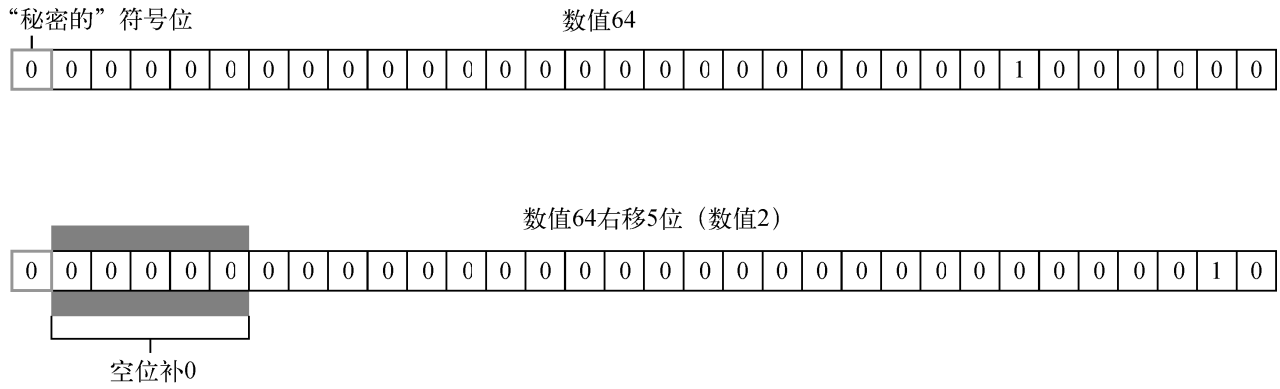


图 3-3

7. **无符号右移**

无符号右移用3个大于号表示（>>>），会将数值的所有32位都向右移。对于正数，无符号右移与有符号右移结果相同。仍然以前面有符号右移的例子为例，64向右移动5位，会变成2：

```
let oldValue = 64;              // 等于二进制1000000
let newValue = oldValue >>> 5;  // 等于二进制10，即十进制2
```

对于负数，有时候差异会非常大。与有符号右移不同，无符号右移会给空位补0，而不管符号位是什么。对正数来说，这跟有符号右移效果相同。但对负数来说，结果就差太多了。无符号右移操作符将负数的二进制表示当成正数的二进制表示来处理。因为负数是其绝对值的二补数，所以右移之后结果变得非常之大，如下面的例子所示：

```
let oldValue = -64;               // 等于二进制
11111111111111111111111111000000
let newValue = oldValue >>> 5;    // 等于十进制134217726
```

在对-64无符号右移5位后，结果是134 217 726。这是因为-64的二进制表示是11111111111111111111111111000000，无符号右移却将它当成正值，也就是4 294 967 232。把这个值右移5位后，结果是00000111111111111111111111111110，即134 217 726。

3.5.3 布尔操作符

对于编程语言来说，布尔操作符跟相等操作符几乎同样重要。如果没有能力测试两个值的关系，那么像if-else和循环这样的语句也没什么用了。布尔操作符一共有3个：逻辑非、逻辑与和逻辑或。

1. **逻辑非**

逻辑非操作符由一个叹号（!）表示，可应用给ECMAScript中的任何值。这个操作符始终返回布尔值，无论应用到的是什么数据类型。逻辑非操作符首先将操作数转换为布尔值，然后再对其取反。换句话说，逻辑非操作符会遵循如下规则。

- 如果操作数是对象，则返回false。
- 如果操作数是空字符串，则返回true。
- 如果操作数是非空字符串，则返回false。

- 如果操作数是数值0，则返回true。
- 如果操作数是非0数值（包括Infinity），则返回false。
- 如果操作数是null，则返回true。
- 如果操作数是NaN，则返回true。
- 如果操作数是undefined，则返回true。

以下示例验证了上述行为：

```
console.log(!false);   // true
console.log(!"blue");  // false
console.log(!0);       // true
console.log(!NaN);     // true
console.log(!"");      // true
console.log(!12345);   // false
```

逻辑非操作符也可以用于把任意值转换为布尔值。同时使用两个叹号（!!），相当于调用了转型函数Boolean()。无论操作数是什么类型，第一个叹号总会返回布尔值。第二个叹号对该布尔值取反，从而给出变量真正对应的布尔值。结果与对同一个值使用Boolean()函数是一样的：

```
console.log(!!"blue"); // true
console.log(!!0);      // false
console.log(!!NaN);    // false
console.log(!!"");     // false
console.log(!!12345);  // true
```

2. **逻辑与**

逻辑与操作符由两个和号（&&）表示，应用到两个值，如下所示：

```
let result = true && false;
```

逻辑与操作符遵循如下真值表：

第一个操作数	第二个操作数	结果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑与操作符可用于任何类型的操作数，不限于布尔值。如果有操作数不是布尔值，则逻辑与并不一定会返回布尔值，而是遵循如下规则。

- 如果第一个操作数是对象，则返回第二个操作数。

- 如果第二个操作数是对象，则只有第一个操作数求值为true才会返回该对象。
- 如果两个操作数都是对象，则返回第二个操作数。
- 如果有一个操作数是null，则返回null。
- 如果有一个操作数是NaN，则返回NaN。
- 如果有一个操作数是undefined，则返回undefined。

逻辑与操作符是一种短路操作符，意思就是如果第一个操作数决定了结果，那么永远不会对第二个操作数求值。对逻辑与操作符来说，如果第一个操作数是false，那么无论第二个操作数是什么值，结果也不可能等于true。看下面的例子：

```
let found = true;
let result = (found && someUndeclaredVariable); // 这里会出错
console.log(result); // 不会执行这一行
```

上面的代码之所以会出错，是因为someUndeclaredVariable没有事先声明，所以当逻辑与操作符对它求值时就会报错。变量found的值是true，逻辑与操作符会继续求值变量someUndeclaredVariable。但是由于someUndeclaredVariable没有定义，不能对它应用逻辑与操作符，因此就报错了。假如变量found的值是false，那么就不会报错了：

```
let found = false;
let result = (found && someUndeclaredVariable);  // 不会出错
console.log(result);  // 会执行
```

这里，console.log会成功执行。即使变量someUndeclaredVariable没有定义，由于第一个操作数是false，逻辑与操作符也不会对它求值，因为此时对&&右边的操作数求值是没有意义的。在使用逻辑与操作符时，一定别忘了它的这个短路的特性。

3. **逻辑或**

逻辑或操作符由两个管道符（||）表示，比如：

```
let result = true || false;
```

逻辑或操作符遵循如下真值表：

第一个操作数	第二个操作数	结果
true	true	true
true	false	true
false	true	true
false	false	false

与逻辑与类似，如果有一个操作数不是布尔值，那么逻辑或操作符也不一定返回布尔值。它遵循如下规则。

- 如果第一个操作数是对象，则返回第一个操作数。
- 如果第一个操作数求值为`false`，则返回第二个操作数。
- 如果两个操作数都是对象，则返回第一个操作数。
- 如果两个操作数都是`null`，则返回`null`。
- 如果两个操作数都是`NaN`，则返回`NaN`。
- 如果两个操作数都是`undefined`，则返回`undefined`。

同样与逻辑与类似，逻辑或操作符也具有短路的特性。只不过对逻辑或而言，第一个操作数求值为`true`，第二个操作数就不会再被求值了。看下面的例子：

```
let found = true;
let result = (found || someUndeclaredVariable); // 不会出错
console.log(result); // 会执行
```

跟前面的例子一样，变量`someUndeclaredVariable`也没有定义。但是，因为变量`found`的值为`true`，所以逻辑或操作符不会对变量`someUndeclaredVariable`求值，而直接返回`true`。假如把`found`的值改为`false`，那就会报错了：

```
let found = false;
let result = (found || someUndeclaredVariable); // 这里会出错
console.log(result); // 不会执行这一行
```

利用这个行为，可以避免给变量赋值`null`或`undefined`。比如：

```
let myObject = preferredObject || backupObject;
```

在这个例子中，变量`myObject`会被赋予两个值中的一个。其中，`preferredObject`变量包含首选的值，`backupObject`变量包含备用的值。如果`preferredObject`不是`null`，则它的值就会赋给`myObject`；如果`preferredObject`是`null`，则`backupObject`的值就会赋给`myObject`。这种模式在ECMAScript代码中经常用于变量赋值，本书后面的代码示例中也会经常用到。

3.5.4 乘性操作符

ECMAScript定义了3个乘性操作符：乘法、除法和取模。这些操作符跟它们在Java、C语言及Perl中对应的操作符作用一样，但在处理非数值时，它们也会包含一些自动的类型转换。如果乘性操作符有不是数值的操作数，则该操作数会在后台被使用`Number()`转型函数转换为数值。这意味着空字符串会被当成0，而布尔值`true`会被当成1。

1. **乘法操作符**

乘法操作符由一个星号（`*`）表示，可以用于计算两个数值的乘积。其语法类似于C语言，比如：

```
let result = 34 * 56;
```

不过，乘法操作符在处理特殊值时也有一些特殊的行为。

- 如果操作数都是数值，则执行常规的乘法运算，即两个正值相乘是正值，两个负值相乘也是正值，正负符号不同的值相乘得到负值。如果ECMAScript不能表示乘积，则返回`Infinity`或`-Infinity`。
- 如果有任一操作数是`NaN`，则返回`NaN`。
- 如果是`Infinity`乘以0，则返回`NaN`。
- 如果是`Infinity`乘以非0的有限数值，则根据第二个操作数的符号返回`Infinity`或`-Infinity`。
- 如果是`Infinity`乘以`Infinity`，则返回`Infinity`。
- 如果有不是数值的操作数，则先在后台用`Number()`将其转换为数值，然后再应用上述规则。

2. **除法操作符**

除法操作符由一个斜杠（`/`）表示，用于计算第一个操作数除以第二个操作数的商，比如：

```
let result = 66 / 11;
```

跟乘法操作符一样，除法操作符针对特殊值也有一些特殊的行为。

- 如果操作数都是数值，则执行常规的除法运算，即两个正值相除是正值，两个负值相除也是正值，符号不同的值相除得到负值。如果ECMAScript不能表示商，则返回`Infinity`或`-Infinity`。
- 如果有任一操作数是`NaN`，则返回`NaN`。
- 如果是`Infinity`除以`Infinity`，则返回`NaN`。
- 如果是0除以0，则返回`NaN`。
- 如果是非0的有限值除以0，则根据第一个操作数的符号返回`Infinity`或`-Infinity`。
- 如果是`Infinity`除以任何数值，则根据第二个操作数的符号返回`Infinity`或`-Infinity`。
- 如果有不是数值的操作数，则先在后台用`Number()`函数将其转换为数值，然后再应用上述规则。

3. **取模操作符**

取模（余数）操作符由一个百分比符号（`%`）表示，比如：

```
let result = 26 % 5; // 等于1
```

与其他乘性操作符一样，取模操作符对特殊值也有一些特殊的行为。

- 如果操作数是数值，则执行常规除法运算，返回余数。
- 如果被除数是无限值，除数是有限值，则返回`NaN`。
- 如果被除数是有限值，除数是0，则返回`NaN`。
- 如果是`Infinity`除以`Infinity`，则返回`NaN`。
- 如果被除数是有限值，除数是无限值，则返回被除数。
- 如果被除数是0，除数不是0，则返回0。
- 如果有不是数值的操作数，则先在后台用`Number()`函数将其转换为数值，然后再应用上述规则。

3.5.5 指数操作符

ECMAScript 7新增了指数操作符，`Math.pow()`现在有了自己的操作符`**`，结果是一样的：

```
console.log(Math.pow(3, 2);    // 9
console.log(3 ** 2);           // 9

console.log(Math.pow(16, 0.5); // 4
console.log(16** 0.5);         // 4
```

不仅如此，指数操作符也有自己的指数赋值操作符`**=`，该操作符执行指数运算和结果的赋值操作：

```
let squared = 3;
squared **= 2;
console.log(squared); // 9

let sqrt = 16;
sqrt **= 0.5;
console.log(sqrt); // 4
```

3.5.6 加性操作符

加性操作符，即加法和减法操作符，一般都是编程语言中最简单的操作符。不过，在ECMAScript中，这两个操作符拥有一些特殊的行为。与乘性操作符类似，加性操作符在后台会发生不同数据类型的转换。只不过对这两个操作符来说，转换规则不是那么直观。

1. **加法操作符**

   加法操作符（`+`）用于求两个数的和，比如：

   ```
   let result = 1 + 2;
   ```

   如果两个操作数都是数值，加法操作符执行加法运算并根据如下规则返回结果：

   - 如果有任一操作数是`NaN`，则返回`NaN`；
   - 如果是`Infinity`加`Infinity`，则返回`Infinity`；
   - 如果是`-Infinity`加`-Infinity`，则返回`-Infinity`；
   - 如果是`Infinity`加`-Infinity`，则返回`NaN`；
   - 如果是`+0`加`+0`，则返回`+0`；
   - 如果是`-0`加`+0`，则返回`+0`；
   - 如果是`-0`加`-0`，则返回`-0`。

   不过，如果有一个操作数是字符串，则要应用如下规则：

   - 如果两个操作数都是字符串，则将第二个字符串拼接到第一个字符串后面；
   - 如果只有一个操作数是字符串，则将另一个操作数转换为字符串，再将两个字符串拼接在一起。

如果有任一操作数是对象、数值或布尔值，则调用它们的`toString()`方法以获取字符串，然后再应用前面的关于字符串的规则。对于`undefined`和`null`，则调用`String()`函数，分别获取`"undefined"`和`"null"`。

看下面的例子：

```
let result1 = 5 + 5;        // 两个数值
console.log(result1);       // 10
let result2 = 5 + "5";      // 一个数值和一个字符串
console.log(result2);       // "55"
```

以上代码展示了加法操作符的两种运算模式。正常情况下，`5 + 5`等于10（数值），如前两行代码所示。但是，如果将一个操作数改为字符串，比如`"5"`，则相加的结果就变成了`"55"`（原始字符串值），因为第一个操作数也会被转换为字符串。

ECMAScript中最常犯的一个错误，就是忽略加法操作中涉及的数据类型。比如下面这个例子：

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + num1 + num2;
console.log(message);  // "The sum of 5 and 10 is 510"
```

这里，变量`message`中保存的是一个字符串，是执行两次加法操作之后的结果。有人可能会认为最终得到的字符串是`"The sum of 5 and 10 is 15"`。可是，实际上得到的是`"The sum of 5 and 10 is 510"`。这是因为每次加法运算都是独立完成的。第一次加法的操作数是一个字符串和一个数值（5），结果还是一个字符串。第二次加法仍然是用一个字符串去加一个数值（10），同样也会得到一个字符串。如果想真正执行数学计算，然后把结果追加到字符串末尾，只要使用一对括号即可：

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + (num1 + num2);
console.log(message); // "The sum of 5 and 10 is 15"
```

在此，我们用括号把两个数值变量括了起来，意思是让解释器先执行两个数值的加法，然后再把结果追加给字符串。因此，最终得到的字符串变成了`"The sum of 5 and 10 is 15"`。

2. **减法操作符**

减法操作符（`-`）也是使用很频繁的一种操作符，比如：

```
let result = 2 - 1;
```

与加法操作符一样，减法操作符也有一组规则用于处理ECMAScript中不同类型之间的转换。

- 如果两个操作数都是数值，则执行数学减法运算并返回结果。
- 如果有任一操作数是NaN，则返回NaN。
- 如果是Infinity减Infinity，则返回NaN。
- 如果是-Infinity减-Infinity，则返回NaN。
- 如果是Infinity减-Infinity，则返回Infinity。
- 如果是-Infinity减Infinity，则返回-Infinity。
- 如果是+0减+0，则返回+0。
- 如果是+0减-0，则返回-0。
- 如果是-0减-0，则返回+0。
- 如果有任一操作数是字符串、布尔值、null或undefined，则先在后台使用Number()将其转换为数值，然后再根据前面的规则执行数学运算。如果转换结果是NaN，则减法计算的结果是NaN。
- 如果有任一操作数是对象，则调用其valueOf()方法取得表示它的数值。如果该值是NaN，则减法计算的结果是NaN。如果对象没有valueOf()方法，则调用其toString()方法，然后再将得到的字符串转换为数值。

以下示例演示了上面的规则：

```
let result1 = 5 - true; // true被转换为1，所以结果是4
let result2 = NaN - 1;  // NaN
let result3 = 5 - 3;    // 2
let result4 = 5 - "";   // ""被转换为0，所以结果是5
let result5 = 5 - "2";  // "2"被转换为2，所以结果是3
let result6 = 5 - null; // null被转换为0，所以结果是5
```

3.5.7 关系操作符

关系操作符执行比较两个值的操作，包括小于（<）、大于（>）、小于等于（<=）和大于等于（>=），用法跟数学课上学的一样。这几个操作符都返回布尔值，如下所示：

```
let result1 = 5 > 3; // true
let result2 = 5 < 3; // false
```

与ECMAScript中的其他操作符一样，在将它们应用到不同数据类型时也会发生类型转换和其他行为。

- 如果操作数都是数值，则执行数值比较。
- 如果操作数都是字符串，则逐个比较字符串中对应字符的编码。
- 如果有任一操作数是数值，则将另一个操作数转换为数值，执行数值比较。
- 如果有任一操作数是对象，则调用其valueOf()方法，取得结果后再根据前面的规则执行比较。如果没有valueOf()操作符，则调用toString()方法，取得结果后再根据前面的规则执行比较。
- 如果有任一操作数是布尔值，则将其转换为数值再执行比较。

在使用关系操作符比较两个字符串时，会发生一个有趣的现象。很多人认为小于意味着“字母顺序靠前”，而大于意味着“字母顺序靠后”，实际上不是这么回事。对字符串而言，关系操作符会比较字符串中对应字符的编码，而这些编码是数值。比较完之后，会返回布尔值。问题的关键在于，大写字母的编码都小于小写字母的编码，因此以下这种情况就会发生：

```
let result = "Brick" < "alphabet"; // true
```

在这里，字符串"Brick"被认为小于字符串"alphabet"，因为字母B的编码的66，字母a的编码是97。要得到确实按字母顺序比较的结果，就必须把两者都转换为相同的大小写形式（全大写或全小写），然后再比较：

```
let result = "Brick".toLowerCase() < "alphabet".toLowerCase(); // false
```

将两个操作数都转换为小写，就能保证按照字母表顺序判定"alphabet"在"Brick"前头。

另一个奇怪的现象是在比较两个数值字符串的时候，比如下面这个例子：

```
let result = "23" < "3"; // true
```

这里在比较字符串"23"和"3"时返回true。因为两个操作数都是字符串，所以会逐个比较它们的字符编码（字符"2"的编码是50，而字符"3"的编码是51）。不过，如果有一个操作数是数值，那么比较的结果就对了：

```
let result = "23" < 3; // false
```

因为这次会将字符串"23"转换为数值23，然后再跟3比较，结果当然对了。只要是数值和字符串比较，字符串就会先被转换为数值，然后进行数值比较。对于数值字符串而言，这样能保证结果正确。但如果字符串不能转换成数值呢？比如下面这个例子：

```
let result = "a" < 3; // 因为"a"会转换为NaN，所以结果是false
```

因为字符"a"不能转换成任何有意义的数值，所以只能转换为NaN。这里有一个规则，即任何关系操作符在涉及比较NaN时都返回false。这样一来，下面的例子有趣了：

```
let result1 = NaN < 3;  // false
let result2 = NaN >= 3; // false
```

在大多数比较的场景中，如果一个值不小于另一个值，那就一定大于或等于它。但在比较NaN时，无论是小于还是大于等于，比较的结果都会返回false。

3.5.8 相等操作符

判断两个变量是否相等是编程中最重要的操作之一。在比较字符串、数值和布尔值是否相等时，过程都很直观。但是在比较两个对象是否相等时，情形就比较复杂了。ECMAScript中的相等和不相等操作符，原本在比较之前会执行类型转换，但很快就有人质疑这种转换是否应该发生。最终，ECMAScript提供了两组操作符。第一组是**等于**和**不等于**，它们在比较之前执行转换。第二组是**全等**和**不全等**，它们在比较之前不执行转换。

1. **等于和不等于**

   ECMAScript中的等于操作符用两个等于号（==）表示，如果操作数相等，则会返回true。不等于操作符用叹号和等于号（!=）表示，如果两个操作数不相等，则会返回true。这两个操作符都会先进行类型转换（通常称为**强制类型转换**）再确定操作数是否相等。

   在转换操作数的类型时，相等和不相等操作符遵循如下规则。

   - 如果任一操作数是布尔值，则将其转换为数值再比较是否相等。false转换为0，true转换为1。
   - 如果一个操作数是字符串，另一个操作数是数值，则尝试将字符串转换为数值，再比较是否相等。
   - 如果一个操作数是对象，另一个操作数不是，则调用对象的valueOf()方法取得其原始值，再根据前面的规则进行比较。

   在进行比较时，这两个操作符会遵循如下规则。

   - null和undefined相等。
   - null和undefined不能转换为其他类型的值再进行比较。
   - 如果有任一操作数是NaN，则相等操作符返回false，不相等操作符返回true。记住：即使两个操作数都是NaN，相等操作符也返回false，因为按照规则，NaN不等于NaN。
   - 如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回true。否则，两者不相等。

   下表总结了一些特殊情况及比较的结果。

表达式	结果
null == undefined	true
"NaN" == NaN	false
5 == NaN	false
NaN == NaN	false
NaN != NaN	true
false == 0	true
true == 1	true
true == 2	false
undefined == 0	false
null == 0	false
"5" == 5	true

2. **全等和不全等**

   全等和不全等操作符与相等和不相等操作符类似，只不过它们在比较相等时不转换操作数。全等操作符由3个等于号（===）表示，只有两个操作数在不转换的前提下相等才返回true，比如：

```
let result1 = ("55" == 55);   // true，转换后相等
let result2 = ("55" === 55);  // false，不相等，因为数据类型不同
```

在这个例子中，第一个比较使用相等操作符，比较的是字符串"55"和数值55。如前所述，因为字符串"55"会被转换为数值55，然后再与数值55进行比较，所以返回true。第二个比较使用全等操作符，因为没有转换，字符串和数值当然不能相等，所以返回false。

不全等操作符用一个叹号和两个等于号（!==）表示，只有两个操作数在不转换的前提下不相等才返回true。比如：

```
let result1 = ("55" != 55);  // false，转换后相等
let result2 = ("55" !== 55); // true，不相等，因为数据类型不同
```

这一次，第一个比较使用不相等操作符，它会把字符串"55"转换为数值55，跟第二个操作数相等。既然转换后两个值相等，那就返回false。第二个比较使用不全等操作符。这时候可以这么问："字符串55和数值55有区别吗？"答案是"有"（true）。

另外，虽然null == undefined是true（因为这两个值类似），但null === undefined是false，因为它们不是相同的数据类型。

> **注意** 由于相等和不相等操作符存在类型转换问题，因此推荐使用全等和不全等操作符。这样有助于在代码中保持数据类型的完整性。

3.5.9 条件操作符

条件操作符是ECMAScript中用途最为广泛的操作符之一，语法跟Java中一样：

```
variable = boolean_expression ? true_value : false_value;
```

上面的代码执行了条件赋值操作，即根据条件表达式boolean_expression的值决定将哪个值赋给变量variable。如果boolean_expression是true，则赋值true_value；如果boolean_expression是false，则赋值false_value。比如：

```
let max = (num1 > num2) ? num1 : num2;
```

在这个例子中，max将被赋予一个最大值。这个表达式的意思是，如果num1大于num2（条件表达式为true），则将num1赋给max。否则，如果num1小于num2（条件表达式为false），则将num2赋给max。

3.5.10 赋值操作符

简单赋值用等于号（=）表示，将右手边的值赋给左手边的变量，如下所示：

```
let num = 10;
```

复合赋值使用乘性、加性或位操作符后跟等于号（=）表示。这些赋值操作符是类似如下常见赋值操作的简写形式：

```
let num = 10;
num = num + 10;
```

以上代码的第二行可以通过复合赋值来完成：

```
let num = 10;
num += 10;
```

每个数学操作符以及其他一些操作符都有对应的复合赋值操作符：

- 乘后赋值（*=）
- 除后赋值（/=）
- 取模后赋值（%=）
- 加后赋值（+=）
- 减后赋值（-=）
- 左移后赋值（<<=）
- 右移后赋值（>>=）
- 无符号右移后赋值（>>>=）

这些操作符仅仅是简写语法，使用它们不会提升性能。

3.5.11 逗号操作符

逗号操作符可以用来在一条语句中执行多个操作，如下所示：

```
let num1 = 1, num2 = 2, num3 = 3;
```

在一条语句中同时声明多个变量是逗号操作符最常用的场景。不过，也可以使用逗号操作符来辅助赋值。在赋值时使用逗号操作符分隔值，最终会返回表达式中最后一个值：

```
let num = (5, 1, 4, 8, 0); // num的值为0
```

在这个例子中，num将被赋值为0，因为0是表达式中最后一项。逗号操作符的这种使用场景并不多见，但这种行为的确存在。

3.6 语句

ECMA-262描述了一些语句（也称为**流控制语句**），而ECMAScript中的大部分语法都体现在语句中。语句通常使用一或多个关键字完成既定的任务。语句可以简单，也可以复杂。简单的如告诉函数退出，复杂的如列出一堆要重复执行的指令。

3.6.1 if语句

if语句是使用最频繁的语句之一，语法如下：

```
if (condition) statement1 else statement2
```

这里的条件（condition）可以是任何表达式，并且求值结果不一定是布尔值。ECMAScript会自动调用Boolean()函数将这个表达式的值转换为布尔值。如果条件求值为true，则执行语句statement1；如果条件求值为false，则执行语句statement2。这里的语句可能是一行代码，也可能是一个代码块（即包含在一对花括号中的多行代码）。来看下面的例子：

```
if (i > 25)
  console.log("Greater than 25."); // 只有一行代码的语句
else {
  console.log("Less than or equal to 25."); // 一个语句块
}
```

这里的最佳实践是使用语句块，即使只有一行代码要执行也是如此。这是因为语句块可以避免对什么条件下执行什么产生困惑。

可以像这样连续使用多个if语句：

```
if (condition1) statement1 else if (condition2) statement2 else statement3
```

下面是一个例子：

```
if (i > 25) {
  console.log("Greater than 25.");
} else if (i < 0) {
  console.log("Less than 0.");
} else {
  console.log("Between 0 and 25, inclusive.");
}
```

3.6.2 do-while语句

do-while语句是一种后测试循环语句，即循环体中的代码执行后才会对退出条件进行求值。换句话说，循环体内的代码至少执行一次。do-while的语法如下：

```
do {
  statement
} while (expression);
```

下面是一个例子：

```
let i = 0;
do {
  i += 2;
} while (i < 10);
```

在这个例子中，只要i小于10，循环就会重复执行。i从0开始，每次循环递增2。

> **注意** 后测试循环经常用于这种情形：循环体内代码在退出前至少要执行一次。

3.6.3 while语句

while语句是一种先测试循环语句，即先检测退出条件，再执行循环体内的代码。因此，while循环体内的代码有可能不会执行。下面是while循环的语法：

```
while(expression) statement
```

这是一个例子：

```
let i = 0;
while (i < 10) {
  i += 2;
}
```

在这个例子中，变量i从0开始，每次循环递增2。只要i小于10，循环就会继续。

3.6.4 for语句

for语句也是先测试语句，只不过增加了进入循环之前的初始化代码，以及循环执行后要执行的表达式，语法如下：

```
for (initialization; expression; post-loop-expression) statement
```

下面是一个用例：

```
let count = 10;
for (let i = 0; i < count; i++) {
```

```
  console.log(i);
}
```

以上代码在循环开始前定义了变量i的初始值为0。然后求值条件表达式，如果求值结果为true（i < count），则执行循环体。因此循环体也可能不会被执行。如果循环体被执行了，则循环后表达式也会执行，以便递增变量i。for循环跟下面的while循环是一样的：

```
let count = 10;
let i = 0;
while (i < count) {
  console.log(i);
  i++;
}
```

无法通过while循环实现的逻辑，同样也无法使用for循环实现。因此for循环只是将循环相关的代码封装在了一起而已。

在for循环的初始化代码中，其实是可以不使用变量声明关键字的。不过，初始化定义的迭代器变量在循环执行完成后几乎不可能再用到了。因此，最清晰的写法是使用let声明迭代器变量，这样就可以将这个变量的作用域限定在循环中。

初始化、条件表达式和循环后表达式都不是必需的。因此，下面这种写法可以创建一个无穷循环：

```
for (;;) { // 无穷循环
  doSomething();
}
```

如果只包含条件表达式，那么for循环实际上就变成了while循环：

```
let count = 10;
let i = 0;
for (; i < count; ) {
  console.log(i);
  i++;
}
```

这种多功能性使得for语句在这门语言中使用非常广泛。

3.6.5 for-in语句

for-in语句是一种严格的迭代语句，用于枚举对象中的非符号键属性，语法如下：

```
for (property in expression) statement
```

下面是一个例子：

```
for (const propName in window) {
  document.write(propName);
}
```

这个例子使用for-in循环显示了BOM对象window的所有属性。每次执行循环，都会给变量propName赋予一个window对象的属性作为值，直到window的所有属性都被枚举一遍。与for循环一样，这里控制语句中的const也不是必需的。但为了确保这个局部变量不被修改，推荐使用const。

ECMAScript中对象的属性是无序的，因此for-in语句不能保证返回对象属性的顺序。换句话说，所有可枚举的属性都会返回一次，但返回的顺序可能会因浏览器而异。

如果for-in循环要迭代的变量是null或undefined，则不执行循环体。

3.6.6 for-of语句

for-of语句是一种严格的迭代语句，用于遍历可迭代对象的元素，语法如下：

```
for (property of expression) statement
```

下面是示例：

```
for (const el of [2,4,6,8]) {
  document.write(el);
}
```

在这个例子中，我们使用for-of语句显示了一个包含4个元素的数组中的所有元素。循环会一直持续到将所有元素都迭代完。与for循环一样，这里控制语句中的const也不是必需的。但为了确保这个局部变量不被修改，推荐使用const。

for-of循环会按照可迭代对象的next()方法产生值的顺序迭代元素。关于可迭代对象，本书将在第7章详细介绍。

如果尝试迭代的变量不支持迭代，则for-of语句会抛出错误。

> **注意** ES2018对for-of语句进行了扩展，增加了for-await-of循环，以支持生成期约（promise）的异步可迭代对象。相关内容将在附录A介绍。

3.6.7 标签语句

标签语句用于给语句加标签，语法如下：

```
label: statement
```

下面是一个例子：

```
start: for (let i = 0; i < count; i++) {
  console.log(i);
}
```

在这个例子中，start是一个标签，可以在后面通过break或continue语句引用。标签语句的典型应用场景是嵌套循环。

3.6.8 break和continue语句

break和continue语句为执行循环代码提供了更严格的控制手段。其中，break语句用于立即退出循环，强制执行循环后的下一条语句。而continue语句也用于立即退出循环，但会再次从循环顶部开始执行。下面看一个例子：

```
let num = 0;

for (let i = 1; i < 10; i++) {
  if (i % 5 == 0) {
    break;
  }
  num++;
}

console.log(num); // 4
```

在上面的代码中，for循环会将变量i由1递增到10。而在循环体内，有一个if语句用于检查i能否被5整除（使用取模操作符）。如果是，则执行break语句，退出循环。变量num的初始值为0，表示循环在退出前执行了多少次。当break语句执行后，下一行执行的代码是console.log(num)，显示4。之所以循环执行了4次，是因为当i等于5时，break语句会导致循环退出，该次循环不会执行递增num的代码。如果将break换成continue，则会出现不同的效果：

```
let num = 0;

for (let i = 1; i < 10; i++) {
  if (i % 5 == 0) {
    continue;
  }
  num++;
}

console.log(num); // 8
```

这一次，console.log显示8，即循环被完整执行了8次。当i等于5时，循环会在递增num之前退出，但会执行下一次迭代，此时i是6。然后，循环会一直执行到自然结束，即i等于10。最终num的值是8而不是9，是因为continue语句导致它少递增了一次。

break和continue都可以与标签语句一起使用，返回代码中特定的位置。这通常是在嵌套循环中，如下面的例子所示：

```
let num = 0;

outermost:
for (let i = 0; i < 10; i++) {
  for (let j = 0; j < 10; j++) {
    if (i == 5 && j == 5) {
      break outermost;
    }
    num++;
  }
}

console.log(num); // 55
```

在这个例子中，outermost标签标识的是第一个for语句。正常情况下，每个循环执行10次，意味着num++语句会执行100次，而循环结束时console.log的结果应该是100。但是，break语句带来了一个变数，即要退出到的标签。添加标签不仅让break退出（使用变量j的）内部循环，也会退出（使用变量i）的外部循环。当执行到i和j都等于5时，循环停止执行，此时num的值是55。continue语句也可以使用标签，如下面的例子所示：

```
let num = 0;

outermost:
for (let i = 0; i < 10; i++) {
  for (let j = 0; j < 10; j++) {
    if (i == 5 && j == 5) {
      continue outermost;
    }
    num++;
  }
}

console.log(num); // 95
```

这一次，continue语句会强制循环继续执行，但不是继续执行内部循环，而是继续执行外部循环。当i和j都等于5时，会执行continue，跳到外部循环继续执行，从而导致内部循环少执行5次，结果num等于95。

组合使用标签语句和break、continue能实现复杂的逻辑，但也容易出错。注意标签要使用描述性强的文本，而嵌套也不要太深。

3.6.9 with语句

with语句的用途是将代码作用域设置为特定的对象，其语法是：

```
with (expression) statement;
```

使用with语句的主要场景是针对一个对象反复操作，这时候将代码作用域设置为该对象能提供便利，如下面的例子所示：

```
let qs = location.search.substring(1);
let hostName = location.hostname;
let url = location.href;
```

上面代码中的每一行都用到了location对象。如果使用with语句，就可以少写一些代码：

```
with(location) {
  let qs = search.substring(1);
  let hostName = hostname;
  let url = href;
}
```

这里，with语句用于连接location对象。这意味着在这个语句内部，每个变量首先会被认为是一个局部变量。如果没有找到该局部变量，则会搜索location对象，看它是否有一个同名的属性。如果有，则该变量会被求值为location对象的属性。

严格模式不允许使用with语句，否则会抛出错误。

> **警告** 由于with语句影响性能且难于调试其中的代码，通常不推荐在产品代码中使用with语句。

3.6.10 switch语句

switch语句是与if语句紧密相关的一种流控制语句，从其他语言借鉴而来。ECMAScript中switch语句跟C语言中switch语句的语法非常相似，如下所示：

```
switch (expression) {
  case value1:
    statement
    break;
  case value2:
    statement
    break;
  case value3:
    statement
    break;
  case value4:
    statement
    break;
  default:
    statement
}
```

这里的每个case（条件/分支）相当于：“如果表达式等于后面的值，则执行下面的语句。”break关键字会导致代码执行跳出switch语句。如果没有break，则代码会继续匹配下一个条件。default关键字用于在任何条件都没有满足时指定默认执行的语句（相当于else语句）。

有了switch语句，开发者就用不着写类似这样的代码了：

```
if (i == 25) {
  console.log("25");
} else if (i == 35) {
  console.log("35");
} else if (i == 45) {
  console.log("45");
} else {
  console.log("Other");
}
```

而是可以这样写：

```
switch (i) {
  case 25:
    console.log("25");
    break;
  case 35:
    console.log("35");
    break;
  case 45:
    console.log("45");
    break;
  default:
    console.log("Other");
}
```

为避免不必要的条件判断，最好给每个条件后面都加上break语句。如果确实需要连续匹配几个条件，那么推荐写个注释表明是故意忽略了break，如下所示：

```
switch (i) {
  case 25:
    /*跳过*/
  case 35:
    console.log("25 or 35");
    break;
  case 45:
    console.log("45");
    break;
  default:
    console.log("Other");
}
```

虽然switch语句是从其他语言借鉴过来的，但ECMAScript为它赋予了一些独有的特性。首先，switch语句可以用于所有数据类型（在很多语言中，它只能用于数值），因此可以使用字符串甚至对象。其次，条件的值不需要是常量，也可以是变量或表达式。看下面的例子：

```
switch ("hello world") {
  case "hello" + " world":
    console.log("Greeting was found.");
    break;
  case "goodbye":
    console.log("Closing was found.");
    break;
  default:
    console.log("Unexpected message was found.");
}
```

这个例子在switch语句中使用了字符串。第一个条件实际上使用的是表达式，求值为两个字符串拼接后的结果。因为拼接后的结果等于switch的参数，所以console.log会输出"Greeting was found."。能够在条件判断中使用表达式，就可以在判断中加入更多逻辑：

```
let num = 25;
switch (true) {
  case num < 0:
    console.log("Less than 0.");
    break;
  case num >= 0 && num <= 10:
    console.log("Between 0 and 10.");
    break;
  case num > 10 && num <= 20:
    console.log("Between 10 and 20.");
    break;
  default:
    console.log("More than 20.");
}
```

上面的代码首先在外部定义了变量num，而传给switch语句的参数之所以是true，就是因为每个条件的表达式都会返回布尔值。条件的表达式分别被求值，直到有表达式返回true；否则，就会一直跳到default语句（这个例子正是如此）。

> **注意** switch语句在比较每个条件的值时会使用全等操作符，因此不会强制转换数据类型（比如，字符串"10"不等于数值10）。

3.7 函数

函数对任何语言来说都是核心组件，因为它们可以封装语句，然后在任何地方、任何时间执行。ECMAScript中的函数使用function关键字声明，后跟一组参数，然后是函数体。

> **注意** 第10章会更详细地介绍函数。

以下是函数的基本语法：

```
function functionName(arg0, arg1,...,argN) {
  statements
}
```

下面是一个例子：

```
function sayHi(name, message) {
  console.log("Hello " + name + ", " + message);
}
```

可以通过函数名来调用函数，要传给函数的参数放在括号里（如果有多个参数，则用逗号隔开）。下面是调用函数`sayHi()`的示例：

```
sayHi("Nicholas", "how are you today?");
```

调用这个函数的输出结果是`"Hello Nicholas, how are you today?"`。参数`name`和`message`在函数内部作为字符串被拼接在了一起，最终通过`console.log`输出到控制台。

ECMAScript中的函数不需要指定是否返回值。任何函数在任何时间都可以使用`return`语句来返回函数的值，用法是后跟要返回的值。比如：

```
function sum(num1, num2) {
  return num1 + num2;
}
```

函数`sum()`会将两个值相加并返回结果。注意，除了`return`语句之外没有任何特殊声明表明该函数有返回值。然后就可以这样调用它：

```
const result = sum(5, 10);
```

要注意的是，只要碰到`return`语句，函数就会立即停止执行并退出。因此，`return`语句后面的代码不会被执行。比如：

```
function sum(num1, num2) {
  return num1 + num2;
  console.log("Hello world");  // 不会执行
}
```

在这个例子中，console.log不会执行，因为它在return语句后面。

一个函数里也可以有多个return语句，像这样：

```
function diff(num1, num2) {
  if (num1 < num2) {
    return num2 - num1;
  } else {
    return num1 - num2;
  }
}
```

这个diff()函数用于计算两个数值的差。如果第一个数值小于第二个，则用第二个减第一个；否则，就用第一个减第二个。代码中每个分支都有自己的return语句，返回正确的差值。

return语句也可以不带返回值。这时候，函数会立即停止执行并返回undefined。这种用法最常用于提前终止函数执行，并不是为了返回值。比如在下面的例子中，console.log不会执行：

```
function sayHi(name, message) {
  return;
  console.log("Hello " + name + ", " + message); // 不会执行
}
```

> **注意** 最佳实践是函数要么返回值，要么不返回值。只在某个条件下返回值的函数会带来麻烦，尤其是调试时。

严格模式对函数也有一些限制：

- 函数不能以eval或arguments作为名称；
- 函数的参数不能叫eval或arguments；
- 两个函数的参数不能叫同一个名称。

如果违反上述规则，则会导致语法错误，代码也不会执行。

3.8 小结

JavaScript的核心语言特性在ECMA-262中以伪语言ECMAScript的形式来定义。ECMAScript包含所有基本语法、操作符、数据类型和对象，能完成基本的计算任务，但没有提供获得输入和产生输出的机制。理解ECMAScript及其复杂的细节是完全理解浏览器中JavaScript的关键。下面总结一下ECMAScript中的基本元素。

- ECMAScript中的基本数据类型包括Undefined、Null、Boolean、Number、String和Symbol。
- 与其他语言不同，ECMAScript不区分整数和浮点值，只有Number一种数值数据类型。
- Object是一种复杂数据类型，它是这门语言中所有对象的基类。
- 严格模式为这门语言中某些容易出错的部分施加了限制。
- ECMAScript提供了C语言和类C语言中常见的很多基本操作符，包括数学操作符、布尔操作符、关系操作符、相等操作符和赋值操作符等。
- 这门语言中的流控制语句大多是从其他语言中借鉴而来的，比如if语句、for语句和switch语句等。

ECMAScript中的函数与其他语言中的函数不一样。

- 不需要指定函数的返回值，因为任何函数可以在任何时候返回任何值。
- 不指定返回值的函数实际上会返回特殊值`undefined`。

第 4 章　变量、作用域与内存

> **本章内容**
>
> - 通过变量使用原始值与引用值
> - 理解执行上下文
> - 理解垃圾回收

相比于其他语言，JavaScript中的变量可谓独树一帜。正如ECMA-262所规定的，JavaScript变量是松散类型的，而且变量不过就是特定时间点一个特定值的名称而已。由于没有规则定义变量必须包含什么数据类型，变量的值和数据类型在脚本生命期内可以改变。这样的变量很有意思，很强大，当然也有不少问题。本章会剖析错综复杂的变量。

4.1　原始值与引用值

ECMAScript变量可以包含两种不同类型的数据：原始值和引用值。**原始值**（primitive value）就是最简单的数据，**引用值**（reference value）则是由多个值构成的对象。

在把一个值赋给变量时，JavaScript引擎必须确定这个值是原始值还是引用值。上一章讨论了6种原始值：`Undefined`、`Null`、`Boolean`、`Number`、`String`和`Symbol`。保存原始值的变量是**按值**（by value）访问的，因为我们操作的就是存储在变量中的实际值。

引用值是保存在内存中的对象。与其他语言不同，JavaScript不允许直接访问内存位置，因此也就不能直接操作对象所在的内存空间。在操作对象时，实际上操作的是对该对象的**引用**（reference）而非实际的对象本身。为此，保存引用值的变量是**按引用**（by reference）访问的。

> **注意**　在很多语言中，字符串是使用对象表示的，因此被认为是引用类型。ECMAScript打破了这个惯例。

4.1.1　动态属性

原始值和引用值的定义方式很类似，都是创建一个变量，然后给它赋一个值。不过，在变量保存了这个值之后，可以对这个值做什么，则大有不同。对于引用值而言，可以随时添加、修改和删除其属性和方法。比如，看下面的例子：

```
let person = new Object();
person.name = "Nicholas";
console.log(person.name); // "Nicholas"
```

这里，首先创建了一个对象，并把它保存在变量`person`中。然后，给这个对象添加了一个名为`name`的属性，并给这个属性赋值了一个字符串`"Nicholas"`。在此之后，就可以访问这个新属性，直到对象被销毁或属性被显式地删除。

原始值不能有属性，尽管尝试给原始值添加属性不会报错。比如：

```
let name = "Nicholas";
name.age = 27;
console.log(name.age);  // undefined
```

在此，代码想给字符串name定义一个age属性并给该属性赋值27。紧接着在下一行，属性不见了。记住，只有引用值可以动态添加后面可以使用的属性。

注意，原始类型的初始化可以只使用原始字面量形式。如果使用的是new关键字，则JavaScript会创建一个Object类型的实例，但其行为类似原始值。下面来看看这两种初始化方式的差异：

```
let name1 = "Nicholas";
let name2 = new String("Matt");
name1.age = 27;
name2.age = 26;
console.log(name1.age);    // undefined
console.log(name2.age);    // 26
console.log(typeof name1); // string
console.log(typeof name2); // object
```

4.1.2 复制值

除了存储方式不同，原始值和引用值在通过变量复制时也有所不同。在通过变量把一个原始值赋值到另一个变量时，原始值会被复制到新变量的位置。请看下面的例子：

```
let num1 = 5;
let num2 = num1;
```

这里，num1包含数值5。当把num2初始化为num1时，num2也会得到数值5。这个值跟存储在num1中的5是完全独立的，因为它是那个值的副本。

这两个变量可以独立使用，互不干扰。这个过程如图4-1所示。

复制前的变量对象

num1	5 (Number类型)

复制后的变量对象

num2	5 (Number类型)
num1	5 (Number类型)

图 4-1

在把引用值从一个变量赋给另一个变量时，存储在变量中的值也会被复制到新变量所在的位置。区别在于，这里复制的值实际上是一个指针，它指向存储在堆内存中的对象。操作完成后，两个变量实际上指向同一个对象，因此一个对象上面的变化会在另一个对象上反映出来，如下面的例子所示：

```
let obj1 = new Object();
let obj2 = obj1;
obj1.name = "Nicholas";
console.log(obj2.name); // "Nicholas"
```

在这个例子中，变量obj1保存了一个新对象的实例。然后，这个值被复制到obj2，此时两个变量都指向了同一个对象。在给obj1创建属性name并赋值后，通过obj2也可以访问这个属性，因为它们都指向同一个对象。图4-2展示了变量与堆内存中对象之间的关系。

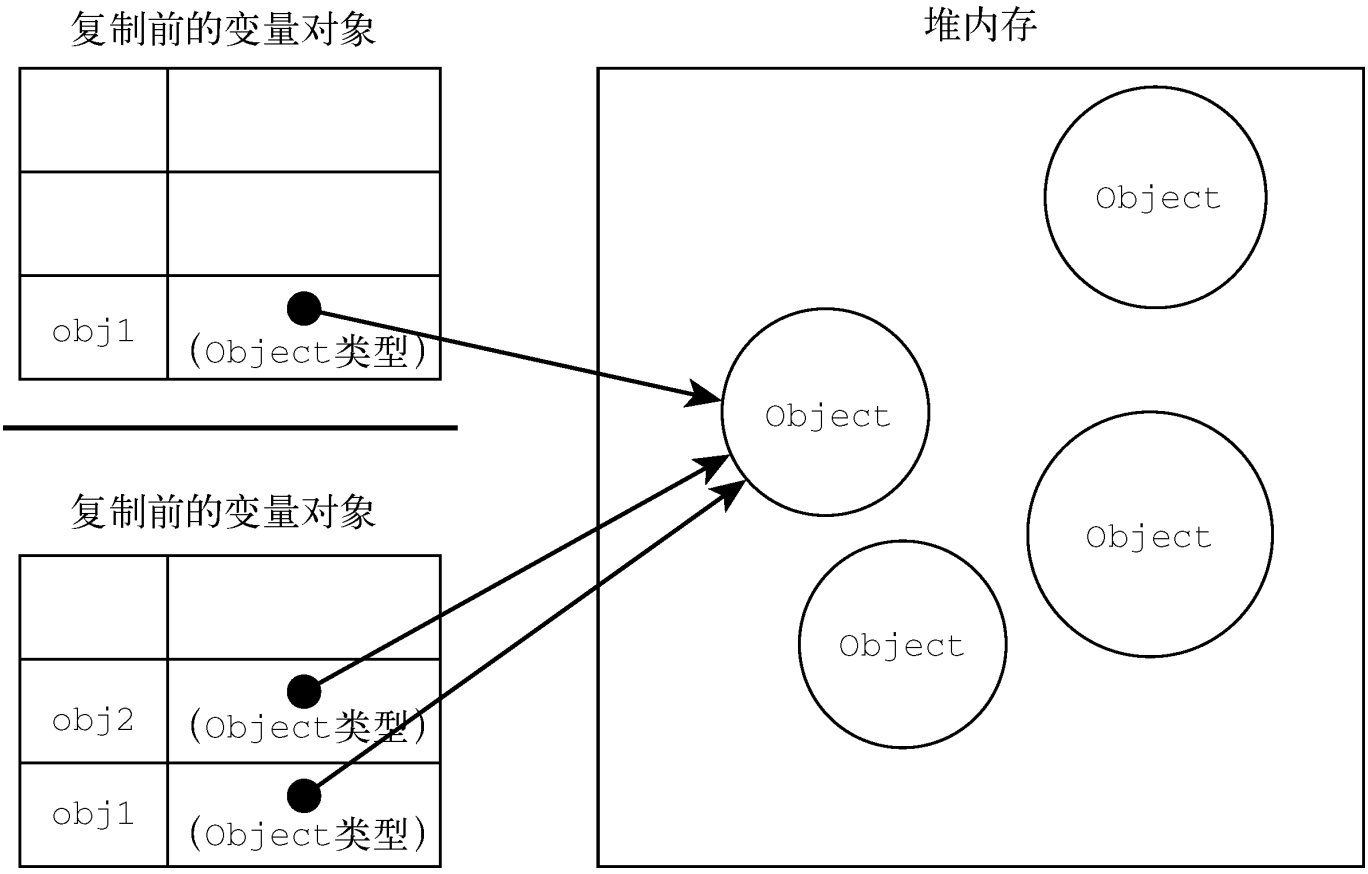


图 4-2

4.1.3 传递参数

ECMAScript中所有函数的参数都是按值传递的。这意味着函数外的值会被复制到函数内部的参数中，就像从一个变量复制到另一个变量一样。如果是原始值，那么就跟原始值变量的复制一样，如果是引用值，那么就跟引用值变量的复制一样。对很多开发者来说，这一块可能会不好理解，毕竟变量有按值和按引用访问，而传参则只有按值传递。

在按值传递参数时，值会被复制到一个局部变量（即一个命名参数，或者用ECMAScript的话说，就是arguments对象中的一个槽位）。在按引用传递参数时，值在内存中的位置会被保存在一个局部变量，这意味着对本地变量的修改会反映到函数外部。（这在ECMAScript中是不可能的。）来看下面这个例子：

```
function addTen(num) {
  num += 10;
  return num;
}

let count = 20;
let result = addTen(count);
console.log(count);  // 20，没有变化
console.log(result); // 30
```

这里，函数addTen()有一个参数num，它其实是一个局部变量。在调用时，变量count作为参数传入。count的值是20，这个值被复制到参数num以便在addTen()内部使用。在函数内部，参数num的值被加上了10，但这不会影响函数外部的原始变量count。参数num和变量count互不干扰，它们只不过碰巧保存了一样的值。如果num是按引用传递的，那么count的值也会被修改为30。这个事实在使用数值这样的原始值时是非常明显的。但是，如果变量中传递的是对象，就没那么清楚了。比如，再看这个例子：

```
function setName(obj) {
  obj.name = "Nicholas";
}

let person = new Object();
setName(person);
console.log(person.name);  // "Nicholas"
```

这一次，我们创建了一个对象并把它保存在变量person中。然后，这个对象被传给setName()方法，并被复制到参数obj中。在函数内部，obj和person都指向同一个对象。结果就是，即使对象是按值传进函数的，obj也会通过引用访问对象。当函数内部给obj设置了name属性时，函数外部的对象也会反映这个变化，因为obj指向的对象保存在全局作用域的堆内存上。很多开发者错误地认为，当在局部作用域中修改对象而变化反映到全局时，就意味着参数是按引用传递的。为证明对象是按值传递的，我们再来看看下面这个修改后的例子：

```
function setName(obj) {
  obj.name = "Nicholas";
  obj = new Object();
  obj.name = "Greg";
}

let person = new Object();
setName(person);
console.log(person.name);  // "Nicholas"
```

这个例子前后唯一的变化就是setName()中多了两行代码，将obj重新定义为一个有着不同name的新对象。当person传入setName()时，其name属性被设置为"Nicholas"。然后变量obj被设置为一个新对象且name属性被设置为"Greg"。如果person是按引用传递的，那么person应该自动将指针改为指向name为"Greg"的对象。可是，当我们再次访问person.name时，它的值是"Nicholas"，这表明函数中参数的值改变之后，原始的引用仍然没变。当obj在函数内部被重写时，它变成了一个指向本地对象的指针。而那个本地对象在函数执行结束时就被销毁了。

> **注意** ECMAScript中函数的参数就是局部变量。

4.1.4 确定类型

前一章提到的typeof操作符最适合用来判断一个变量是否为原始类型。更确切地说，它是判断一个变量是否为字符串、数值、布尔值或undefined的最好方式。如果值是对象或null，那么typeof返回"object"，如下面的例子所示：

```
let s = "Nicholas";
let b = true;
let i = 22;
let u;
let n = null;
let o = new Object();
console.log(typeof s); // string
console.log(typeof i); // number
console.log(typeof b); // boolean
console.log(typeof u); // undefined
console.log(typeof n); // object
console.log(typeof o); // object
```

typeof虽然对原始值很有用，但它对引用值的用处不大。我们通常不关心一个值是不是对象，而是想知道它是什么类型的对象。为了解决这个问题，ECMAScript提供了instanceof操作符，语法如下：

```
result = variable instanceof constructor
```

如果变量是给定引用类型（由其原型链决定，将在第8章详细介绍）的实例，则instanceof操作符返回true。来看下面的例子：

```
console.log(person instanceof Object);  // 变量persion是Object吗?
console.log(colors instanceof Array);   // 变量colors是Array吗?
console.log(pattern instanceof RegExp); // 变量pattern是RegExp吗?
```

按照定义，所有引用值都是Object的实例，因此通过instanceof操作符检测任何引用值和Object构造函数都会返回true。类似地，如果用instanceof检测原始值，则始终会返回false，因为原始值不是对象。

> **注意** typeof操作符在用于检测函数时也会返回"function"。当在Safari（直到Safari 5）和Chrome（直到Chrome 7）中用于检测正则表达式时，由于实现细节的原因，typeof也会返

> 回"function"。ECMA-262规定，任何实现内部[[Call]]方法的对象都应该在typeof检测时返回"function"。因为上述浏览器中的正则表达式实现了这个方法，所以typeof对正则表达式也返回"function"。在IE和Firefox中，typeof对正则表达式返回"object"。

4.2 执行上下文与作用域

执行上下文（以下简称"上下文"）的概念在JavaScript中是颇为重要的。变量或函数的上下文决定了它们可以访问哪些数据，以及它们的行为。每个上下文都有一个关联的**变量对象**（variable object），而这个上下文中定义的所有变量和函数都存在于这个对象上。虽然无法通过代码访问变量对象，但后台处理数据会用到它。

全局上下文是最外层的上下文。根据ECMAScript实现的宿主环境，表示全局上下文的对象可能不一样。在浏览器中，全局上下文就是我们常说的window对象（第12章会详细介绍），因此所有通过var定义的全局变量和函数都会成为window对象的属性和方法。使用let和const的顶级声明不会定义在全局上下文中，但在作用域链解析上效果是一样的。上下文在其所有代码都执行完毕后会被销毁，包括定义在它上面的所有变量和函数（全局上下文在应用程序退出前才会被销毁，比如关闭网页或退出浏览器）。

每个函数调用都有自己的上下文。当代码执行流进入函数时，函数的上下文被推到一个上下文栈上。在函数执行完之后，上下文栈会弹出该函数上下文，将控制权返还给之前的执行上下文。ECMAScript程序的执行流就是通过这个上下文栈进行控制的。

上下文中的代码在执行的时候，会创建变量对象的一个**作用域链**（scope chain）。这个作用域链决定了各级上下文中的代码在访问变量和函数时的顺序。代码正在执行的上下文的变量对象始终位于作用域链的最前端。如果上下文是函数，则其**活动对象**（activation object）用作变量对象。活动对象最初只有一个定义变量：arguments。（全局上下文中没有这个变量。）作用域链中的下一个变量对象来自包含上下文，再下一个对象来自再下一个包含上下文。以此类推直至全局上下文；全局上下文的变量对象始终是作用域链的最后一个变量对象。

代码执行时的标识符解析是通过沿作用域链逐级搜索标识符名称完成的。搜索过程始终从作用域链的最前端开始，然后逐级往后，直到找到标识符。（如果没有找到标识符，那么通常会报错。）

看一看下面这个例子：

```
var color = "blue";

function changeColor() {
  if (color === "blue") {
    color = "red";
  } else {
    color = "blue";
  }
}

changeColor();
```

对这个例子而言，函数changeColor()的作用域链包含两个对象：一个是它自己的变量对象（就是定义arguments对象的那个），另一个是全局上下文的变量对象。这个函数内部之所以能够访问变量color，就是因为可以在作用域链中找到它。

此外，局部作用域中定义的变量可用于在局部上下文中替换全局变量。看一看下面这个例子：

```
var color = "blue";

function changeColor() {
  let anotherColor = "red";

  function swapColors() {
    let tempColor = anotherColor;
    anotherColor = color;
    color = tempColor;

    // 这里可以访问color、anotherColor和tempColor
  }

  // 这里可以访问color和anotherColor，但访问不到tempColor
  swapColors();
}

// 这里只能访问color
changeColor();
```

以上代码涉及3个上下文：全局上下文、changeColor()的局部上下文和swapColors()的局部上下文。全局上下文中有一个变量color和一个函数chageColor()。changeColor()的局部上下文中有一个变量anotherColor和一个函数swapColors()，但在这里可以访问全局上下文中的变量color。swapColors()的局部上下文中有一个变量tempColor，只能在这个上下文中访问到。全局上下文和changeColor()的局部上下文都无法访问到tempColor。而在swapColors()中则可以访问另外两个上下文中的变量，因为它们都是父上下文。图4-3展示了前面这个例子的作用域链。

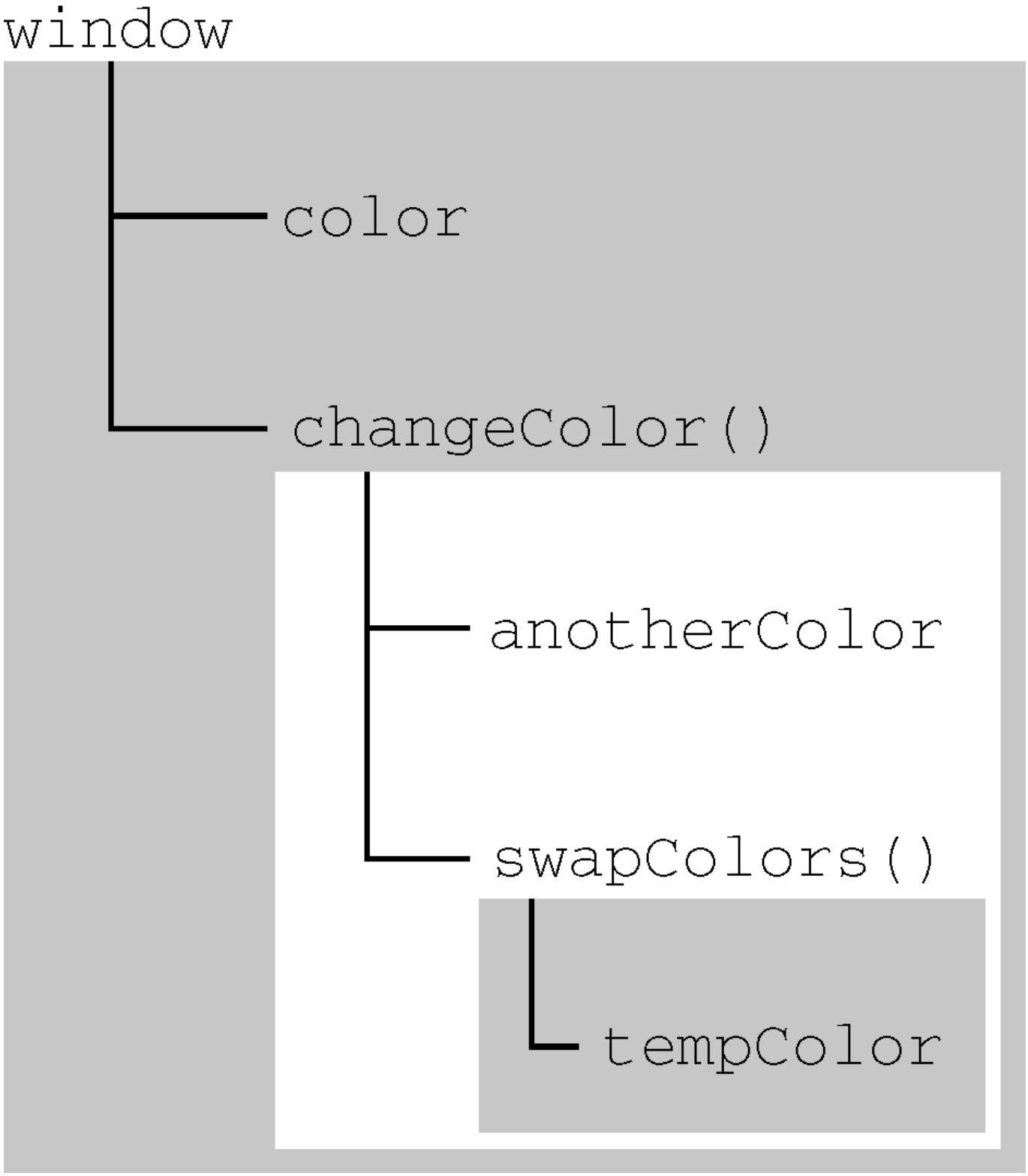


图 4-3

图4-3中的矩形表示不同的上下文。内部上下文可以通过作用域链访问外部上下文中的一切，但外部上下文无法访问内部上下文中的任何东西。上下文之间的连接是线性的、有序的。每个上下文都可以到上一级上下文中去搜索变量和函数，但任何上下文都不能到下一级上下文中去搜索。`swapColors()`局部上下文的作用域链中有3个对象：`swapColors()`的变量对象、`changeColor()`的变量对象和全局变量对象。`swapColors()`的局部上下文首先从自己的变量对象开始搜索变量和函数，搜不到就去搜索上一级变量对象。`changeColor()`上下文的作用域链中只有2个对象：它自己的变量对象和全局变量对象。因此，它不能访问`swapColors()`的上下文。

> **注意** 函数参数被认为是当前上下文中的变量，因此也跟上下文中的其他变量遵循相同的访问规则。

4.2.1 作用域链增强

虽然执行上下文主要有全局上下文和函数上下文两种（eval()调用内部存在第三种上下文），但有其他方式来增强作用域链。某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在代码执行后会被删除。通常在两种情况下会出现这个现象，即代码执行到下面任意一种情况时：

- try/catch语句的catch块
- with语句

这两种情况下，都会在作用域链前端添加一个变量对象。对with语句来说，会向作用域链前端添加指定的对象；对catch语句而言，则会创建一个新的变量对象，这个变量对象会包含要抛出的错误对象的声明。看下面的例子：

```
function buildUrl() {
  let qs = "?debug=true";

  with(location){
    let url = href + qs;
  }

  return url;
}
```

这里，with语句将location对象作为上下文，因此location会被添加到作用域链前端。buildUrl()函数中定义了一个变量qs。当with语句中的代码引用变量href时，实际上引用的是location.href，也就是自己变量对象的属性。在引用qs时，引用的则是定义在buildUrl()中的那个变量，它定义在函数上下文的变量对象上。而在with语句中使用var声明的变量url会成为函数上下文的一部分，可以作为函数的值被返回；但像这里使用let声明的变量url，因为被限制在块级作用域（稍后介绍），所以在with块之外没有定义。

> **注意** IE的实现在IE8之前是有偏差的，即它们会将catch语句中捕获的错误添加到执行上下文的变量对象上，而不是catch语句的变量对象上，导致在catch块外部都可以访问到错误。IE9纠正了这个问题。

4.2.2 变量声明

ES6之后，JavaScript的变量声明经历了翻天覆地的变化。直到ECMAScript 5.1，var都是声明变量的唯一关键字。ES6不仅增加了let和const两个关键字，而且还让这两个关键字压倒性地超越var成为首选。

1. **使用var的函数作用域声明**

   在使用var声明变量时，变量会被自动添加到最接近的上下文。在函数中，最接近的上下文就是函数的局部上下文。在with语句中，最接近的上下文也是函数上下文。如果变量未经声明就被初始化了，那么它就会自动被添加到全局上下文，如下面的例子所示：

   ```
   function add(num1, num2) {
     var sum = num1 + num2;
     return sum;
   }
   ```

```
let result = add(10, 20); // 30
console.log(sum);         // 报错：sum在这里不是有效变量
```

这里，函数add()定义了一个局部变量sum，保存加法操作的结果。这个值作为函数的值被返回，但变量sum在函数外部是访问不到的。如果省略上面例子中的关键字var，那么sum在add()被调用之后就变成可以访问的了，如下所示：

```
function add(num1, num2) {
  sum = num1 + num2;
  return sum;
}

let result = add(10, 20); // 30
console.log(sum);         // 30
```

这一次，变量sum被用加法操作的结果初始化时并没有使用var声明。在调用add()之后，sum被添加到了全局上下文，在函数退出之后依然存在，从而在后面可以访问到。

> **注意** 未经声明而初始化变量是JavaScript编程中一个非常常见的错误，会导致很多问题。为此，读者在初始化变量之前一定要先声明变量。在严格模式下，未经声明就初始化变量会报错。

var声明会被拿到函数或全局作用域的顶部，位于作用域中所有代码之前。这个现象叫作“提升”（hoisting）。提升让同一作用域中的代码不必考虑变量是否已经声明就可以直接使用。可是在实践中，提升也会导致合法却奇怪的现象，即在变量声明之前使用变量。下面的例子展示了在全局作用域中两段等价的代码：

```
var name = "Jake";

// 等价于：

name = 'Jake';
var name;
```

下面是两个等价的函数：

```
function fn1() {
  var name = 'Jake';
}

// 等价于：
function fn2() {
  var name;
  name = 'Jake';
}
```

通过在声明之前打印变量，可以验证变量会被提升。声明的提升意味着会输出`undefined`而不是`Reference Error`：

```
console.log(name); // undefined
var name = 'Jake';

function() {
  console.log(name); // undefined
  var name = 'Jake';
}
```

2. **使用`let`的块级作用域声明**

ES6新增的`let`关键字跟`var`很相似，但它的作用域是块级的，这也是JavaScript中的新概念。块级作用域由最近的一对包含花括号`{}`界定。换句话说，`if`块、`while`块、`function`块，甚至连单独的块也是`let`声明变量的作用域。

```
if (true) {
  let a;
}
console.log(a); // ReferenceError: a没有定义

while (true) {
  let b;
}
console.log(b); // ReferenceError: b没有定义

function foo() {
  let c;
}
console.log(c); // ReferenceError: c没有定义
                // 这没什么可奇怪的
                // var声明也会导致报错

// 这不是对象字面量，而是一个独立的块
// JavaScript解释器会根据其中内容识别出它来
{
  let d;
}
console.log(d); // ReferenceError: d没有定义
```

`let`与`var`的另一个不同之处是在同一作用域内不能声明两次。重复的`var`声明会被忽略，而重复的`let`声明会抛出`SyntaxError`。

```
var a;
var a;
// 不会报错
```

```
{
  let b;
  let b;
}
// SyntaxError: 标识符b已经声明过了
```

let的行为非常适合在循环中声明迭代变量。使用var声明的迭代变量会泄漏到循环外部，这种情况应该避免。来看下面两个例子：

```
for (var i = 0; i < 10; ++i) {}
console.log(i); // 10

for (let j = 0; j < 10; ++j) {}
console.log(j); // ReferenceError: j没有定义
```

严格来讲，let在JavaScript运行时中也会被提升，但由于"暂时性死区"（temporal dead zone）的缘故，实际上不能在声明之前使用let变量。因此，从写JavaScript代码的角度说，let的提升跟var是不一样的。

3. **使用const的常量声明**

除了let，ES6同时还增加了const关键字。使用const声明的变量必须同时初始化为某个值。一经声明，在其生命周期的任何时候都不能再重新赋予新值。

```
const a; // SyntaxError: 常量声明时没有初始化

const b = 3;
console.log(b); // 3
b = 4; // TypeError: 给常量赋值
```

const除了要遵循以上规则，其他方面与let声明是一样的：

```
if (true) {
  const a = 0;
}
console.log(a); // ReferenceError: a没有定义

while (true) {
  const b = 1;
}
console.log(b); // ReferenceError: b没有定义

function foo() {
  const c = 2;
}
console.log(c); // ReferenceError: c没有定义
```

```
{
  const d = 3;
}
console.log(d); // ReferenceError: d没有定义
```

const声明只应用到顶级原语或者对象。换句话说，赋值为对象的const变量不能再被重新赋值为其他引用值，但对象的键则不受限制。

```
const o1 = {};
o1 = {}; // TypeError: 给常量赋值

const o2 = {};
o2.name = 'Jake';
console.log(o2.name); // 'Jake'
```

如果想让整个对象都不能修改，可以使用Object.freeze()，这样再给属性赋值时虽然不会报错，但会静默失败：

```
const o3 = Object.freeze({});
o3.name = 'Jake';
console.log(o3.name); // undefined
```

由于const声明暗示变量的值是单一类型且不可修改，JavaScript运行时编译器可以将其所有实例都替换成实际的值，而不会通过查询表进行变量查找。谷歌的V8引擎就执行这种优化。

> **注意** 开发实践表明，如果开发流程并不会因此而受很大影响，就应该尽可能地多使用const声明，除非确实需要一个将来会重新赋值的变量。这样可以从根本上保证提前发现重新赋值导致的bug。

4. **标识符查找**

当在特定上下文中为读取或写入而引用一个标识符时，必须通过搜索确定这个标识符表示什么。搜索开始于作用域链前端，以给定的名称搜索对应的标识符。如果在局部上下文中找到该标识符，则搜索停止，变量确定；如果没有找到变量名，则继续沿作用域链搜索。（注意，作用域链中的对象也有一个原型链，因此搜索可能涉及每个对象的原型链。）这个过程一直持续到搜索至全局上下文的变量对象。如果仍然没有找到标识符，则说明其未声明。

为更好地说明标识符查找，我们来看一个例子：

```
var color = 'blue';

function getColor() {
  return color;
}

console.log(getColor()); // 'blue'
```

在这个例子中，调用函数getColor()时会引用变量color。为确定color的值会进行两步搜索。第一步，搜索getColor()的变量对象，查找名为color的标识符。结果没找到，于是继续搜索下一个变量对象（来自全局上下文），然后就找到了名为color的标识符。因为全局变量对象上有color的定义，所以搜索结束。

对这个搜索过程而言，引用局部变量会让搜索自动停止，而不继续搜索下一级变量对象。也就是说，如果局部上下文中有一个同名的标识符，那就不能在该上下文中引用父上下文中的同名标识符，如下面的例子所示：

```
var color = 'blue';

function getColor() {
  let color = 'red';
  return color;
}

console.log(getColor()); // 'red'
```

使用块级作用域声明并不会改变搜索流程，但可以给词法层级添加额外的层次：

```
var color = 'blue';

function getColor() {
  let color = 'red';
  {
    let color = 'green';
    return color;
  }
}

console.log(getColor()); // 'green'
```

在这个修改后的例子中，getColor()内部声明了一个名为color的局部变量。在调用这个函数时，变量会被声明。在执行到函数返回语句时，代码引用了变量color。于是开始在局部上下文中搜索这个标识符，结果找到了值为'green'的变量color。因为变量已找到，搜索随即停止，所以就使用这个局部变量。这意味着函数会返回'green'。在局部变量color声明之后的任何代码都无法访问全局变量color，除非使用完全限定的写法window.color。

> **注意** 标识符查找并非没有代价。访问局部变量比访问全局变量要快，因为不用切换作用域。不过，JavaScript引擎在优化标识符查找上做了很多工作，将来这个差异可能就微不足道了。

4.3 垃圾回收

JavaScript是使用垃圾回收的语言，也就是说执行环境负责在代码执行时管理内存。在C和C++等语言中，跟踪内存使用对开发者来说是个很大的负担，也是很多问题的来源。JavaScript为开发者卸下了这个负担，通过自动内存管理实现内存分配和闲置资源回收。基本思路很简单：确定哪个变量不会再使用，然后释放它占用的内

存。这个过程是周期性的，即垃圾回收程序每隔一定时间（或者说在代码执行过程中某个预定的收集时间）就会自动运行。垃圾回收过程是一个近似且不完美的方案，因为某块内存是否还有用，属于“不可判定的”问题，意味着靠算法是解决不了的。

我们以函数中局部变量的正常生命周期为例。函数中的局部变量会在函数执行时存在。此时，栈（或堆）内存会分配空间以保存相应的值。函数在内部使用了变量，然后退出。此时，就不再需要那个局部变量了，它占用的内存可以释放，供后面使用。这种情况下显然不再需要局部变量了，但并不是所有时候都会这么明显。垃圾回收程序必须跟踪记录哪个变量还会使用，以及哪个变量不会再使用，以便回收内存。如何标记未使用的变量也许有不同的实现方式。不过，在浏览器的发展史上，用到过两种主要的标记策略：标记清理和引用计数。

4.3.1 标记清理

JavaScript最常用的垃圾回收策略是**标记清理**（mark-and-sweep）。当变量进入上下文，比如在函数内部声明一个变量时，这个变量会被加上存在于上下文中的标记。而不在上下文中的变量，逻辑上讲，永远不应该释放它们的内存，因为只要上下文中的代码在运行，就有可能用到它们。当变量离开上下文时，也会被加上离开上下文的标记。

给变量加标记的方式有很多种。比如，当变量进入上下文时，反转某一位；或者可以维护“在上下文中”和“不在上下文中”两个变量列表，可以把变量从一个列表转移到另一个列表。标记过程的实现并不重要，关键是策略。

垃圾回收程序运行的时候，会标记内存中存储的所有变量（记住，标记方法有很多种）。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉。在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了。随后垃圾回收程序做一次**内存清理**，销毁带标记的所有值并收回它们的内存。

到了2008年，IE、Firefox、Opera、Chrome和Safari都在自己的JavaScript实现中采用标记清理（或其变体），只是在运行垃圾回收的频率上有所差异。

4.3.2 引用计数

另一种没那么常用的垃圾回收策略是**引用计数**（reference counting）。其思路是对每个值都记录它被引用的次数。声明变量并给它赋一个引用值时，这个值的引用数为1。如果同一个值又被赋给另一个变量，那么引用数加1。类似地，如果保存对该值引用的变量被其他值给覆盖了，那么引用数减1。当一个值的引用数为0时，就说明没办法再访问到这个值了，因此可以安全地收回其内存了。垃圾回收程序下次运行的时候就会释放引用数为0的值的内存。

引用计数最早由Netscape Navigator 3.0采用，但很快就遇到了严重的问题：循环引用。所谓**循环引用**，就是对象A有一个指针指向对象B，而对象B也引用了对象A。比如：

```
function problem() {
  let objectA = new Object();
  let objectB = new Object();

  objectA.someOtherObject = objectB;
  objectB.anotherObject = objectA;
}
```

在这个例子中，`objectA`和`objectB`通过各自的属性相互引用，意味着它们的引用数都是2。在标记清理策略下，这不是问题，因为在函数结束后，这两个对象都不在作用域中。而在引用计数策略下，`objectA`和

objectB在函数结束后还会存在，因为它们的引用数永远不会变成0。如果函数被多次调用，则会导致大量内存永远不会被释放。为此，Netscape在4.0版放弃了引用计数，转而采用标记清理。事实上，引用计数策略的问题还不止于此。

在IE8及更早版本的IE中，并非所有对象都是原生JavaScript对象。BOM和DOM中的对象是C++实现的组件对象模型（COM，Component Object Model）对象，而COM对象使用引用计数实现垃圾回收。因此，即使这些版本IE的JavaScript引擎使用标记清理，JavaScript存取的COM对象依旧使用引用计数。换句话说，只要涉及COM对象，就无法避开循环引用问题。下面这个简单的例子展示了涉及COM对象的循环引用问题：

```
let element = document.getElementById("some_element");
let myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

这个例子在一个DOM对象（element）和一个原生JavaScript对象（myObject）之间制造了循环引用。myObject变量有一个名为element的属性指向DOM对象element，而element对象有一个someObject属性指回myObject对象。由于存在循环引用，因此DOM元素的内存永远不会被回收，即使它已经被从页面上删除了也是如此。

为避免类似的循环引用问题，应该在确保不使用的情况下切断原生JavaScript对象与DOM元素之间的连接。比如，通过以下代码可以清除前面的例子中建立的循环引用：

```
myObject.element = null;
element.someObject = null;
```

把变量设置为null实际上会切断变量与其之前引用值之间的关系。当下次垃圾回收程序运行时，这些值就会被删除，内存也会被回收。

为了补救这一点，IE9把BOM和DOM对象都改成了JavaScript对象，这同时也避免了由于存在两套垃圾回收算法而导致的问题，还消除了常见的内存泄漏现象。

> **注意** 还有其他一些可能导致循环引用的情形，本书后面会介绍到。

4.3.3 性能

垃圾回收程序会周期性运行，如果内存中分配了很多变量，则可能造成性能损失，因此垃圾回收的时间调度很重要。尤其是在内存有限的移动设备上，垃圾回收有可能会明显拖慢渲染的速度和帧速率。开发者不知道什么时候运行时会收集垃圾，因此最好的办法是在写代码时就要做到：无论什么时候开始收集垃圾，都能让它尽快结束工作。

现代垃圾回收程序会基于对JavaScript运行时环境的探测来决定何时运行。探测机制因引擎而异，但基本上都是根据已分配对象的大小和数量来判断的。比如，根据V8团队2016年的一篇博文的说法：“在一次完整的垃圾回收之后，V8的堆增长策略会根据活跃对象的数量外加一些余量来确定何时再次垃圾回收。”

由于调度垃圾回收程序方面的问题会导致性能下降，IE曾饱受诟病。它的策略是根据分配数，比如分配了256个变量、4096个对象/数组字面量和数组槽位（slot），或者64KB字符串。只要满足其中某个条件，垃圾回收程序

就会运行。这样实现的问题在于，分配那么多变量的脚本，很可能在其整个生命周期内始终需要那么多变量，结果就会导致垃圾回收程序过于频繁地运行。由于对性能的严重影响，IE7最终更新了垃圾回收程序。

IE7发布后，JavaScript引擎的垃圾回收程序被调优为动态改变分配变量、字面量或数组槽位等会触发垃圾回收的阈值。IE7的起始阈值都与IE6的相同。如果垃圾回收程序回收的内存不到已分配的15%，这些变量、字面量或数组槽位的阈值就会翻倍。如果有一次回收的内存达到已分配的85%，则阈值重置为默认值。这么一个简单的修改，极大地提升了重度依赖JavaScript的网页在浏览器中的性能。

> **警告** 在某些浏览器中是有可能（但不推荐）主动触发垃圾回收的。在IE中，`window.CollectGarbage()`方法会立即触发垃圾回收。在Opera 7及更高版本中，调用`window.opera.collect()`也会启动垃圾回收程序。

4.3.4 内存管理

在使用垃圾回收的编程环境中，开发者通常无须关心内存管理。不过，JavaScript运行在一个内存管理与垃圾回收都很特殊的环境。分配给浏览器的内存通常比分配给桌面软件的要少很多，分配给移动浏览器的就更少了。这更多出于安全考虑而不是别的，就是为了避免运行大量JavaScript的网页耗尽系统内存而导致操作系统崩溃。这个内存限制不仅影响变量分配，也影响调用栈以及能够同时在一个线程中执行的语句数量。

将内存占用量保持在一个较小的值可以让页面性能更好。优化内存占用的最佳手段就是保证在执行代码时只保存必要的数据。如果数据不再必要，那么把它设置为`null`，从而释放其引用。这也可以叫作**解除引用**。这个建议最适合全局变量和全局对象的属性。局部变量在超出作用域后会被自动解除引用，如下面的例子所示：

```
function createPerson(name){
  let localPerson = new Object();
  localPerson.name = name;
  return localPerson;
}

let globalPerson = createPerson("Nicholas");

// 解除globalPerson对值的引用

globalPerson = null;
```

在上面的代码中，变量`globalPerson`保存着`createPerson()`函数调用返回的值。在`createPerson()`内部，`localPerson`创建了一个对象并给它添加了一个`name`属性。然后，`localPerson`作为函数值被返回，并被赋值给`globalPerson`。`localPerson`在`createPerson()`执行完成超出上下文后会自动被解除引用，不需要显式处理。但`globalPerson`是一个全局变量，应该在不再需要时手动解除其引用，最后一行就是这么做的。

不过要注意，解除对一个值的引用并不会自动导致相关内存被回收。解除引用的关键在于确保相关的值已经不在上下文里了，因此它在下次垃圾回收时会被回收。

1. **通过`const`和`let`声明提升性能**

   ES6增加这两个关键字不仅有助于改善代码风格，而且同样有助于改进垃圾回收的过程。因为`const`和`let`都以块（而非函数）为作用域，所以相比于使用`var`，使用这两个新关键字可能会更早地让垃圾回收程序介入，尽早回收应该回收的内存。在块作用域比函数作用域更早终止的情况下，这就有可能发生。

2. **隐藏类和删除操作**

根据JavaScript所在的运行环境，有时候需要根据浏览器使用的JavaScript引擎来采取不同的性能优化策略。截至2017年，Chrome是最流行的浏览器，使用V8 JavaScript引擎。V8在将解释后的JavaScript代码编译为实际的机器码时会利用“隐藏类”。如果你的代码非常注重性能，那么这一点可能对你很重要。

运行期间，V8会将创建的对象与隐藏类关联起来，以跟踪它们的属性特征。能够共享相同隐藏类的对象性能会更好，V8会针对这种情况进行优化，但不一定总能够做到。比如下面的代码：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
}

let a1 = new Article();
let a2 = new Article();
```

V8会在后台配置，让这两个类实例共享相同的隐藏类，因为这两个实例共享同一个构造函数和原型。假设之后又添加了下面这行代码：

```
a2.author = 'Jake';
```

此时两个`Article`实例就会对应两个不同的隐藏类。根据这种操作的频率和隐藏类的大小，这有可能对性能产生明显影响。

当然，解决方案就是避免JavaScript的“先创建再补充”（ready-fire-aim）式的动态属性赋值，并在构造函数中一次性声明所有属性，如下所示：

```
function Article(opt_author) {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = opt_author;
}

let a1 = new Article();
let a2 = new Article('Jake');
```

这样，两个实例基本上就一样了（不考虑`hasOwnProperty`的返回值），因此可以共享一个隐藏类，从而带来潜在的性能提升。不过要记住，使用`delete`关键字会导致生成相同的隐藏类片段。看一下这个例子：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}

let a1 = new Article();
```

```
let a2 = new Article();

delete a1.author;
```

在代码结束后，即使两个实例使用了同一个构造函数，它们也不再共享一个隐藏类。动态删除属性与动态添加属性导致的后果一样。最佳实践是把不想要的属性设置为null。这样可以保持隐藏类不变和继续共享，同时也能达到删除引用值供垃圾回收程序回收的效果。比如：

```
function Article() {
  this.title = 'Inauguration Ceremony Features Kazoo Band';
  this.author = 'Jake';
}

let a1 = new Article();
let a2 = new Article();

a1.author = null;
```

3. **内存泄漏**

写得不好的JavaScript可能出现难以察觉且有害的内存泄漏问题。在内存有限的设备上，或者在函数会被调用很多次的情况下，内存泄漏可能是个大问题。JavaScript中的内存泄漏大部分是由不合理的引用导致的。

意外声明全局变量是最常见但也最容易修复的内存泄漏问题。下面的代码没有使用任何关键字声明变量：

```
function setName() {
  name = 'Jake';
}
```

此时，解释器会把变量name当作window的属性来创建（相当于window.name = 'Jake'）。可想而知，在window对象上创建的属性，只要window本身不被清理就不会消失。这个问题很容易解决，只要在变量声明前头加上var、let或const关键字即可，这样变量就会在函数执行完毕后离开作用域。

定时器也可能会悄悄地导致内存泄漏。下面的代码中，定时器的回调通过闭包引用了外部变量：

```
let name = 'Jake';
setInterval(() => {
  console.log(name);
}, 100);
```

只要定时器一直运行，回调函数中引用的name就会一直占用内存。垃圾回收程序当然知道这一点，因而就不会清理外部变量。

使用JavaScript闭包很容易在不知不觉间造成内存泄漏。请看下面的例子：

```
let outer = function() {
  let name = 'Jake';
  return function() {
    return name;
  };
};
```

这会导致分配给name的内存被泄漏。以上代码创建了一个内部闭包，只要outer函数存在就不能清理name，因为闭包一直在引用着它。假如name的内容很大（不止是一个小字符串），那可能就是个大问题了。

4. **静态分配与对象池**

为了提升JavaScript性能，最后要考虑的一点往往就是压榨浏览器了。此时，一个关键问题就是如何减少浏览器执行垃圾回收的次数。开发者无法直接控制什么时候开始收集垃圾，但可以间接控制触发垃圾回收的条件。理论上，如果能够合理使用分配的内存，同时避免多余的垃圾回收，那就可以保住因释放内存而损失的性能。

浏览器决定何时运行垃圾回收程序的一个标准就是对象更替的速度。如果有很多对象被初始化，然后一下子又都超出了作用域，那么浏览器就会采用更激进的方式调度垃圾回收程序运行，这样当然会影响性能。看一看下面的例子，这是一个计算二维矢量加法的函数：

```
function addVector(a, b) {
  let resultant = new Vector();
  resultant.x = a.x + b.x;
  resultant.y = a.y + b.y;
  return resultant;
}
```

调用这个函数时，会在堆上创建一个新对象，然后修改它，最后再把它返回给调用者。如果这个矢量对象的生命周期很短，那么它会很快失去所有对它的引用，成为可以被回收的值。假如这个矢量加法函数频繁被调用，那么垃圾回收调度程序会发现这里对象更替的速度很快，从而会更频繁地安排垃圾回收。

该问题的解决方案是不要动态创建矢量对象，比如可以修改上面的函数，让它使用一个已有的矢量对象：

```
function addVector(a, b, resultant) {
  resultant.x = a.x + b.x;
  resultant.y = a.y + b.y;
  return resultant;
}
```

当然，这需要在其他地方实例化矢量参数resultant，但这个函数的行为没有变。那么在哪里创建矢量可以不让垃圾回收调度程序盯上呢？

一个策略是使用对象池。在初始化的某一时刻，可以创建一个对象池，用来管理一组可回收的对象。应用程序可以向这个对象池请求一个对象、设置其属性、使用它，然后在操作完成后再把它还给对象池。由于没发生对象初始化，垃圾回收探测就不会发现有对象更替，因此垃圾回收程序就不会那么频繁地运行。下面是一个对象池的伪实现：

```
// vectorPool是已有的对象池
let v1 = vectorPool.allocate();
let v2 = vectorPool.allocate();
let v3 = vectorPool.allocate();

v1.x = 10;
v1.y = 5;
v2.x = -3;
v2.y = -6;

addVector(v1, v2, v3);

console.log([v3.x, v3.y]); // [7, -1]

vectorPool.free(v1);
vectorPool.free(v2);
vectorPool.free(v3);

// 如果对象有属性引用了其他对象
// 则这里也需要把这些属性设置为null
v1 = null;
v2 = null;
v3 = null;
```

如果对象池只按需分配矢量（在对象不存在时创建新的，在对象存在时则复用存在的），那么这个实现本质上是一种贪婪算法，有单调增长但为静态的内存。这个对象池必须使用某种结构维护所有对象，数组是比较好的选择。不过，使用数组来实现，必须留意不要招致额外的垃圾回收。比如下面这个例子：

```
let vectorList = new Array(100);
let vector = new Vector();
vectorList.push(vector);
```

由于JavaScript数组的大小是动态可变的，引擎会删除大小为100的数组，再创建一个新的大小为200的数组。垃圾回收程序会看到这个删除操作，说不定因此很快就会跑来收一次垃圾。要避免这种动态分配操作，可以在初始化时就创建一个大小够用的数组，从而避免上述先删除再创建的操作。不过，必须事先想好这个数组有多大。

> **注意** 静态分配是优化的一种极端形式。如果你的应用程序被垃圾回收严重地拖了后腿，可以利用它提升性能。但这种情况并不多见。大多数情况下，这都属于过早优化，因此不用考虑。

4.4 小结

JavaScript变量可以保存两种类型的值：原始值和引用值。原始值可能是以下6种原始数据类型之一：`Undefined`、`Null`、`Boolean`、`Number`、`String`和`Symbol`。原始值和引用值有以下特点。

- 原始值大小固定，因此保存在栈内存上。
- 从一个变量到另一个变量复制原始值会创建该值的第二个副本。
- 引用值是对象，存储在堆内存上。
- 包含引用值的变量实际上只包含指向相应对象的一个指针，而不是对象本身。
- 从一个变量到另一个变量复制引用值只会复制指针，因此结果是两个变量都指向同一个对象。
- `typeof`操作符可以确定值的原始类型，而`instanceof`操作符用于确保值的引用类型。

任何变量（不管包含的是原始值还是引用值）都存在于某个执行上下文中（也称为作用域）。这个上下文（作用域）决定了变量的生命周期，以及它们可以访问代码的哪些部分。执行上下文可以总结如下。

- 执行上下文分全局上下文、函数上下文和块级上下文。
- 代码执行流每进入一个新上下文，都会创建一个作用域链，用于搜索变量和函数。
- 函数或块的局部上下文不仅可以访问自己作用域内的变量，而且也可以访问任何包含上下文乃至全局上下文中的变量。
- 全局上下文只能访问全局上下文中的变量和函数，不能直接访问局部上下文中的任何数据。
- 变量的执行上下文用于确定什么时候释放内存。

JavaScript是使用垃圾回收的编程语言，开发者不需要操心内存分配和回收。JavaScript的垃圾回收程序可以总结如下。

- 离开作用域的值会被自动标记为可回收，然后在垃圾回收期间被删除。
- 主流的垃圾回收算法是标记清理，即先给当前不使用的值加上标记，再回来回收它们的内存。
- 引用计数是另一种垃圾回收策略，需要记录值被引用了多少次。JavaScript引擎不再使用这种算法，但某些旧版本的IE仍然会受这种算法的影响，原因是JavaScript会访问非原生JavaScript对象（如DOM元素）。
- 引用计数在代码中存在循环引用时会出现问题。
- 解除变量的引用不仅可以消除循环引用，而且对垃圾回收也有帮助。为促进内存回收，全局对象、全局对象的属性和循环引用都应该在不需要时解除引用。

第 5 章　基本引用类型

> **本章内容**
>
> - 理解对象
> - 基本JavaScript数据类型
> - 原始值与原始值包装类型

引用值（或者对象）是某个特定**引用类型**的实例。在ECMAScript中，引用类型是把数据和功能组织到一起的结构，经常被人错误地称作“类”。虽然从技术上讲JavaScript是一门面向对象语言，但ECMAScript缺少传统的面向对象编程语言所具备的某些基本结构，包括类和接口。引用类型有时候也被称为**对象定义**，因为它们描述了自己的对象应有的属性和方法。

> **注意**　引用类型虽然有点像类，但跟类并不是一个概念。为避免混淆，本章后面不会使用术语“类”。

对象被认为是某个特定引用类型的**实例**。新对象通过使用`new`操作符后跟一个**构造函数**（constructor）来创建。构造函数就是用来创建新对象的函数，比如下面这行代码：

```
let now = new Date();
```

这行代码创建了引用类型Date的一个新实例，并将它保存在变量now中。Date()在这里就是构造函数，它负责创建一个只有默认属性和方法的简单对象。ECMAScript提供了很多像Date这样的原生引用类型，帮助开发者实现常见的任务。

> **注意** 函数也是一种引用类型，但有关函数的内容太多了，一章放不下，所以本书专门用第10章来介绍函数。

5.1 Date

ECMAScript的Date类型参考了Java早期版本中的java.util.Date。为此，Date类型将日期保存为自协调世界时（UTC，Universal Time Coordinated）时间1970年1月1日午夜（零时）至今所经过的毫秒数。使用这种存储格式，Date类型可以精确表示1970年1月1日之前及之后285 616年的日期。

要创建日期对象，就使用new操作符来调用Date构造函数：

```
let now = new Date();
```

在不给Date构造函数传参数的情况下，创建的对象将保存当前日期和时间。要基于其他日期和时间创建日期对象，必须传入其毫秒表示（UNIX纪元1970年1月1日午夜之后的毫秒数）。ECMAScript为此提供了两个辅助方法：Date.parse()和Date.UTC()。

Date.parse()方法接收一个表示日期的字符串参数，尝试将这个字符串转换为表示该日期的毫秒数。ECMA-262第5版定义了Date.parse()应该支持的日期格式，填充了第3版遗留的空白。所有实现都必须支持下列日期格式：

- “月/日/年”，如"5/23/2019"；
- “月名 日, 年”，如"May 23, 2019"；
- “周几 月名 日 年 时:分:秒 时区”，如"Tue May 23 2019 00:00:00 GMT-0700"；
- ISO 8601扩展格式“YYYY-MM-DDTHH:mm:ss.sssZ”，如2019-05-23T00:00:00（只适用于兼容ES5的实现）。

比如，要创建一个表示“2019年5月23日”的日期对象，可以使用以下代码：

```
let someDate = new Date(Date.parse("May 23, 2019"));
```

如果传给Date.parse()的字符串并不表示日期，则该方法会返回NaN。如果直接把表示日期的字符串传给Date构造函数，那么Date会在后台调用Date.parse()。换句话说，下面这行代码跟前面那行代码是等价的：

```
let someDate = new Date("May 23, 2019");
```

这两行代码得到的日期对象相同。

> **注意** 不同的浏览器对Date类型的实现有很多问题。比如，很多浏览器会选择用当前日期替代越界的日期，因此有些浏览器会将"January 32, 2019"解释为"February 1, 2019"。Opera则会插入当前月的当前日，返回"January 当前日, 2019"。就是说，如果是在9月21日运行代码，会返回"January 21, 2019"。

Date.UTC()方法也返回日期的毫秒表示，但使用的是跟Date.parse()不同的信息来生成这个值。传给Date.UTC()的参数是年、零起点月数（1月是0，2月是1，以此类推）、日（1~31）、时（0~23）、分、秒和毫秒。这些参数中，只有前两个（年和月）是必需的。如果不提供日，那么默认为1日。其他参数的默认值都是0。下面是使用Date.UTC()的两个例子：

```
// GMT时间2000年1月1日零点
let y2k = new Date(Date.UTC(2000, 0));

// GMT时间2005年5月5日下午5点55分55秒
let allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

这个例子创建了两个日期 。第一个日期是2000年1月1日零点（GMT），2000代表年，0代表月（1月）。因为没有其他参数（日取1，其他取0），所以结果就是该月第1天零点。第二个日期表示2005年5月5日下午5点55分55秒（GMT）。虽然日期里面涉及的都是5，但月数必须用4，因为月数是零起点的。小时也必须是17，因为这里采用的是24小时制，即取值范围是0~23。其他参数就都很直观了。

与Date.parse()一样，Date.UTC()也会被Date构造函数隐式调用，但有一个区别：这种情况下创建的是本地日期，不是GMT日期。不过Date构造函数跟Date.UTC()接收的参数是一样的。因此，如果第一个参数是数值，则构造函数假设它是日期中的年，第二个参数就是月，以此类推。前面的例子也可以这样来写：

```
// 本地时间2000年1月1日零点
let y2k = new Date(2000, 0);

// 本地时间2005年5月5日下午5点55分55秒
let allFives = new Date(2005, 4, 5, 17, 55, 55);
```

以上代码创建了与前面例子中相同的两个日期，但这次的两个日期是（由于系统设置决定的）本地时区的日期。

ECMAScript还提供了Date.now()方法，返回表示方法执行时日期和时间的毫秒数。这个方法可以方便地用在代码分析中：

```
// 起始时间
let start = Date.now();

// 调用函数
doSomething();

// 结束时间
let stop = Date.now(),
  result = stop - start;
```

5.1.1 继承的方法

与其他类型一样，Date类型重写了toLocaleString()、toString()和valueOf()方法。但与其他类型不同，重写后这些方法的返回值不一样。Date类型的toLocaleString()方法返回与浏览器运行的本地环境一致的日期和时间。这通常意味着格式中包含针对时间的AM（上午）或PM（下午），但不包含时区信息（具体格式可能因浏览器而不同）。toString()方法通常返回带时区信息的日期和时间，而时间也是以24小时制（0~23）表示的。下面给出了toLocaleString()和toString()返回的2019年2月1日零点的示例（地区为"en-US"的PST，即Pacific Standard Time，太平洋标准时间）：

```
toLocaleString() - 2/1/2019 12:00:00 AM

toString() - Thu Feb 1 2019 00:00:00 GMT-0800 (Pacific Standard Time)
```

现代浏览器在这两个方法的输出上已经趋于一致。在比较老的浏览器上，每个方法返回的结果可能在每个浏览器上都是不同的。这些差异意味着toLocaleString()和toString()可能只对调试有用，不能用于显示。

Date类型的valueOf()方法根本就不返回字符串，这个方法被重写后返回的是日期的毫秒表示。因此，操作符（如小于号和大于号）可以直接使用它返回的值。比如下面的例子：

```
let date1 = new Date(2019, 0, 1);    // 2019年1月1日
let date2 = new Date(2019, 1, 1);    // 2019年2月1日

console.log(date1 < date2); // true
console.log(date1 > date2); // false
```

日期2019年1月1日在2019年2月1日之前，所以说前者小于后者没问题。因为2019年1月1日的毫秒表示小于2019年2月1日的毫秒表示，所以用小于号比较这两个日期时会返回true。这也是确保日期先后的一个简单方式。

5.1.2 日期格式化方法

Date类型有几个专门用于格式化日期的方法，它们都会返回字符串：

- toDateString()显示日期中的周几、月、日、年（格式特定于实现）；
- toTimeString()显示日期中的时、分、秒和时区（格式特定于实现）；
- toLocaleDateString()显示日期中的周几、月、日、年（格式特定于实现和地区）；
- toLocaleTimeString()显示日期中的时、分、秒（格式特定于实现）；
- toUTCString()显示完整的UTC日期（格式特定于实现）。

这些方法的输出与toLocaleString()和toString()一样，会因浏览器而异。因此不能用于在用户界面上一致地显示日期。

> **注意** 还有一个方法叫toGMTString()，这个方法跟toUTCString()是一样的，目的是为了向后兼容。不过，规范建议新代码使用toUTCString()。

5.1.3 日期/时间组件方法

Date类型剩下的方法（见下表）直接涉及取得或设置日期值的特定部分。注意表中"UTC日期"，指的是没有时区偏移（将日期转换为GMT）时的日期。

方法	说明
getTime()	返回日期的毫秒表示；与valueOf()相同
setTime(*milliseconds*)	设置日期的毫秒表示，从而修改整个日期
getFullYear()	返回4位数年（即2019而不是19）
getUTCFullYear()	返回UTC日期的4位数年
setFullYear(*year*)	设置日期的年（*year*必须是4位数）
setUTCFullYear(*year*)	设置UTC日期的年（*year*必须是4位数）
getMonth()	返回日期的月（0表示1月，11表示12月）
getUTCMonth()	返回UTC日期的月（0表示1月，11表示12月）
setMonth(*month*)	设置日期的月（*month*为大于0的数值，大于11加年）
setUTCMonth(*month*)	设置UTC日期的月（*month*为大于0的数值，大于11加年）
getDate()	返回日期中的日（1~31）
getUTCDate()	返回UTC日期中的日（1~31）
setDate(*date*)	设置日期中的日（如果*date*大于该月天数，则加月）
setUTCDate(*date*)	设置UTC日期中的日（如果*date*大于该月天数，则加月）
getDay()	返回日期中表示周几的数值（0表示周日，6表示周六）
getUTCDay()	返回UTC日期中表示周几的数值（0表示周日，6表示周六）
getHours()	返回日期中的时（0~23）
getUTCHours()	返回UTC日期中的时（0~23）
setHours(*hours*)	设置日期中的时（如果*hours*大于23，则加日）
setUTCHours(*hours*)	设置UTC日期中的时（如果*hours*大于23，则加日）
getMinutes()	返回日期中的分（0~59）
getUTCMinutes()	返回UTC日期中的分（0~59）
setMinutes(*minutes*)	设置日期中的分（如果*minutes*大于59，则加时）
setUTCMinutes(*minutes*)	设置UTC日期中的分（如果*minutes*大于59，则加时）
getSeconds()	返回日期中的秒（0~59）
getUTCSeconds()	返回UTC日期中的秒（0~59）
setSeconds(*seconds*)	设置日期中的秒（如果*seconds*大于59，则加分）
setUTCSeconds(*seconds*)	设置UTC日期中的秒（如果*seconds*大于59，则加分）

方法	说明
`getMilliseconds()`	返回日期中的毫秒
`getUTCMilliseconds()`	返回UTC日期中的毫秒
`setMilliseconds(*milliseconds*)`	设置日期中的毫秒
`setUTCMilliseconds(*milliseconds*)`	设置UTC日期中的毫秒
`getTimezoneOffset()`	返回以分钟计的UTC与本地时区的偏移量（如美国EST即“东部标准时间”返回300，进入夏令时的地区可能有所差异）

5.2 RegExp

ECMAScript通过`RegExp`类型支持正则表达式。正则表达式使用类似Perl的简洁语法来创建：

```
let expression = /pattern/flags;
```

这个正则表达式的`pattern`（模式）可以是任何简单或复杂的正则表达式，包括字符类、限定符、分组、向前查找和反向引用。每个正则表达式可以带零个或多个`flags`（标记），用于控制正则表达式的行为。下面给出了表示匹配模式的标记。

- `g`：全局模式，表示查找字符串的全部内容，而不是找到第一个匹配的内容就结束。
- `i`：不区分大小写，表示在查找匹配时忽略`pattern`和字符串的大小写。
- `m`：多行模式，表示查找到一行文本末尾时会继续查找。
- `y`：粘附模式，表示只查找从`lastIndex`开始及之后的字符串。
- `u`：Unicode模式，启用Unicode匹配。
- `s`：`dotAll`模式，表示元字符`.`匹配任何字符（包括`\n`或`\r`）。

使用不同模式和标记可以创建出各种正则表达式，比如：

```
// 匹配字符串中的所有"at"
let pattern1 = /at/g;

// 匹配第一个"bat"或"cat"，忽略大小写
let pattern2 = /[bc]at/i;

// 匹配所有以"at"结尾的三字符组合，忽略大小写
let pattern3 = /.at/gi;
```

与其他语言中的正则表达式类似，所有**元字符**在模式中也必须转义，包括：

```
( [ { \ ^ $ | ) ] } ? * + .
```

元字符在正则表达式中都有一种或多种特殊功能，所以要匹配上面这些字符本身，就必须使用反斜杠来转义。下面是几个例子：

```
// 匹配第一个"bat"或"cat"，忽略大小写
let pattern1 = /[bc]at/i;

// 匹配第一个"[bc]at"，忽略大小写
let pattern2 = /\[bc\]at/i;

// 匹配所有以"at"结尾的三字符组合，忽略大小写
let pattern3 = /.at/gi;

// 匹配所有".at"，忽略大小写
let pattern4 = /\.at/gi;
```

这里的pattern1匹配"bat"或"cat"，不区分大小写。要直接匹配"[bc]at"，左右中括号都必须像pattern2中那样使用反斜杠转义。在pattern3中，点号表示"at"前面的任意字符都可以匹配。如果想匹配".at"，那么要像pattern4中那样对点号进行转义。

前面例子中的正则表达式都是使用字面量形式定义的。正则表达式也可以使用RegExp构造函数来创建，它接收两个参数：模式字符串和（可选的）标记字符串。任何使用字面量定义的正则表达式也可以通过构造函数来创建，比如：

```
// 匹配第一个"bat"或"cat"，忽略大小写
let pattern1 = /[bc]at/i;

// 跟pattern1一样，只不过是用构造函数创建的
let pattern2 = new RegExp("[bc]at", "i");
```

这里的pattern1和pattern2是等效的正则表达式。注意，RegExp构造函数的两个参数都是字符串。因为RegExp的模式参数是字符串，所以在某些情况下需要二次转义。所有元字符都必须二次转义，包括转义字符序列，如\n（\转义后的字符串是\\，在正则表达式字符串中则要写成\\\\）。下表展示了几个正则表达式的字面量形式，以及使用RegExp构造函数创建时对应的模式字符串。

字面量模式	对应的字符串
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

此外，使用RegExp也可以基于已有的正则表达式实例，并可选择性地修改它们的标记：

```
const re1 = /cat/g;
console.log(re1);  // "/cat/g"

const re2 = new RegExp(re1);
console.log(re2);  // "/cat/g"

const re3 = new RegExp(re1, "i");
console.log(re3);  // "/cat/i"
```

5.2.1 RegExp实例属性

每个RegExp实例都有下列属性，提供有关模式的各方面信息。

- global：布尔值，表示是否设置了g标记。
- ignoreCase：布尔值，表示是否设置了i标记。
- unicode：布尔值，表示是否设置了u标记。
- sticky：布尔值，表示是否设置了y标记。
- lastIndex：整数，表示在源字符串中下一次搜索的开始位置，始终从0开始。
- multiline：布尔值，表示是否设置了m标记。
- dotAll：布尔值，表示是否设置了s标记。
- source：正则表达式的字面量字符串（不是传给构造函数的模式字符串），没有开头和结尾的斜杠。
- flags：正则表达式的标记字符串。始终以字面量而非传入构造函数的字符串模式形式返回（没有前后斜杠）。

通过这些属性可以全面了解正则表达式的信息，不过实际开发中用得并不多，因为模式声明中包含这些信息。下面是一个例子：

```
let pattern1 = /\[bc\]at/i;

console.log(pattern1.global);      // false
console.log(pattern1.ignoreCase);  // true
console.log(pattern1.multiline);   // false
console.log(pattern1.lastIndex);   // 0
console.log(pattern1.source);      // "\[bc\]at"
console.log(pattern1.flags);       // "i"

let pattern2 = new RegExp("\\[bc\\]at", "i");

console.log(pattern2.global);      // false
console.log(pattern2.ignoreCase);  // true
console.log(pattern2.multiline);   // false
console.log(pattern2.lastIndex);   // 0
console.log(pattern2.source);      // "\[bc\]at"
console.log(pattern2.flags);       // "i"
```

注意，虽然第一个模式是通过字面量创建的，第二个模式是通过RegExp构造函数创建的，但两个模式的source和flags属性是相同的。source和flags属性返回的是规范化之后可以在字面量中使用的形式。

5.2.2 RegExp实例方法

RegExp实例的主要方法是exec()，主要用于配合捕获组使用。这个方法只接收一个参数，即要应用模式的字符串。如果找到了匹配项，则返回包含第一个匹配信息的数组；如果没找到匹配项，则返回null。返回的数组虽然是Array的实例，但包含两个额外的属性：index和input。index是字符串中匹配模式的起始位置，input是要查找的字符串。这个数组的第一个元素是匹配整个模式的字符串，其他元素是与表达式中的捕获组匹配的字符串。如果模式中没有捕获组，则数组只包含一个元素。来看下面的例子：

```
let text = "mom and dad and baby";
let pattern = /mom( and dad( and baby)?)?/gi;

let matches = pattern.exec(text);
console.log(matches.index);   // 0
console.log(matches.input);   // "mom and dad and baby"
console.log(matches[0]);      // "mom and dad and baby"
console.log(matches[1]);      // " and dad and baby"
console.log(matches[2]);      // " and baby"
```

在这个例子中，模式包含两个捕获组：最内部的匹配项" and baby"，以及外部的匹配项" and dad"或" and dad and baby"。调用exec()后找到了一个匹配项。因为整个字符串匹配模式，所以matchs数组的index属性就是0。数组的第一个元素是匹配的整个字符串，第二个元素是匹配第一个捕获组的字符串，第三个元素是匹配第二个捕获组的字符串。

如果模式设置了全局标记，则每次调用exec()方法会返回一个匹配的信息。如果没有设置全局标记，则无论对同一个字符串调用多少次exec()，也只会返回第一个匹配的信息。

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

let matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches[0]);         // cat
console.log(pattern.lastIndex);  // 0

matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches[0]);         // cat
console.log(pattern.lastIndex);  // 0
```

上面例子中的模式没有设置全局标记，因此调用exec()只返回第一个匹配项（"cat"）。lastIndex在非全局模式下始终不变。

如果在这个模式上设置了g标记，则每次调用exec()都会在字符串中向前搜索下一个匹配项，如下面的例子所示：

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

```
let matches = pattern.exec(text);
console.log(matches.index);      // 0
console.log(matches[0]);         // cat
console.log(pattern.lastIndex);  // 3

matches = pattern.exec(text);
console.log(matches.index);      // 5
console.log(matches[0]);         // bat
console.log(pattern.lastIndex);  // 8

matches = pattern.exec(text);
console.log(matches.index);      // 10
console.log(matches[0]);         // sat
console.log(pattern.lastIndex);  // 13
```

这次模式设置了全局标记，因此每次调用exec()都会返回字符串中的下一个匹配项，直到搜索到字符串末尾。注意模式的lastIndex属性每次都会变化。在全局匹配模式下，每次调用exec()都会更新lastIndex值，以反映上次匹配的最后一个字符的索引。

如果模式设置了粘附标记y，则每次调用exec()就只会在lastIndex的位置上寻找匹配项。粘附标记覆盖全局标记。

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

let matches = pattern.exec(text);
console.log(matches.index);       // 0
console.log(matches[0]);          // cat
console.log(pattern.lastIndex);   // 3

// 以索引3对应的字符开头找不到匹配项，因此exec()返回null
// exec()没找到匹配项，于是将lastIndex设置为0
matches = pattern.exec(text);
console.log(matches);             // null
console.log(pattern.lastIndex);   // 0

// 向前设置lastIndex可以让粘附的模式通过exec()找到下一个匹配项：
pattern.lastIndex = 5;
matches = pattern.exec(text);
console.log(matches.index);       // 5
console.log(matches[0]);          // bat
console.log(pattern.lastIndex);   // 8
```

正则表达式的另一个方法是test()，接收一个字符串参数。如果输入的文本与模式匹配，则参数返回true，否则返回false。这个方法适用于只想测试模式是否匹配，而不需要实际匹配内容的情况。test()经常用在if语句中：

```
let text = "000-00-0000";
let pattern = /\d{3}-\d{2}-\d{4}/;
```

```
if (pattern.test(text)) {
  console.log("The pattern was matched.");
}
```

在这个例子中，正则表达式用于测试特定的数值序列。如果输入的文本与模式匹配，则显示匹配成功的消息。这个用法常用于验证用户输入，此时我们只在乎输入是否有效，不关心为什么无效。

无论正则表达式是怎么创建的，继承的方法`toLocaleString()`和`toString()`都返回正则表达式的字面量表示。比如：

```
let pattern = new RegExp("\\[bc\\]at", "gi");
console.log(pattern.toString());       // /\[bc\]at/gi
console.log(pattern.toLocaleString()); // /\[bc\]at/gi
```

这里的模式是通过`RegExp`构造函数创建的，但`toLocaleString()`和`toString()`返回的都是其字面量的形式。

> **注意** 正则表达式的`valueOf()`方法返回正则表达式本身。

5.2.3 RegExp构造函数属性

`RegExp`构造函数本身也有几个属性。（在其他语言中，这种属性被称为静态属性。）这些属性适用于作用域中的所有正则表达式，而且会根据最后执行的正则表达式操作而变化。这些属性还有一个特点，就是可以通过两种不同的方式访问它们。换句话说，每个属性都有一个全名和一个简写。下表列出了`RegExp`构造函数的属性。

全名	简写	说明
`input`	`$_`	最后搜索的字符串
`lastMatch`	`$&`	最后匹配的文本
`lastParen`	`$+`	最后匹配的捕获组
`leftContext`	`$`` | input字符串中出现在lastMatch`前面的文本	
`rightContext`	`$'`	`input`字符串中出现在`lastMatch`后面的文本

通过这些属性可以提取出与`exec()`和`test()`执行的操作相关的信息。来看下面的例子：

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

if (pattern.test(text)) {
  console.log(RegExp.input);        // this has been a short summer
  console.log(RegExp.leftContext);  // this has been a
  console.log(RegExp.rightContext); // summer
```

```
  console.log(RegExp.lastMatch);    // short
  console.log(RegExp.lastParen);    // s
}
```

以上代码创建了一个模式，用于搜索任何后跟"hort"的字符，并把第一个字符放在了捕获组中。不同属性包含的内容如下。

- input属性中包含原始的字符串。
- leftConext属性包含原始字符串中"short"之前的内容，rightContext属性包含"short"之后的内容。
- lastMatch属性包含匹配整个正则表达式的上一个字符串，即"short"。
- lastParen属性包含捕获组的上一次匹配，即"s"。

这些属性名也可以替换成简写形式，只不过要使用中括号语法来访问，如下面的例子所示，因为大多数简写形式都不是合法的ECMAScript标识符：

```
let text = "this has been a short summer";
let pattern = /(.)hort/g;

/*
 * 注意：Opera不支持简写属性名
 * IE不支持多行匹配
 */
if (pattern.test(text)) {
  console.log(RegExp.$_);       // this has been a short summer
  console.log(RegExp["$`"]);    // this has been a
  console.log(RegExp["$'"]);    // summer
  console.log(RegExp["$&"]);    // short
  console.log(RegExp["$+"]);    // s
  console.log(RegExp["$*"]);    // false
}
```

RegExp还有其他几个构造函数属性，可以存储最多9个捕获组的匹配项。这些属性通过RegExp.$1~RegExp.$9来访问，分别包含第1~9个捕获组的匹配项。在调用exec()或test()时，这些属性就会被填充，然后就可以像下面这样使用它们：

```
let text = "this has been a short summer";
let pattern = /(..)or(.)/g;

if (pattern.test(text)) {
  console.log(RegExp.$1);  // sh
  console.log(RegExp.$2);  // t
}
```

在这个例子中，模式包含两个捕获组。调用test()搜索字符串之后，因为找到了匹配项所以返回true，而且可以打印出通过RegExp构造函数的$1和$2属性取得的两个捕获组匹配的内容。

> **注意** RegExp构造函数的所有属性都没有任何Web标准出处，因此不要在生产环境中使用它们。

5.2.4 模式局限

虽然ECMAScript对正则表达式的支持有了长足的进步，但仍然缺少Perl语言中的一些高级特性。下列特性目前还没有得到ECMAScript的支持（想要了解更多信息，可以参考Regular-Expressions.info网站）：

- \A和\Z锚（分别匹配字符串的开始和末尾）
- 联合及交叉类
- 原子组
- x（忽略空格）匹配模式
- 条件式匹配
- 正则表达式注释

虽然还有这些局限，但ECMAScript的正则表达式已经非常强大，可以用于大多数模式匹配任务。

5.3 原始值包装类型

为了方便操作原始值，ECMAScript提供了3种特殊的引用类型：Boolean、Number和String。这些类型具有本章介绍的其他引用类型一样的特点，但也具有与各自原始类型对应的特殊行为。每当用到某个原始值的方法或属性时，后台都会创建一个相应原始包装类型的对象，从而暴露出操作原始值的各种方法。来看下面的例子：

```
let s1 = "some text";
let s2 = s1.substring(2);
```

在这里，s1是一个包含字符串的变量，它是一个原始值。第二行紧接着在s1上调用了substring()方法，并把结果保存在s2中。我们知道，原始值本身不是对象，因此逻辑上不应该有方法。而实际上这个例子又确实按照预期运行了。这是因为后台进行了很多处理，从而实现了上述操作。具体来说，当第二行访问s1时，是以读模式访问的，也就是要从内存中读取变量保存的值。在以读模式访问字符串值的任何时候，后台都会执行以下3步：

(1) 创建一个String类型的实例；

(2) 调用实例上的特定方法；

(3) 销毁实例。

可以把这3步想象成执行了如下3行ECMAScript代码：

```
let s1 = new String("some text");
let s2 = s1.substring(2);
s1 = null;
```

这种行为可以让原始值拥有对象的行为。对布尔值和数值而言，以上3步也会在后台发生，只不过使用的是Boolean和Number包装类型而已。

引用类型与原始值包装类型的主要区别在于对象的生命周期。在通过new实例化引用类型后，得到的实例会在离开作用域时被销毁，而自动创建的原始值包装对象则只存在于访问它的那行代码执行期间。这意味着不能在

运行时给原始值添加属性和方法。比如下面的例子：

```
let s1 = "some text";
s1.color = "red";
console.log(s1.color);  // undefined
```

这里的第二行代码尝试给字符串s1添加了一个color属性。可是，第三行代码访问color属性时，它却不见了。原因就是第二行代码运行时会临时创建一个String对象，而当第三行代码执行时，这个对象已经被销毁了。实际上，第三行代码在这里创建了自己的String对象，但这个对象没有color属性。

可以显式地使用Boolean、Number和String构造函数创建原始值包装对象。不过应该在确实必要时再这么做，否则容易让开发者疑惑，分不清它们到底是原始值还是引用值。在原始值包装类型的实例上调用typeof会返回"object"，所有原始值包装对象都会转换为布尔值true。

另外，Object构造函数作为一个工厂方法，能够根据传入值的类型返回相应原始值包装类型的实例。比如：

```
let obj = new Object("some text");
console.log(obj instanceof String);  // true
```

如果传给Object的是字符串，则会创建一个String的实例。如果是数值，则会创建Number的实例。布尔值则会得到Boolean的实例。

注意，使用new调用原始值包装类型的构造函数，与调用同名的转型函数并不一样。例如：

```
let value = "25";
let number = Number(value);     // 转型函数
console.log(typeof number);     // "number"
let obj = new Number(value);    // 构造函数
console.log(typeof obj);        // "object"
```

在这个例子中，变量number中保存的是一个值为25的原始数值，而变量obj中保存的是一个Number的实例。

虽然不推荐显式创建原始值包装类型的实例，但它们对于操作原始值的功能是很重要的。每个原始值包装类型都有相应的一套方法来方便数据操作。

5.3.1 Boolean

Boolean是对应布尔值的引用类型。要创建一个Boolean对象，就使用Boolean构造函数并传入true或false，如下例所示：

```
let booleanObject = new Boolean(true);
```

Boolean的实例会重写valueOf()方法，返回一个原始值true或false。toString()方法被调用时也会被覆盖，返回字符串"true"或"false"。不过，Boolean对象在ECMAScript中用得很少。不仅如此，它们还容易引

起误会，尤其是在布尔表达式中使用Boolean对象时，比如：

```
let falseObject = new Boolean(false);
let result = falseObject && true;
console.log(result); // true

let falseValue = false;
result = falseValue && true;
console.log(result); // false
```

在这段代码中，我们创建一个值为false的Boolean对象。然后，在一个布尔表达式中通过&&操作将这个对象与一个原始值true组合起来。在布尔算术中，false && true等于false。可是，这个表达式是对falseObject对象而不是对它表示的值（false）求值。前面刚刚说过，所有对象在布尔表达式中都会自动转换为true，因此falseObject在这个表达式里实际上表示一个true值。那么true && true当然是true。

除此之外，原始值和引用值（Boolean对象）还有几个区别。首先，typeof操作符对原始值返回"boolean"，但对引用值返回"object"。同样，Boolean对象是Boolean类型的实例，在使用instaceof操作符时返回true，但对原始值则返回false，如下所示：

```
console.log(typeof falseObject);              // object
console.log(typeof falseValue);               // boolean
console.log(falseObject instanceof Boolean); // true
console.log(falseValue instanceof Boolean);  // false
```

理解原始布尔值和Boolean对象之间的区别非常重要，强烈建议永远不要使用后者。

5.3.2 Number

Number是对应数值的引用类型。要创建一个Number对象，就使用Number构造函数并传入一个数值，如下例所示：

```
let numberObject = new Number(10);
```

与Boolean类型一样，Number类型重写了valueOf()、toLocaleString()和toString()方法。valueOf()方法返回Number对象表示的原始数值，另外两个方法返回数值字符串。toString()方法可选地接收一个表示基数的参数，并返回相应基数形式的数值字符串，如下所示：

```
let num = 10;
console.log(num.toString());   // "10"
console.log(num.toString(2));  // "1010"
console.log(num.toString(8));  // "12"
console.log(num.toString(10)); // "10"
console.log(num.toString(16)); // "a"
```

除了继承的方法，Number类型还提供了几个用于将数值格式化为字符串的方法。

toFixed()方法返回包含指定小数点位数的数值字符串，如：

```
let num = 10;
console.log(num.toFixed(2)); // "10.00"
```

这里的toFixed()方法接收了参数2，表示返回的数值字符串要包含两位小数。结果返回值为"10.00"，小数位填充了0。如果数值本身的小数位超过了参数指定的位数，则四舍五入到最接近的小数位：

```
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()、toLcaleString()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，`trimeLeft()`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 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"被修改为"cond"，而字符串的剩余部分保持不变。通过将第一个参数改为带全局标记的正则表达式，字符串中的所有"at"都被替换成了"ond"。

第二个参数是字符串的情况下，有几个特殊的字符序列，可以用来插入正则表达式操作的值。ECMA-262中规定了下表中的值。

字符序列	替换文本
$$	$
$&	匹配整个模式的子字符串。与RegExp.lastMatch相同
$'	匹配的子字符串之前的字符串。与RegExp.rightContext相同
$`` | 匹配的子字符串之后的字符串。与 RegExp.leftContext`相同	
$*n*	匹配第*n*个捕获组的字符串，其中*n*是0~9。比如，$1是匹配第一个捕获组的字符串，$2是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串

字符序列	替换文本
$*nn*	匹配第*nn*个捕获组字符串，其中*nn*是01~99。比如，$01是匹配第一个捕获组的字符串，$02是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串

使用这些特殊的序列，可以在替换文本中使用之前匹配的内容，如下面的例子所示：

```
let text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
console.log(result);  // word (cat), word (bat), word (sat), word (fat)
```

这里，每个以"at"结尾的词都会被替换成"word"后跟一对小括号，其中包含捕获组匹配的内容$1。

replace()的第二个参数可以是一个函数。在只有一个匹配项时，这个函数会收到3个参数：与整个模式匹配的字符串、匹配项在字符串中的开始位置，以及整个字符串。在有多个捕获组的情况下，每个匹配捕获组的字符串也会作为参数传给这个函数，但最后两个参数还是与整个模式匹配的开始位置和原始字符串。这个函数应该返回一个字符串，表示应该把匹配项替换成什么。使用函数作为第二个参数可以更细致地控制替换过程，如下所示：

```
function htmlEscape(text) {
  return text.replace(/[<>"&]/g, function(match, pos, originalText) {
    switch(match) {
      case "<":
        return "&lt;";
      case ">":
        return "&gt;";
      case "&":
        return "&amp;";
      case "\"":
        return "&quot;";
    }
  });
}

console.log(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
// "&lt;p class=&quot;greeting&quot;&gt;Hello world!</p>"
```

这里，函数htmlEscape()用于将一段HTML中的4个字符替换成对应的实体：小于号、大于号、和号，还有双引号（都必须经过转义）。实现这个任务最简单的办法就是用一个正则表达式查找这些字符，然后定义一个函数，根据匹配的每个字符分别返回特定的HTML实体。

最后一个与模式匹配相关的字符串方法是split()。这个方法会根据传入的分隔符将字符串拆分成数组。作为分隔符的参数可以是字符串，也可以是RegExp对象。（字符串分隔符不会被这个方法当成正则表达式。）还可以传入第二个参数，即数组大小，确保返回的数组不会超过指定大小。来看下面的例子：

```
let colorText = "red,blue,green,yellow";
let colors1 = colorText.split(",");       // ["red", "blue", "green",
"yellow"]
let colors2 = colorText.split(",", 2);    // ["red", "blue"]
let colors3 = colorText.split(/[^,]+/);   // ["", ",", ",", ",", ""]
```

在这里，字符串colorText是一个逗号分隔的颜色名称符串。调用split(",")会得到包含这些颜色名的数组，基于逗号进行拆分。要把数组元素限制为2个，传入第二个参数2即可。最后，使用正则表达式可以得到一个包含逗号的数组。注意在最后一次调用split()时，返回的数组前后包含两个空字符串。这是因为正则表达式指定的分隔符出现在了字符串开头（"red"）和末尾（"yellow"）。

12. **localeCompare()方法**

最后一个方法是localeCompare()，这个方法比较两个字符串，返回如下3个值中的一个。

- 如果按照字母表顺序，字符串应该排在字符串参数前头，则返回负值。（通常是-1，具体还要看与实际值相关的实现。）
- 如果字符串与字符串参数相等，则返回0。
- 如果按照字母表顺序，字符串应该排在字符串参数后头，则返回正值。（通常是1，具体还要看与实际值相关的实现。）

下面是一个例子：

```
let stringValue = "yellow";
console.log(stringValue.localeCompare("brick"));  // 1
console.log(stringValue.localeCompare("yellow")); // 0
console.log(stringValue.localeCompare("zoo"));    // -1
```

在这里，字符串"yellow"与3个不同的值进行了比较："brick"、"yellow"和"zoo"。"brick"按字母表顺序应该排在"yellow"前头，因此localeCompare()返回1。"yellow"等于"yellow"，因此"localeCompare()"返回0。最后，"zoo"在"yellow"后面，因此localeCompare()返回-1。强调一下，因为返回的具体值可能因具体实现而异，所以最好像下面的示例中一样使用localeCompare()：

```
function determineOrder(value) {
  let result = stringValue.localeCompare(value);
  if (result < 0) {
    console.log(`The string 'yellow' comes before the string '${value}'.`);
  } else if (result > 0) {
    console.log(`The string 'yellow' comes after the string '${value}'.`);
  } else {
    console.log(`The string 'yellow' is equal to the string '${value}'.`);
  }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

这样一来，就可以保证在所有实现中都能正确判断字符串的顺序了。

`localeCompare()`的独特之处在于，实现所在的地区（国家和语言）决定了这个方法如何比较字符串。在美国，英语是ECMAScript实现的标准语言，`localeCompare()`区分大小写，大写字母排在小写字母前面。但其他地区未必是这种情况。

13. **HTML方法**

早期的浏览器开发商认为使用JavaScript动态生成HTML标签是一个需求。因此，早期浏览器扩展了规范，增加了辅助生成HTML标签的方法。下表总结了这些HTML方法。不过，这些方法基本上已经没有人使用了，因为结果通常不是语义化的标记。

方法	输出
`anchor(*name*)`	`<a name="*name*">*string*</a>`
`big()`	`<big>*string*</big>`
`bold()`	`<b>*string*</b>`
`fixed()`	`<tt>*string*</tt>`
`fontcolor(*color*)`	`<font color="*color*">*string*</font>`
`fontsize(*size*)`	`<font size="*size*">*string*</font>`
`italics()`	`<i>*string*</i>`
`link(url)`	`<a href="*url*">*string*</a>`
`small()`	`<small>*string*</small>`
`strike()`	`<strike>*string*</strike>`
`sub()`	`<sub>*string*</sub>`
`sup()`	`<sup>*string*</sup>`

5.4 单例内置对象

ECMA-262对内置对象的定义是“任何由ECMAScript实现提供、与宿主环境无关，并在ECMAScript程序开始执行时就存在的对象”。这就意味着，开发者不用显式地实例化内置对象，因为它们已经实例化好了。前面我们已经接触了大部分内置对象，包括`Object`、`Array`和`String`。本节介绍ECMA-262定义的另外两个单例内置对象：`Global`和`Math`。

5.4.1 `Global`

`Global`对象是ECMAScript中最特别的对象，因为代码不会显式地访问它。ECMA-262规定`Global`对象为一种兜底对象，它所针对的是不属于任何对象的属性和方法。事实上，不存在全局变量或全局函数这种东西。在全局作用域中定义的变量和函数都会变成`Global`对象的属性 。本书前面介绍的函数，包括`isNaN()`、`isFinite()`、`parseInt()`和`parseFloat()`，实际上都是`Global`对象的方法。除了这些，`Global`对象上还有另外一些方法。

1. **URL编码方法**

encodeURI()和encodeURIComponent()方法用于编码统一资源标识符（URI），以便传给浏览器。有效的URI不能包含某些字符，比如空格。使用URI编码方法来编码URI可以让浏览器能够理解它们，同时又以特殊的UTF-8编码替换掉所有无效字符。

ecnodeURI()方法用于对整个URI进行编码，比如"www.wrox.com/illegal value.js"。而encodeURIComponent()方法用于编码URI中单独的组件，比如前面URL中的"illegal value.js"。这两个方法的主要区别是，encodeURI()不会编码属于URL组件的特殊字符，比如冒号、斜杠、问号、井号，而encodeURIComponent()会编码它发现的所有非标准字符。来看下面的例子：

```
let uri = "http:// www.wrox.com/illegal value.js#start";

// "http:// www.wrox.com/illegal%20value.js#start"
console.log(encodeURI(uri));

// "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start"
console.log(encodeURIComponent(uri));
```

这里使用encodeURI()编码后，除空格被替换为%20之外，没有任何变化。而ecnodeURIComponent()方法将所有非字母字符都替换成了相应的编码形式。这就是使用encodeURI()编码整个URI，但只使用encodeURIComponent()编码那些会追加到已有URI后面的字符串的原因。

> **注意** 一般来说，使用encodeURIComponent()应该比使用encodeURI()的频率更高，这是因为编码查询字符串参数比编码基准URI的次数更多。

与encodeURI()和encodeURIComponent()相对的是decodeURI()和decodeURIComponent()。decodeURI()只对使用encodeURI()编码过的字符解码。例如，%20会被替换为空格，但%23不会被替换为井号（#），因为井号不是由encodeURI()替换的。类似地，decodeURIComponent()解码所有被encodeURIComponent()编码的字符，基本上就是解码所有特殊值。来看下面的例子：

```
let uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start";

// http%3A%2F%2Fwww.wrox.com%2Fillegal value.js%23start
console.log(decodeURI(uri));

// http:// www.wrox.com/illegal value.js#start
console.log(decodeURIComponent(uri));
```

这里，uri变量中包含一个使用encodeURIComponent()编码过的字符串。首先输出的是使用decodeURI()解码的结果，可以看到只用空格替换了%20。然后是使用decodeURIComponent()解码的结果，其中替换了所有特殊字符，并输出了没有包含任何转义的字符串。（这个字符串不是有效的URL。）

> **注意** URI方法encodeURI()、encodeURIComponent()、decodeURI()和decodeURIComponent()取代了escape()和unescape()方法，后者在ECMA-262第3版中就已经废弃了。URI方法始终是首选方法，因为它们对所有Unicode字符进行编码，而原来的方法只能正确编码ASCII字符。不要在生产环境中使用escape()和unescape()。

2. **`eval()`方法**

最后一个方法可能是整个ECMAScript语言中最强大的了，它就是`eval()`。这个方法就是一个完整的ECMAScript解释器，它接收一个参数，即一个要执行的ECMAScript（JavaScript）字符串。来看一个例子：

```
eval("console.log('hi')");
```

上面这行代码的功能与下一行等价：

```
console.log("hi");
```

当解释器发现`eval()`调用时，会将参数解释为实际的ECMAScript语句，然后将其插入到该位置。通过`eval()`执行的代码属于该调用所在上下文，被执行的代码与该上下文拥有相同的作用域链。这意味着定义在包含上下文中的变量可以在`eval()`调用内部被引用，比如下面这个例子：

```
let msg = "hello world";
eval("console.log(msg)");  // "hello world"
```

这里，变量`msg`是在`eval()`调用的外部上下文中定义的，而`console.log()`显示了文本`"hello world"`。这是因为第二行代码会被替换成一行真正的函数调用代码。类似地，可以在`eval()`内部定义一个函数或变量，然后在外部代码中引用，如下所示：

```
eval("function sayHi() { console.log('hi'); }");
sayHi();
```

这里，函数`sayHi()`是在`eval()`内部定义的。因为该调用会被替换为真正的函数定义，所以才可能在下一行代码中调用`sayHi()`。对于变量也是一样的：

```
eval("let msg = 'hello world';");
console.log(msg);  // Reference Error: msg is not defined
```

通过`eval()`定义的任何变量和函数都不会被提升，这是因为在解析代码的时候，它们是被包含在一个字符串中的。它们只是在`eval()`执行的时候才会被创建。

在严格模式下，在`eval()`内部创建的变量和函数无法被外部访问。换句话说，最后两个例子会报错。同样，在严格模式下，赋值给`eval`也会导致错误：

```
"use strict";
eval = "hi";  // 导致错误
```

> **注意** 解释代码字符串的能力是非常强大的，但也非常危险。在使用`eval()`的时候必须极为慎重，特别是在解释用户输入的内容时。因为这个方法会对XSS利用暴露出很大的攻击面。恶意用户可能插入会导致你网站或应用崩溃的代码。

3. **`Global`对象属性**

`Global`对象有很多属性，其中一些前面已经提到过了。像`undefined`、`NaN`和`Infinity`等特殊值都是`Global`对象的属性。此外，所有原生引用类型构造函数，比如`Object`和`Function`，也都是`Global`对象的属性。下表列出了所有这些属性。

属性	说明
`undefined`	特殊值`undefined`
`NaN`	特殊值`NaN`
`Infinity`	特殊值`Infinity`
`Object`	`Object`的构造函数
`Array`	`Array`的构造函数
`Function`	`Function`的构造函数
`Boolean`	`Boolean`的构造函数
`String`	`String`的构造函数
`Number`	`Number`的构造函数
`Date`	`Date`的构造函数
`RegExp`	`RegExp`的构造函数
`Symbol`	`Symbol`的伪构造函数
`Error`	`Error`的构造函数
`EvalError`	`EvalError`的构造函数
`RangeError`	`RangeError`的构造函数
`ReferenceError`	`ReferenceError`的构造函数
`SyntaxError`	`SyntaxError`的构造函数
`TypeError`	`TypeError`的构造函数
`URIError`	`URIError`的构造函数

4. **`window`对象**

虽然ECMA-262没有规定直接访问`Global`对象的方式，但浏览器将`window`对象实现为`Global`对象的代理。因此，所有全局作用域中声明的变量和函数都变成了`window`的属性。来看下面的例子：

```
var color = "red";

function sayColor() {
  console.log(window.color);
}

window.sayColor(); // "red"
```

这里定义了一个名为color的全局变量和一个名为sayColor()的全局函数。在sayColor()内部，通过window.color访问了color变量，说明全局变量变成了window的属性。接着，又通过window对象直接调用了window.sayColor()函数，从而输出字符串。

> **注意** window对象在JavaScript中远不止实现了ECMAScript的Global对象那么简单。关于window对象的更多介绍，请参考第12章。

另一种获取Global对象的方式是使用如下的代码：

```
let global = function() {
  return this;
}();
```

这段代码创建一个立即调用的函数表达式，返回了this的值。如前所述，当一个函数在没有明确（通过成为某个对象的方法，或者通过call()/apply()）指定this值的情况下执行时，this值等于Global对象。因此，调用一个简单返回this的函数是在任何执行上下文中获取Global对象的通用方式。

5.4.2 Math

ECMAScript提供了Math对象作为保存数学公式、信息和计算的地方。Math对象提供了一些辅助计算的属性和方法。

> **注意** Math对象上提供的计算要比直接在JavaScript实现的快得多，因为Math对象上的计算使用了JavaScript引擎中更高效的实现和处理器指令。但使用Math计算的问题是精度会因浏览器、操作系统、指令集和硬件而异。

1. **Math对象属性**

   Math对象有一些属性，主要用于保存数学中的一些特殊值。下表列出了这些属性。

属性	说明
Math.E	自然对数的基数e的值
Math.LN10	10为底的自然对数
Math.LN2	2为底的自然对数
Math.LOG2E	以2为底e的对数
Math.LOG10E	以10为底e的对数

属性	说明
Math.PI	π的值
Math.SQRT1_2	1/2的平方根
Math.SQRT2	2的平方根

这些值的含义和用法超出了本书的范畴，但都是ECMAScript规范定义的，并可以在你需要时使用。

2. **min()和max()方法**

Math对象也提供了很多辅助执行简单或复杂数学计算的方法。

min()和max()方法用于确定一组数值中的最小值和最大值。这两个方法都接收任意多个参数，如下面的例子所示：

```
let max = Math.max(3, 54, 32, 16);
console.log(max);  // 54

let min = Math.min(3, 54, 32, 16);
console.log(min);  // 3
```

在3、54、32和16中，Math.max()返回54，Math.min()返回3。使用这两个方法可以避免使用额外的循环和if语句来确定一组数值的最大最小值。

要知道数组中的最大值和最小值，可以像下面这样使用扩展操作符：

```
let values = [1, 2, 3, 4, 5, 6, 7, 8];
let max = Math.max(...values);
```

3. **舍入方法**

接下来是用于把小数值舍入为整数的4个方法：Math.ceil()、Math.floor()、Math.round()和Math.fround()。这几个方法处理舍入的方式如下所述。

- Math.ceil()方法始终向上舍入为最接近的整数。
- Math.floor()方法始终向下舍入为最接近的整数。
- Math.round()方法执行四舍五入。
- Math.fround()方法返回数值最接近的单精度（32位）浮点值表示。

以下示例展示了这些方法的用法：

```
console.log(Math.ceil(25.9));   // 26
console.log(Math.ceil(25.5));   // 26
console.log(Math.ceil(25.1));   // 26

console.log(Math.round(25.9));  // 26
```

```
console.log(Math.round(25.5));  // 26
console.log(Math.round(25.1));  // 25

console.log(Math.fround(0.4));  // 0.4000000059604645
console.log(Math.fround(0.5));  // 0.5
console.log(Math.fround(25.9)); // 25.899999618530273

console.log(Math.floor(25.9));  // 25
console.log(Math.floor(25.5));  // 25
console.log(Math.floor(25.1));  // 25
```

对于25和26（不包含）之间的所有值，`Math.ceil()`都会返回26，因为它始终向上舍入。`Math.round()`只在数值大于等于`25.5`时返回26，否则返回25。最后，`Math.floor()`对所有25和26（不包含）之间的值都返回25。

4. **`random()`方法**

`Math.random()`方法返回一个0~1范围内的随机数，其中包含0但不包含1。对于希望显示随机名言或随机新闻的网页，这个方法是非常方便的。可以基于如下公式使用`Math.random()`从一组整数中随机选择一个数：

```
number = Math.floor(Math.random() * total_number_of_choices +
first_possible_value)
```

这里使用了`Math.floor()`方法，因为`Math.random()`始终返回小数，即便乘以一个数再加上一个数也是小数。因此，如果想从1~10范围内随机选择一个数，代码就是这样的：

```
let num = Math.floor(Math.random() * 10 + 1);
```

这样就有10个可能的值（1~10），其中最小的值是1。如果想选择一个2~10范围内的值，则代码就要写成这样：

```
let num = Math.floor(Math.random() * 9 + 2);
```

2~10只有9个数，所以可选总数（`total_number_of_choices`）是9，而最小可能的值（`first_possible_value`）是2。很多时候，通过函数来算出可选总数和最小可能的值可能更方便，比如：

```
function selectFrom(lowerValue, upperValue) {
  let choices = upperValue - lowerValue + 1;
  return Math.floor(Math.random() * choices + lowerValue);
}
```

```
let num = selectFrom(2,10);
console.log(num);  // 2~10范围内的值，其中包含2和10
```

这里的函数selectFrom()接收两个参数：应该返回的最小值和最大值。通过将这两个值相减再加1得到可选总数，然后再套用上面的公式。于是，调用selectFrom(2,10)就可以从2~10（包含）范围内选择一个值了。使用这个函数，从一个数组中随机选择一个元素就很容易，比如：

```
let colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
let color = colors[selectFrom(0, colors.length-1)];
```

在这个例子中，传给selecFrom()的第二个参数是数组长度减1，即数组最大的索引值。

> **注意** Math.random()方法在这里出于演示目的是没有问题的。如果是为了加密而需要生成随机数（传给生成器的输入需要较高的不确定性），那么建议使用window.crypto.getRandomValues()。

5. **其他方法**

Math对象还有很多涉及各种简单或高阶数运算的方法。讨论每种方法的具体细节或者它们的适用场景超出了本书的范畴。不过，下表还是总结了Math对象的其他方法。

方法	说明
Math.abs(*x*)	返回*x*的绝对值
Math.e*x*p(*x*)	返回Math.E的*x*次幂
Math.e*x*pm1(*x*)	等于Math.e*x*p(*x*) - 1
Math.log(*x*)	返回*x*的自然对数
Math.log1p(*x*)	等于1 + Math.log(*x*)
Math.pow(*x*, *power*)	返回*x*的*power*次幂
Math.pow(*...nums*)	返回*nums*中每个数平方和的平方根
Math.clz32(*x*)	返回32位整数*x*的前置零的数量
Math.sign(*x*)	返回表示*x*符号的1、0、-0或-1
Math.trunc(*x*)	返回*x*的整数部分，删除所有小数
Math.sqrt(*x*)	返回*x*的平方根
Math.cbrt(*x*)	返回*x*的立方根
Math.acos(*x*)	返回*x*的反余弦
Math.acosh(*x*)	返回*x*的反双曲余弦
Math.asin(*x*)	返回*x*的反正弦
Math.asinh(*x*)	返回*x*的反双曲正弦

方法	说明
`Math.atan(*x*)`	返回*x*的反正切
`Math.atanh(*x*)`	返回*x*的反双曲正切
`Math.atan2(*y*, *x*)`	返回*y*/*x*的反正切
`Math.cos(*x*)`	返回*x*的余弦
`Math.sin(*x*)`	返回*x*的正弦
`Math.tan(*x*)`	返回*x*的正切

即便这些方法都是由ECMA-262定义的，对正弦、余弦、正切等计算的实现仍然取决于浏览器，因为计算这些值的方式有很多种。结果，这些方法的精度可能因实现而异。

5.5 小结

JavaScript中的对象称为引用值，几种内置的引用类型可用于创建特定类型的对象。

- 引用值与传统面向对象编程语言中的类相似，但实现不同。
- `Date`类型提供关于日期和时间的信息，包括当前日期、时间及相关计算。
- `RegExp`类型是ECMAScript支持正则表达式的接口，提供了大多数基础的和部分高级的正则表达式功能。

JavaScript比较独特的一点是，函数实际上是`Function`类型的实例，也就是说函数也是对象。因为函数也是对象，所以函数也有方法，可以用于增强其能力。

由于原始值包装类型的存在，JavaScript中的原始值可以被当成对象来使用。有3种原始值包装类型：`Boolean`、`Number`和`String`。它们都具备如下特点。

- 每种包装类型都映射到同名的原始类型。
- 以读模式访问原始值时，后台会实例化一个原始值包装类型的对象，借助这个对象可以操作相应的数据。
- 涉及原始值的语句执行完毕后，包装对象就会被销毁。

当代码开始执行时，全局上下文中会存在两个内置对象：`Global`和`Math`。其中，`Global`对象在大多数ECMAScript实现中无法直接访问。不过，浏览器将其实现为`window`对象。所有全局变量和函数都是`Global`对象的属性。`Math`对象包含辅助完成复杂计算的属性和方法。

第 6 章　集合引用类型

> **本章内容**
>
> - 对象
> - 数组与定型数组
> - `Map`、`WeakMap`、`Set`以及`WeakSet`类型

6.1 Object

到目前为止，大多数引用值的示例使用的是`Object`类型。`Object`是ECMAScript中最常用的类型之一。虽然`Object`的实例没有多少功能，但很适合存储和在应用程序间交换数据。

显式地创建Object的实例有两种方式。第一种是使用new操作符和Object构造函数，如下所示：

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
```

另一种方式是使用**对象字面量**（object literal）表示法。对象字面量是对象定义的简写形式，目的是为了简化包含大量属性的对象的创建。比如，下面的代码定义了与前面示例相同的person对象，但使用的是对象字面量表示法：

```
let person = {
  name: "Nicholas",
  age: 29
};
```

在这个例子中，左大括号（{）表示对象字面量开始，因为它出现在一个**表达式上下文**（expression context）中。在ECMAScript中，表达式上下文指的是期待返回值的上下文。赋值操作符表示后面要期待一个值，因此左大括号表示一个表达式的开始。同样是左大括号，如果出现在**语句上下文**（statement context）中，比如if语句的条件后面，则表示一个语句块的开始。

接下来指定了name属性，后跟一个冒号，然后是属性的值。逗号用于在对象字面量中分隔属性，因此字符串"Nicholas"后面有一个逗号，而29后面没有，因为age是这个对象的最后一个属性。在最后一个属性后面加上逗号在非常老的浏览器中会导致报错，但所有现代浏览器都支持这种写法。

在对象字面量表示法中，属性名可以是字符串或数值，比如：

```
let person = {
  "name": "Nicholas",
  "age": 29,
  5: true
};
```

这个例子会得到一个带有属性name、age和5的对象。注意，数值属性会自动转换为字符串。

当然也可以用对象字面量表示法来定义一个只有默认属性和方法的对象，只要使用一对大括号，中间留空就行了：

```
let person = {}; // 与new Object()相同
person.name = "Nicholas";
person.age = 29;
```

这个例子跟本节开始的第一个例子是等效的，虽然看起来有点怪。对象字面量表示法通常只在为了让属性一目了然时才使用。

> **注意** 在使用对象字面量表示法定义对象时，并不会实际调用`Object`构造函数。

虽然使用哪种方式创建`Object`实例都可以，但实际上开发者更倾向于使用对象字面量表示法。这是因为对象字面量代码更少，看起来也更有封装所有相关数据的感觉。事实上，对象字面量已经成为给函数传递大量可选参数的主要方式，比如：

```
function displayInfo(args) {
  let output = "";

  if (typeof args.name == "string"){
    output += "Name: " + args.name + "\n";
  }

  if (typeof args.age == "number") {
    output += "Age: " + args.age + "\n";
  }

  alert(output);
}

displayInfo({
  name: "Nicholas",
  age: 29
});

displayInfo({
  name: "Greg"
});
```

这里，函数`displayInfo()`接收一个名为`args`的参数。这个参数可能有属性`name`或`age`，也可能两个属性都有或者都没有。函数内部会使用`typeof`操作符测试每个属性是否存在，然后根据属性有无构造并显示一条消息。然后，这个函数被调用了两次，每次都通过一个对象字面量传入了不同的数据。两种情况下，函数都正常运行。

> **注意** 这种模式非常适合函数有大量可选参数的情况。一般来说，命名参数更直观，但在可选参数过多的时候就显得笨拙了。最好的方式是对必选参数使用命名参数，再通过一个对象字面量来封装多个可选参数。

虽然属性一般是通过**点语法**来存取的，这也是面向对象语言的惯例，但也可以使用中括号来存取属性。在使用中括号时，要在括号内使用属性名的字符串形式，比如：

```
console.log(person["name"]); // "Nicholas"
console.log(person.name);    // "Nicholas"
```

从功能上讲，这两种存取属性的方式没有区别。使用中括号的主要优势就是可以通过变量访问属性，就像下面这个例子中一样：

```
let propertyName = "name";
console.log(person[propertyName]); // "Nicholas"
```

另外，如果属性名中包含可能会导致语法错误的字符，或者包含关键字/保留字时，也可以使用中括号语法。比如：

```
person["first name"] = "Nicholas";
```

因为"first name"中包含一个空格，所以不能使用点语法来访问。不过，属性名中是可以包含非字母数字字符的，这时候只要用中括号语法存取它们就行了。

通常，点语法是首选的属性存取方式，除非访问属性时必须使用变量。

> **注意** 第8章将更全面、深入地介绍Object类型。

6.2 Array

除了Object，Array应该就是ECMAScript中最常用的类型了。ECMAScript数组跟其他编程语言的数组有很大区别。跟其他语言中的数组一样，ECMAScript数组也是一组有序的数据，但跟其他语言不同的是，数组中每个槽位可以存储任意类型的数据。这意味着可以创建一个数组，它的第一个元素是字符串，第二个元素是数值，第三个是对象。ECMAScript数组也是动态大小的，会随着数据添加而自动增长。

6.2.1 创建数组

有几种基本的方式可以创建数组。一种是使用Array构造函数，比如：

```
let colors = new Array();
```

如果知道数组中元素的数量，那么可以给构造函数传入一个数值，然后length属性就会被自动创建并设置为这个值。比如，下面的代码会创建一个初始length为20的数组：

```
let colors = new Array(20);
```

也可以给Array构造函数传入要保存的元素。比如，下面的代码会创建一个包含3个字符串值的数组：

```
let colors = new Array("red", "blue", "green");
```

创建数组时可以给构造函数传一个值。这时候就有点问题了，因为如果这个值是数值，则会创建一个长度为指定数值的数组；而如果这个值是其他类型的，则会创建一个只包含该特定值的数组。下面看一个例子：

```

```

```
let colors = new Array(3);     // 创建一个包含3个元素的数组
let names = new Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
```

在使用Array构造函数时，也可以省略new操作符。结果是一样的，比如：

```
let colors = Array(3);     // 创建一个包含3个元素的数组
let names = Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
```

另一种创建数组的方式是使用**数组字面量**（array literal）表示法。数组字面量是在中括号中包含以逗号分隔的元素列表，如下面的例子所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含3个元素的数组
let names = [];                         // 创建一个空数组
let values = [1,2,];                    // 创建一个包含2个元素的数组
```

在这个例子中，第一行创建一个包含3个字符串的数组。第二行用一对空中括号创建了一个空数组。第三行展示了在数组最后一个值后面加逗号的效果：values是一个包含两个值（1和2）的数组。

> **注意** 与对象一样，在使用数组字面量表示法创建数组不会调用Array构造函数。

Array构造函数还有两个ES6新增的用于创建数组的静态方法：from()和of()。from()用于将类数组结构转换为数组实例，而of()用于将一组参数转换为数组实例。

Array.from()的第一个参数是一个类数组对象，即任何可迭代的结构，或者有一个length属性和可索引元素的结构。这种方式可用于很多场合：

```
// 字符串会被拆分为单字符数组
console.log(Array.from("Matt")); // ["M", "a", "t", "t"]

// 可以使用from()将集合和映射转换为一个新数组
const m = new Map().set(1, 2)
                   .set(3, 4);
const s = new Set().add(1)
                   .add(2)
                   .add(3)
                   .add(4);

console.log(Array.from(m)); // [[1, 2], [3, 4]]
console.log(Array.from(s)); // [1, 2, 3, 4]

// Array.from()对现有数组执行浅复制
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1);

console.log(a1);        // [1, 2, 3, 4]
alert(a1 === a2); // false
```

```
// 可以使用任何可迭代对象
const iter = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;

  }
};
console.log(Array.from(iter)); // [1, 2, 3, 4]

// arguments对象可以被轻松地转换为数组
function getArgsArray() {
  return Array.from(arguments);
}
console.log(getArgsArray(1, 2, 3, 4)); // [1, 2, 3, 4]

// from()也能转换带有必要属性的自定义对象
const arrayLikeObject = {
  0: 1,
  1: 2,
  2: 3,
  3: 4,
  length: 4
};
console.log(Array.from(arrayLikeObject)); // [1, 2, 3, 4]
```

`Array.from()`还接收第二个可选的映射函数参数。这个函数可以直接增强新数组的值，而无须像调用`Array.from().map()`那样先创建一个中间数组。还可以接收第三个可选参数，用于指定映射函数中`this`的值。但这个重写的`this`值在箭头函数中不适用。

```
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1, x => x**2);
const a3 = Array.from(a1, function(x) {return x**this.exponent}, {exponent: 2});
console.log(a2);  // [1, 4, 9, 16]
console.log(a3);  // [1, 4, 9, 16]
```

`Array.of()`可以把一组参数转换为数组。这个方法用于替代在ES6之前常用的`Array.prototype.slice.call(arguments)`，一种异常笨拙的将`arguments`对象转换为数组的写法：

```
console.log(Array.of(1, 2, 3, 4)); // [1, 2, 3, 4]
console.log(Array.of(undefined));  // [undefined]
```

6.2.2 数组空位

使用数组字面量初始化数组时，可以使用一串逗号来创建空位（hole）。ECMAScript会将逗号之间相应索引位置的值当成空位，ES6规范重新定义了该如何处理这些空位。

可以像下面这样创建一个空位数组：

```
const options = [,,,,,]; // 创建包含5个元素的数组
console.log(options.length);   // 5
console.log(options);          // [,,,,,]
```

ES6新增的方法和迭代器与早期ECMAScript版本中存在的方法行为不同。ES6新增方法普遍将这些空位当成存在的元素，只不过值为`undefined`：

```
const options = [1,,,,5];

for (const option of options) {
  console.log(option === undefined);
}
// false
// true
// true
// true
// false

const a = Array.from([,,,]); // 使用ES6的Array.from()创建的包含3个空位的数组
for (const val of a) {
  alert(val === undefined);
}
// true
// true
// true

alert(Array.of(...[,,,])); // [undefined, undefined, undefined]

for (const [index, value] of options.entries()) {
  alert(value);
}
// 1
// undefined
// undefined
// undefined
// 5
```

ES6之前的方法则会忽略这个空位，但具体的行为也会因方法而异：

```
const options = [1,,,,5];

// map()会跳过空位置
console.log(options.map(() => 6));  // [6, undefined, undefined, undefined, 6]
```

```
// join()视空位置为空字符串
console.log(options.join('-'));     // "1----5"
```

> **注意** 由于行为不一致和存在性能隐患，因此实践中要避免使用数组空位。如果确实需要空位，则可以显式地用undefined值代替。

6.2.3 数组索引

要取得或设置数组的值，需要使用中括号并提供相应值的数字索引，如下所示：

```
let colors = ["red", "blue", "green"];  // 定义一个字符串数组
alert(colors[0]);                       // 显示第一项
colors[2] = "black";                    // 修改第三项
colors[3] = "brown";                    // 添加第四项
```

在中括号中提供的索引表示要访问的值。如果索引小于数组包含的元素数，则返回存储在相应位置的元素，就像示例中colors[0]显示"red"一样。设置数组的值方法也是一样的，就是替换指定位置的值。如果把一个值设置给超过数组最大索引的索引，就像示例中的colors[3]，则数组长度会自动扩展到该索引值加1（示例中设置的索引3，所以数组长度变成了4）。

数组中元素的数量保存在length属性中，这个属性始终返回0或大于0的值，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含3个字符串的数组
let names = [];                         // 创建一个空数组

alert(colors.length); // 3
alert(names.length);  // 0
```

数组length属性的独特之处在于，它不是只读的。通过修改length属性，可以从数组末尾删除或添加元素。来看下面的例子：

```
let colors = ["red", "blue", "green"]; // 创建一个包含3个字符串的数组
colors.length = 2;
alert(colors[2]);  // undefined
```

这里，数组colors一开始有3个值。将length设置为2，就删除了最后一个（位置2的）值，因此colors[2]就没有值了。如果将length设置为大于数组元素数的值，则新添加的元素都将以undefined填充，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含3个字符串的数组
colors.length = 4;
alert(colors[3]);  // undefined
```

这里将数组colors的length设置为4，虽然数组只包含3个元素。位置3在数组中不存在，因此访问其值会返回特殊值undefined。

使用length属性可以方便地向数组末尾添加元素，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含3个字符串的数组
colors[colors.length] = "black";        // 添加一种颜色（位置3）
colors[colors.length] = "brown";        // 再添加一种颜色（位置4）
```

数组中最后一个元素的索引始终是length - 1，因此下一个新增槽位的索引就是length。每次在数组最后一个元素后面新增一项，数组的length属性都会自动更新，以反映变化。这意味着第二行的colors[colors.length]会在位置3添加一个新元素，下一行则会在位置4添加一个新元素。新的长度会在新增元素被添加到当前数组外部的位置上时自动更新。换句话说，就是length属性会更新为位置加上1，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含3个字符串的数组
colors[99] = "black";                   // 添加一种颜色（位置99）
alert(colors.length);                   // 100
```

这里，colors数组有一个值被插入到位置99，结果新length就变成了100（99 + 1）。这中间的所有元素，即位置3~98，实际上并不存在，因此在访问时会返回undefined。

> **注意** 数组最多可以包含4 294 967 295个元素，这对于大多数编程任务应该足够了。如果尝试添加更多项，则会导致抛出错误。以这个最大值作为初始值创建数组，可能导致脚本运行时间过长的错误。

6.2.4 检测数组

一个经典的ECMAScript问题是判断一个对象是不是数组。在只有一个网页（因而只有一个全局作用域）的情况下，使用instanceof操作符就足矣：

```
if (value instanceof Array){
  // 操作数组
}
```

使用instanceof的问题是假定只有一个全局执行上下文。如果网页里有多个框架，则可能涉及两个不同的全局执行上下文，因此就会有两个不同版本的Array构造函数。如果要把数组从一个框架传给另一个框架，则这个数组的构造函数将有别于在第二个框架内本地创建的数组。

为解决这个问题，ECMAScript提供了Array.isArray()方法。这个方法的目的就是确定一个值是否为数组，而不用管它是在哪个全局执行上下文中创建的。来看下面的例子：

```
if (Array.isArray(value)){
  // 操作数组
}
```

6.2.5 迭代器方法

在ES6中，Array的原型上暴露了3个用于检索数组内容的方法：keys()、values()和entries()。keys()返回数组索引的迭代器，values()返回数组元素的迭代器，而entries()返回索引/值对的迭代器：

```
const a = ["foo", "bar", "baz", "qux"];

// 因为这些方法都返回迭代器，所以可以将它们的内容
// 通过Array.from()直接转换为数组实例
const aKeys = Array.from(a.keys());
const aValues = Array.from(a.values());
const aEntries = Array.from(a.entries());

console.log(aKeys);     // [0, 1, 2, 3]
console.log(aValues);   // ["foo", "bar", "baz", "qux"]
console.log(aEntries);  // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

使用ES6的解构可以非常容易地在循环中拆分键/值对：

```
const a = ["foo", "bar", "baz", "qux"];

for (const [idx, element] of a.entries()) [
  alert(idx);
  alert(element);
}
// 0
// foo
// 1
// bar
// 2
// baz
// 3
// qux
```

> **注意** 虽然这些方法是ES6规范定义的，但在2017年底的时候仍有浏览器没有实现它们。

6.2.6 复制和填充方法

ES6新增了两个方法：批量复制方法fill()，以及填充数组方法copyWithin()。这两个方法的函数签名类似，都需要指定既有数组实例上的一个范围，包含开始索引，不包含结束索引。使用这个方法创建的数组不能缩放。

使用fill()方法可以向一个已有的数组中插入全部或部分相同的值。开始索引用于指定开始填充的位置，它是可选的。如果不提供结束索引，则一直填充到数组末尾。负值索引从数组末尾开始计算。也可以将负索引想象成数组长度加上它得到的一个正索引：

```
const zeroes = [0, 0, 0, 0, 0];

// 用5填充整个数组
zeroes.fill(5);
console.log(zeroes);  // [5, 5, 5, 5, 5]
zeroes.fill(0);       // 重置

// 用6填充索引大于等于3的元素
zeroes.fill(6, 3);
console.log(zeroes);  // [0, 0, 0, 6, 6]
zeroes.fill(0);       // 重置

// 用7填充索引大于等于1且小于3的元素
zeroes.fill(7, 1, 3);
console.log(zeroes);  // [0, 7, 7, 0, 0];
zeroes.fill(0);       // 重置

// 用8填充索引大于等于1且小于4的元素
// (-4 + zeroes.length = 1)
// (-1 + zeroes.length = 4)
zeroes.fill(8, -4, -1);
console.log(zeroes);  // [0, 8, 8, 8, 0];
```

`fill()`静默忽略超出数组边界、零长度及方向相反的索引范围：

```
const zeroes = [0, 0, 0, 0, 0];

// 索引过低，忽略
zeroes.fill(1, -10, -6);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引过高，忽略
zeroes.fill(1, 10, 15);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引反向，忽略
zeroes.fill(2, 4, 2);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引部分可用，填充可用部分
zeroes.fill(4, 3, 10)
console.log(zeroes);  // [0, 0, 0, 4, 4]
```

与`fill()`不同，`copyWithin()`会按照指定范围浅复制数组中的部分内容，然后将它们插入到指定索引开始的位置。开始索引和结束索引则与`fill()`使用同样的计算方法：

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

```
reset();

// 从ints中复制索引0开始的内容，插入到索引5开始的位置
// 在源索引或目标索引到达数组边界时停止
ints.copyWithin(5);
console.log(ints);  // [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
reset();

// 从ints中复制索引5开始的内容，插入到索引0开始的位置
ints.copyWithin(0, 5);
console.log(ints);  // [5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
reset();

// 从ints中复制索引0开始到索引3结束的内容
// 插入到索引4开始的位置
ints.copyWithin(4, 0, 3);
alert(ints);  // [0, 1, 2, 3, 0, 1, 2, 7, 8, 9]
reset();

// JavaScript引擎在插值前会完整复制范围内的值
// 因此复制期间不存在重写的风险
ints.copyWithin(2, 0, 6);
alert(ints);  // [0, 1, 0, 1, 2, 3, 4, 5, 8, 9]
reset();

// 支持负索引值，与fill()相对于数组末尾计算正向索引的过程是一样的
ints.copyWithin(-4, -7, -3);
alert(ints);  // [0, 1, 2, 3, 4, 5, 3, 4, 5, 6]
```

`copyWithin()`静默忽略超出数组边界、零长度及方向相反的索引范围：

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引过低，忽略
ints.copyWithin(1, -15, -12);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset()

// 索引过高，忽略
ints.copyWithin(1, 12, 15);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引反向，忽略
ints.copyWithin(2, 4, 2);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引部分可用，复制、填充可用部分
```

```
ints.copyWithin(4, 7, 10)
alert(ints);  // [0, 1, 2, 3, 7, 8, 9, 7, 8, 9];
```

6.2.7 转换方法

前面提到过，所有对象都有toLocaleString()、toString()和valueOf()方法。其中，valueOf()返回的还是数组本身。而toString()返回由数组中每个值的等效字符串拼接而成的一个逗号分隔的字符串。也就是说，对数组的每个值都会调用其toString()方法，以得到最终的字符串。来看下面的例子：

```
let colors = ["red", "blue", "green"]; // 创建一个包含3个字符串的数组
alert(colors.toString());   // red,blue,green
alert(colors.valueOf());    // red,blue,green
alert(colors);              // red,blue,green
```

首先是被显式调用的toString()和valueOf()方法，它们分别返回了数组的字符串表示，即将所有字符串组合起来，以逗号分隔。最后一行代码直接用alert()显示数组，因为alert()期待字符串，所以会在后台调用数组的toString()方法，从而得到跟前面一样的结果。

toLocaleString()方法也可能返回跟toString()和valueOf()相同的结果，但也不一定。在调用数组的toLocaleString()方法时，会得到一个逗号分隔的数组值的字符串。它与另外两个方法唯一的区别是，为了得到最终的字符串，会调用数组每个值的toLocaleString()方法，而不是toString()方法。看下面的例子：

```
let person1 = {
  toLocaleString() {
    return "Nikolaos";
  },

  toString() {
    return "Nicholas";
  }
};

let person2 = {
  toLocaleString() {
    return "Grigorios";
  },

  toString() {
    return "Greg";
  }
};

let people = [person1, person2];
alert(people);                   // Nicholas,Greg
alert(people.toString());        // Nicholas,Greg
alert(people.toLocaleString());  // Nikolaos,Grigorios
```

这里定义了两个对象person1和person2，它们都定义了toString()和toLocaleString()方法，而且返回不同的值。然后又创建了一个包含这两个对象的数组people。在将数组传给alert()时，输出的是"Nicholas,Greg"，这是因为会在数组每一项上调用toString()方法（与下一行显式调用toString()方法结果一样）。而在调用数组的toLocaleString()方法时，结果变成了"Nikolaos, Grigorios"，这是因为调用了数组每一项的toLocaleString()方法。

继承的方法toLocaleString()以及toString()都返回数组值的逗号分隔的字符串。如果想使用不同的分隔符，则可以使用join()方法。join()方法接收一个参数，即字符串分隔符，返回包含所有项的字符串。来看下面的例子：

```
let colors = ["red", "green", "blue"];
alert(colors.join(","));     // red,green,blue
alert(colors.join("||"));    // red||green||blue
```

这里在colors数组上调用了join()方法，得到了与调用toString()方法相同的结果。传入逗号，结果就是逗号分隔的字符串。最后一行给join()传入了双竖线，得到了字符串"red||green||blue"。如果不给join()传入任何参数，或者传入undefined，则仍然使用逗号作为分隔符。

> **注意** 如果数组中某一项是null或undefined，则在join()、toLocaleString()、toString()和valueOf()返回的结果中会以空字符串表示。

6.2.8 栈方法

ECMAScript给数组提供几个方法，让它看起来像是另外一种数据结构。数组对象可以像栈一样，也就是一种限制插入和删除项的数据结构。栈是一种后进先出（LIFO，Last-In-First-Out）的结构，也就是最近添加的项先被删除。数据项的插入（称为**推入**，push）和删除（称为**弹出**，pop）只在栈的一个地方发生，即栈顶。ECMAScript数组提供了push()和pop()方法，以实现类似栈的行为。

push()方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度。pop()方法则用于删除数组的最后一项，同时减少数组的length值，返回被删除的项。来看下面的例子：

```
let colors = new Array();                 // 创建一个数组
let count = colors.push("red", "green");  // 推入两项
alert(count);                             // 2

count = colors.push("black");  // 再推入一项
alert(count);                  // 3

let item = colors.pop();       // 取得最后一项
alert(item);                   // black
alert(colors.length);          // 2
```

这里创建了一个当作栈来使用的数组（注意不需要任何额外的代码，push()和pop()都是数组的默认方法）。首先，使用push()方法把两个字符串推入数组末尾，将结果保存在变量count中（结果为2）。

然后，再推入另一个值，再把结果保存在count中。因为现在数组中有3个元素，所以push()返回3。在调用pop()时，会返回数组的最后一项，即字符串"black"。此时数组还有两个元素。

栈方法可以与数组的其他任何方法一起使用，如下例所示：

```
let colors = ["red", "blue"];
colors.push("brown");        // 再添加一项
colors[3] = "black";         // 添加一项
alert(colors.length);        // 4

let item = colors.pop();     // 取得最后一项
alert(item);                 // black
```

这里先初始化了包含两个字符串的数组，然后通过push()添加了第三个值，第四个值是通过直接在位置3上赋值添加的。调用pop()时，返回了字符串"black"，也就是最后添加到数组的字符串。

6.2.9 队列方法

就像栈是以LIFO形式限制访问的数据结构一样，队列以先进先出（FIFO，First-In-First-Out）形式限制访问。队列在列表末尾添加数据，但从列表开头获取数据。因为有了在数据末尾添加数据的push()方法，所以要模拟队列就差一个从数组开头取得数据的方法了。这个数组方法叫shift()，它会删除数组的第一项并返回它，然后数组长度减1。使用shift()和push()，可以把数组当成队列来使用：

```
let colors = new Array();                 // 创建一个数组
let count = colors.push("red", "green");  // 推入两项
alert(count);                             // 2

count = colors.push("black"); // 再推入一项
alert(count);                  // 3

let item = colors.shift();  // 取得第一项
alert(item);                // red
alert(colors.length);       // 2
```

这个例子创建了一个数组并用push()方法推入三个值。加粗的那行代码使用shift()方法取得了数组的第一项，即"red"。删除这一项之后，"green"成为第一个元素，"black"成为第二个元素，数组此时就包含两项。

ECMAScript也为数组提供了unshift()方法。顾名思义，unshift()就是执行跟shift()相反的操作：在数组开头添加任意多个值，然后返回新的数组长度。通过使用unshift()和pop()，可以在相反方向上模拟队列，即在数组开头添加新数据，在数组末尾取得数据，如下例所示：

```
let colors = new Array();                   // 创建一个数组
let count = colors.unshift("red", "green");  // 从数组开头推入两项
alert(count);                               // 2

count = colors.unshift("black");  // 再推入一项
alert(count);                     // 3

let item = colors.pop();  // 取得最后一项
```

```
alert(item);              // green
alert(colors.length);     // 2
```

这里，先创建一个数组，再通过unshift()填充数组。首先，给数组添加"red"和"green"，再添加"black"，得到["black","red","green"]。调用pop()时，删除最后一项"green"并返回它。

6.2.10 排序方法

数组有两个方法可以用来对元素重新排序：reverse()和sort()。顾名思义，reverse()方法就是将数组元素反向排列。比如：

```
let values = [1, 2, 3, 4, 5];
values.reverse();
alert(values);  // 5,4,3,2,1
```

这里，数组values的初始状态为[1,2,3,4,5]。通过调用reverse()反向排序，得到了[5,4,3,2,1]。这个方法很直观，但不够灵活，所以才有了sort()方法。

默认情况下，sort()会按照升序重新排列数组元素，即最小的值在前面，最大的值在后面。为此，sort()会在每一项上调用String()转型函数，然后比较字符串来决定顺序。即使数组的元素都是数值，也会先把数组转换为字符串再比较、排序。比如：

```
let values = [0, 1, 5, 10, 15];
values.sort();
alert(values);  // 0,1,10,15,5
```

一开始数组中数值的顺序是正确的，但调用sort()会按照这些数值的字符串形式重新排序。因此，即使5小于10，但字符串"10"在字符串"5"的前头，所以10还是会排到5前面。很明显，这在多数情况下都不是最合适的。为此，sort()方法可以接收一个**比较函数**，用于判断哪个值应该排在前面。

比较函数接收两个参数，如果第一个参数应该排在第二个参数前面，就返回负值；如果两个参数相等，就返回0；如果第一个参数应该排在第二个参数后面，就返回正值。下面是使用简单比较函数的一个例子：

```
function compare(value1, value2) {
  if (value1 < value2) {
    return -1;
  } else if (value1 > value2) {
    return 1;
  } else {
    return 0;
  }
}
```

这个比较函数可以适用于大多数数据类型，可以把它当作参数传给sort()方法，如下所示：

```
let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);  // 0,1,5,10,15
```

在给sort()方法传入比较函数后，数组中的数值在排序后保持了正确的顺序。当然，比较函数也可以产生降序效果，只要把返回值交换一下即可：

```
function compare(value1, value2) {
  if (value1 < value2) {
    return 1;
  } else if (value1 > value2) {
    return -1;
  } else {
    return 0;
  }
}

let values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);  // 15,10,5,1,0
```

此外，这个比较函数还可简写为一个箭头函数：

```
let values = [0, 1, 5, 10, 15];
values.sort((a, b) => a < b ? 1 : a > b ? -1 : 0);
alert(values); // 15,10,5,1,0
```

在这个修改版函数中，如果第一个值应该排在第二个值后面则返回1，如果第一个值应该排在第二个值前面则返回-1。交换这两个返回值之后，较大的值就会排在前头，数组就会按照降序排序。当然，如果只是想反转数组的顺序，reverse()更简单也更快。

> **注意** reverse()和sort()都返回调用它们的数组的引用。

如果数组的元素是数值，或者是其valueOf()方法返回数值的对象（如Date对象），这个比较函数还可以写得更简单，因为这时可以直接用第二个值减去第一个值：

```
function compare(value1, value2){
  return value2 - value1;
}
```

比较函数就是要返回小于0、0和大于0的数值，因此减法操作完全可以满足要求。

6.2.11 操作方法

对于数组中的元素，我们有很多操作方法。比如，concat()方法可以在现有数组全部元素基础上创建一个新数组。它首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组。如果传入一个或多个数组，则concat()会把这些数组的每一项都添加到结果数组。如果参数不是数组，则直接把它们添加到结果数组末尾。来看下面的例子：

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);

console.log(colors);   // ["red", "green","blue"]
console.log(colors2);  // ["red", "green", "blue", "yellow", "black", "brown"]
```

这里先创建一个包含3个值的数组colors。然后colors调用concat()方法，传入字符串"yellow"和一个包含"black"和"brown"的数组。保存在colors2中的结果就是["red", "green", "blue", "yellow", "black", "brown"]。原始数组colors保持不变。

打平数组参数的行为可以重写，方法是在参数数组上指定一个特殊的符号：Symbol.isConcatSpreadable。这个符号能够阻止concat()打平参数数组。相反，把这个值设置为true可以强制打平类数组对象：

```
let colors = ["red", "green", "blue"];
let newColors = ["black", "brown"];
let moreNewColors = {
  [Symbol.isConcatSpreadable]: true,
  length: 2,
  0: "pink",
  1: "cyan"
};

newColors[Symbol.isConcatSpreadable] = false;

// 强制不打平数组
let colors2 = colors.concat("yellow", newColors);

// 强制打平类数组对象
let colors3 = colors.concat(moreNewColors);

console.log(colors);   // ["red", "green", "blue"]
console.log(colors2);  // ["red", "green", "blue", "yellow", ["black", "brown"]]
console.log(colors3);  // ["red", "green", "blue", "pink", "cyan"]
```

接下来，方法slice()用于创建一个包含原有数组中一个或多个元素的新数组。slice()方法可以接收一个或两个参数：返回元素的开始索引和结束索引。如果只有一个参数，则slice()会返回该索引到数组末尾的所有元素。如果有两个参数，则slice()返回从开始索引到结束索引对应的所有元素，其中不包含结束索引对应的元素。记住，这个操作不影响原始数组。来看下面的例子：

```
let colors = ["red", "green", "blue", "yellow", "purple"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1, 4);
```

```
alert(colors2);  // green,blue,yellow,purple
alert(colors3);  // green,blue,yellow
```

这里，colors数组一开始有5个元素。调用slice()传入1会得到包含4个元素的新数组。其中不包括"red"，这是因为拆分操作要从位置1开始，即从"green"开始。得到的colors2数组包含"green"、"blue"、"yellow"和"purple"。colors3数组是通过调用slice()并传入1和4得到的，即从位置1开始复制到位置3。因此colors3包含"green"、"blue"和"yellow"。

> **注意** 如果slice()的参数有负值，那么就以数值长度加上这个负值的结果确定位置。比如，在包含5个元素的数组上调用slice(-2,-1)，就相当于调用slice(3,4)。如果结束位置小于开始位置，则返回空数组。

或许最强大的数组方法就属splice()了，使用它的方式可以有很多种。splice()的主要目的是在数组中间插入元素，但有3种不同的方式使用这个方法。

- **删除**。需要给splice()传2个参数：要删除的第一个元素的位置和要删除的元素数量。可以从数组中删除任意多个元素，比如splice(0, 2)会删除前两个元素。
- **插入**。需要给splice()传3个参数：开始位置、0（要删除的元素数量）和要插入的元素，可以在数组中指定的位置插入元素。第三个参数之后还可以传第四个、第五个参数，乃至任意多个要插入的元素。比如，splice(2, 0, "red", "green")会从数组位置2开始插入字符串"red"和"green"。
- **替换**。splice()在删除元素的同时可以在指定位置插入新元素，同样要传入3个参数：开始位置、要删除元素的数量和要插入的任意多个元素。要插入的元素数量不一定跟删除的元素数量一致。比如，splice(2, 1, "red", "green")会在位置2删除一个元素，然后从该位置开始向数组中插入"red"和"green"。

splice()方法始终返回这样一个数组，它包含从数组中被删除的元素（如果没有删除元素，则返回空数组）。以下示例展示了上述3种使用方式。

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0,1);  // 删除第一项
alert(colors);                     // green,blue
alert(removed);                    // red，只有一个元素的数组

removed = colors.splice(1, 0, "yellow", "orange");   // 在位置1插入两个元素
alert(colors);                                       // green,yellow,orange,blue
alert(removed);                                      // 空数组

removed = colors.splice(1, 1, "red", "purple");  // 插入两个值，删除一个元素
alert(colors);                                   // green,red,purple,orange,blue
alert(removed);                                  // yellow，只有一个元素的数组
```

这个例子中，colors数组一开始包含3个元素。第一次调用splice()时，只删除了第一项，colors中还有"green"和"blue"。第二次调用slice()时，在位置1插入两项，然后colors包含"green"、"yellow"、"orange"和"blue"。这次没删除任何项，因此返回空数组。最后一次调用splice()时删除了位置1上的一项，同时又插入了"red"和"purple"。最后，colors数组包含"green"、"red"、"purple"、"orange"和"blue"。

6.2.12 搜索和位置方法

ECMAScript提供两类搜索数组的方法：按严格相等搜索和按断言函数搜索。

1. **严格相等**

   ECMAScript提供了3个严格相等的搜索方法：`indexOf()`、`lastIndexOf()`和`includes()`。其中，前两个方法在所有版本中都可用，而第三个方法是ECMAScript 7新增的。这些方法都接收两个参数：要查找的元素和一个可选的起始搜索位置。`indexOf()`和`includes()`方法从数组前头（第一项）开始向后搜索，而`lastIndexOf()`从数组末尾（最后一项）开始向前搜索。

   `indexOf()`和`lastIndexOf()`都返回要查找的元素在数组中的位置，如果没找到则返回-1。`includes()`返回布尔值，表示是否至少找到一个与指定元素匹配的项。在比较第一个参数跟数组每一项时，会使用全等（`===`）比较，也就是说两项必须严格相等。下面来看一些例子：

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

alert(numbers.indexOf(4));          // 3
alert(numbers.lastIndexOf(4));      // 5
alert(numbers.includes(4));         // true

alert(numbers.indexOf(4, 4));       // 5
alert(numbers.lastIndexOf(4, 4));   // 3
alert(numbers.includes(4, 7));      // false

let person = { name: "Nicholas" };
let people = [{ name: "Nicholas" }];
let morePeople = [person];

alert(people.indexOf(person));      // -1
alert(morePeople.indexOf(person));  // 0
alert(people.includes(person));     // false
alert(morePeople.includes(person)); // true
```

2. **断言函数**

   ECMAScript也允许按照定义的断言函数搜索数组，每个索引都会调用这个函数。断言函数的返回值决定了相应索引的元素是否被认为匹配。

   断言函数接收3个参数：元素、索引和数组本身。其中元素是数组中当前搜索的元素，索引是当前元素的索引，而数组就是正在搜索的数组。断言函数返回真值，表示是否匹配。

   `find()`和`findIndex()`方法使用了断言函数。这两个方法都从数组的最小索引开始。`find()`返回第一个匹配的元素，`findIndex()`返回第一个匹配元素的索引。这两个方法也都接收第二个可选的参数，用于指定断言函数内部`this`的值。

```
const people = [
  {
    name: "Matt",
    age: 27
```

```
  },
  {
    name: "Nicholas",
    age: 29
  }
];

alert(people.find((element, index, array) => element.age < 28));
// {name: "Matt", age: 27}

alert(people.findIndex((element, index, array) => element.age < 28));
// 0
```

找到匹配项后，这两个方法都不再继续搜索。

```
const evens = [2, 4, 6];

// 找到匹配后，永远不会检查数组的最后一个元素
evens.find((element, index, array) => {
  console.log(element);
  console.log(index);
  console.log(array);
  return element === 4;
});
// 2
// 0
// [2, 4, 6]
// 4
// 1
// [2, 4, 6]
```

6.2.13 迭代方法

ECMAScript为数组定义了5个迭代方法。每个方法接收两个参数：以每一项为参数运行的函数，以及可选的作为函数运行上下文的作用域对象（影响函数中`this`的值）。传给每个方法的函数接收3个参数：数组元素、元素索引和数组本身。因具体方法而异，这个函数的执行结果可能会也可能不会影响方法的返回值。数组的5个迭代方法如下。

- `every()`：对数组每一项都运行传入的函数，如果对每一项函数都返回`true`，则这个方法返回`true`。
- `filter()`：对数组每一项都运行传入的函数，函数返回`true`的项会组成数组之后返回。
- `forEach()`：对数组每一项都运行传入的函数，没有返回值。
- `map()`：对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组。
- `some()`：对数组每一项都运行传入的函数，如果有一项函数返回`true`，则这个方法返回`true`。

这些方法都不改变调用它们的数组。

在这些方法中，`every()`和`some()`是最相似的，都是从数组中搜索符合某个条件的元素。对`every()`来说，传入的函数必须对每一项都返回`true`，它才会返回`true`；否则，它就返回`false`。而对`some()`来说，只要有一项让传入的函数返回`true`，它就会返回`true`。下面是一个例子：

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

let everyResult = numbers.every((item, index, array) => item > 2);
alert(everyResult);  // false

let someResult = numbers.some((item, index, array) => item > 2);
alert(someResult);   // true
```

以上代码调用了every()和some()，传入的函数都是在给定项大于2时返回true。every()返回false是因为并不是每一项都能达到要求。而some()返回true是因为至少有一项满足条件。

下面再看一看filter()方法。这个方法基于给定的函数来决定某一项是否应该包含在它返回的数组中。比如，要返回一个所有数值都大于2的数组，可以使用如下代码：

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

let filterResult = numbers.filter((item, index, array) => item > 2);
alert(filterResult);  // 3,4,5,4,3
```

这里，调用filter()返回的数组包含3、4、5、4、3，因为只有对这些项传入的函数才返回true。这个方法非常适合从数组中筛选满足给定条件的元素。

接下来map()方法也会返回一个数组。这个数组的每一项都是对原始数组中同样位置的元素运行传入函数而返回的结果。例如，可以将一个数组中的每一项都乘以2，并返回包含所有结果的数组，如下所示：

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

let mapResult = numbers.map((item, index, array) => item * 2);

alert(mapResult);  // 2,4,6,8,10,8,6,4,2
```

以上代码返回了一个数组，包含原始数组中每个值乘以2的结果。这个方法非常适合创建一个与原始数组元素一一对应的新数组。

最后，再来看一看forEach()方法。这个方法只会对每一项运行传入的函数，没有返回值。本质上，forEach()方法相当于使用for循环遍历数组。比如：

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];

numbers.forEach((item, index, array) => {
  // 执行某些操作
});
```

数组的这些迭代方法通过执行不同操作方便了对数组的处理。

6.2.14 归并方法

ECMAScript为数组提供了两个归并方法：reduce()和reduceRight()。这两个方法都会迭代数组的所有项，并在此基础上构建一个最终返回值。reduce()方法从数组第一项开始遍历到最后一项。而reduceRight()从最后一项开始遍历至第一项。

这两个方法都接收两个参数：对每一项都会运行的归并函数，以及可选的以之为归并起点的初始值。传给reduce()和reduceRight()的函数接收4个参数：上一个归并值、当前项、当前项的索引和数组本身。这个函数返回的任何值都会作为下一次调用同一个函数的第一个参数。如果没有给这两个方法传入可选的第二个参数（作为归并起点值），则第一次迭代将从数组的第二项开始，因此传给归并函数的第一个参数是数组的第一项，第二个参数是数组的第二项。

可以使用reduce()函数执行累加数组中所有数值的操作，比如：

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduce((prev, cur, index, array) => prev + cur);

alert(sum);  // 15
```

第一次执行归并函数时，prev是1，cur是2。第二次执行时，prev是3（1 + 2），cur是3（数组第三项）。如此递进，直到把所有项都遍历一次，最后返回归并结果。

reduceRight()方法与之类似，只是方向相反。来看下面的例子：

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduceRight(function(prev, cur, index, array){
  return prev + cur;
});
alert(sum); // 15
```

在这里，第一次调用归并函数时prev是5，而cur是4。当然，最终结果相同，因为归并操作都是简单的加法。

究竟是使用reduce()还是reduceRight()，只取决于遍历数组元素的方向。除此之外，这两个方法没什么区别。

6.3 定型数组

定型数组（typed array）是ECMAScript新增的结构，目的是提升向原生库传输数据的效率。实际上，JavaScript并没有“TypedArray”类型，它所指的其实是一种特殊的包含数值类型的数组。为理解如何使用定型数组，有必要先了解一下它的用途。

6.3.1 历史

随着浏览器的流行，不难想象人们会满怀期待地通过它来运行复杂的3D应用程序。早在2006年，Mozilla、Opera等浏览器提供商就实验性地在浏览器中增加了用于渲染复杂图形应用程序的编程平台，无须安装任何插件。其目标是开发一套JavaScript API，从而充分利用3D图形API和GPU加速，以便在<canvas>元素上渲染复杂的图形。

1. **WebGL**

   最后的JavaScript API是基于OpenGL ES（OpenGL for Embedded Systems）2.0规范的。OpenGL ES是OpenGL专注于2D和3D计算机图形的子集。这个新API被命名为WebGL（Web Grapics Library），于2011年发布1.0版。有了它，开发者就能够编写涉及复杂图形的应用程序，它会被兼容WebGL的浏览器原生解释执行。

   在WebGL的早期版本中，因为JavaScript数组与原生数组之间不匹配，所以出现了性能问题。图形驱动程序API通常不需要以JavaScript默认双精度浮点格式传递给它们的数值，而这恰恰是JavaScript数组在内存中的格式。因此，每次WebGL与JavaScript运行时之间传递数组时，WebGL绑定都需要在目标环境分配新数组，以其当前格式迭代数组，然后将数值转型为新数组中的适当格式，而这些要花费很多时间。

2. **定型数组**

   这当然是难以接受的，Mozilla为解决这个问题而实现了`CanvasFloatArray`。这是一个提供JavaScript接口的、C语言风格的浮点值数组。JavaScript运行时使用这个类型可以分配、读取和写入数组。这个数组可以直接传给底层图形驱动程序API，也可以直接从底层获取到。最终，`CanvasFloatArray`变成了`Float32Array`，也就是今天定型数组中可用的第一个“类型”。

6.3.2 `ArrayBuffer`

`Float32Array`实际上是一种“视图”，可以允许JavaScript运行时访问一块名为`ArrayBuffer`的预分配内存。`ArrayBuffer`是所有定型数组及视图引用的基本单位。

> **注意** `SharedArrayBuffer`是`ArrayBuffer`的一个变体，可以无须复制就在执行上下文间传递它。关于这种类型，请参考第27章。

`ArrayBuffer()`是一个普通的JavaScript构造函数，可用于在内存中分配特定数量的字节空间。

```
const buf = new ArrayBuffer(16);  // 在内存中分配16字节
alert(buf.byteLength);            // 16
```

`ArrayBuffer`一经创建就不能再调整大小。不过，可以使用`slice()`复制其全部或部分到一个新实例中：

```
const buf1 = new ArrayBuffer(16);
const buf2 = buf1.slice(4, 12);
alert(buf2.byteLength);  // 8
```

`ArrayBuffer`某种程度上类似于C++的`malloc()`，但也有几个明显的区别。

- `malloc()`在分配失败时会返回一个`null`指针。`ArrayBuffer`在分配失败时会抛出错误。
- `malloc()`可以利用虚拟内存，因此最大可分配尺寸只受可寻址系统内存限制。`ArrayBuffer`分配的内存不能超过`Number.MAX_SAFE_INTEGER`（$2^{53}-1$）字节。
- `malloc()`调用成功不会初始化实际的地址。声明`ArrayBuffer`则会将所有二进制位初始化为0。
- 通过`malloc()`分配的堆内存除非调用`free()`或程序退出，否则系统不能再使用。而通过声明`ArrayBuffer`分配的堆内存可以被当成垃圾回收，不用手动释放。

不能仅通过对ArrayBuffer的引用就读取或写入其内容。要读取或写入ArrayBuffer，就必须通过视图。视图有不同的类型，但引用的都是ArrayBuffer中存储的二进制数据。

6.3.3 DataView

第一种允许你读写ArrayBuffer的视图是DataView。这个视图专为文件I/O和网络I/O设计，其API支持对缓冲数据的高度控制，但相比于其他类型的视图性能也差一些。DataView对缓冲内容没有任何预设，也不能迭代。

必须在对已有的ArrayBuffer读取或写入时才能创建DataView实例。这个实例可以使用全部或部分ArrayBuffer，且维护着对该缓冲实例的引用，以及视图在缓冲中开始的位置。

```
const buf = new ArrayBuffer(16);

// DataView默认使用整个ArrayBuffer
const fullDataView = new DataView(buf);
alert(fullDataView.byteOffset);      // 0
alert(fullDataView.byteLength);      // 16
alert(fullDataView.buffer === buf);  // true

// 构造函数接收一个可选的字节偏移量和字节长度
//   byteOffset=0表示视图从缓冲起点开始
//   byteLength=8限制视图为前8个字节
const firstHalfDataView = new DataView(buf, 0, 8);
alert(firstHalfDataView.byteOffset);      // 0
alert(firstHalfDataView.byteLength);      // 8
alert(firstHalfDataView.buffer === buf);  // true

// 如果不指定，则DataView会使用剩余的缓冲
//   byteOffset=8表示视图从缓冲的第9个字节开始
//   byteLength未指定，默认为剩余缓冲
const secondHalfDataView = new DataView(buf, 8);
alert(secondHalfDataView.byteOffset);      // 8
alert(secondHalfDataView.byteLength);      // 8
alert(secondHalfDataView.buffer === buf);  // true
```

要通过DataView读取缓冲，还需要几个组件。

- 首先是要读或写的字节偏移量。可以看成DataView中的某种“地址”。
- DataView应该使用ElementType来实现JavaScript的Number类型到缓冲内二进制格式的转换。
- 最后是内存中值的字节序。默认为大端字节序。

1. **ElementType**

   DataView对存储在缓冲内的数据类型没有预设。它暴露的API强制开发者在读、写时指定一个ElementType，然后DataView就会忠实地为读、写而完成相应的转换。

   ECMAScript 6支持8种不同的ElementType（见下表）。

ElementType	字节	说明	等价的C类型	值的范围

ElementType	字节	说明	等价的C类型	值的范围
Int8	1	8位有符号整数	signed char	-128~127
Uint8	1	8位无符号整数	unsigned char	0~255
Int16	2	16位有符号整数	short	-32 768~32 767
Uint16	2	16位无符号整数	unsigned short	0~65 535
Int32	4	32位有符号整数	int	-2 147 483 648~2 147 483 647
Uint32	4	32位无符号整数	unsigned int	0~4 294 967 295
Float32	4	32位IEEE-754浮点数	float	-3.4e+38~+3.4e+38
Float64	8	64位IEEE-754浮点数	double	-1.7e+308~+1.7e+308

DataView为上表中的每种类型都暴露了get和set方法，这些方法使用byteOffset（字节偏移量）定位要读取或写入值的位置。类型是可以互换使用的，如下例所示：

```
// 在内存中分配两个字节并声明一个DataView
const buf = new ArrayBuffer(2);
const view = new DataView(buf);

// 说明整个缓冲确实所有二进制位都是0
// 检查第一个和第二个字符
alert(view.getInt8(0));  // 0
alert(view.getInt8(1));  // 0
// 检查整个缓冲
alert(view.getInt16(0)); // 0

// 将整个缓冲都设置为1
// 255的二进制表示是11111111 (2^8 - 1)
view.setUint8(0, 255);

// DataView会自动将数据转换为特定的ElementType
// 255的十六进制表示是0xFF
view.setUint8(1, 0xFF);

// 现在，缓冲里都是1了
// 如果把它当成二补数的有符号整数，则应该是-1
alert(view.getInt16(0)); // -1
```

2. **字节序**

前面例子中的缓冲有意回避了字节序的问题。“字节序”指的是计算系统维护的一种字节顺序的约定。DataView只支持两种约定：大端字节序和小端字节序。大端字节序也称为“网络字节序”，意思是最高有效位保存在第一个字节，而最低有效位保存在最后一个字节。小端字节序正好相反，即最低有效位保存在第一个字节，最高有效位保存在最后一个字节。

JavaScript运行时所在系统的原生字节序决定了如何读取或写入字节，但DataView并不遵守这个约定。对一段内存而言，DataView是一个中立接口，它会遵循你指定的字节序。DataView的所有API方法都以大端字节序作为默认值，但接收一个可选的布尔值参数，设置为true即可启用小端字节序。

```
// 在内存中分配两个字节并声明一个DataView
const buf = new ArrayBuffer(2);
const view = new DataView(buf);

// 填充缓冲，让第一位和最后一位都是1
view.setUint8(0, 0x80); // 设置最左边的位等于1
view.setUint8(1, 0x01); // 设置最右边的位等于1

// 缓冲内容（为方便阅读，人为加了空格）
// 0x8  0x0  0x0  0x1
// 1000 0000 0000 0001

// 按大端字节序读取Uint16
// 0x80是高字节，0x01是低字节
// 0x8001 = 2^15 + 2^0 = 32768 + 1 = 32769
alert(view.getUint16(0)); // 32769

// 按小端字节序读取Uint16
// 0x01是高字节，0x80是低字节
// 0x0180 = 2^8 + 2^7 = 256 + 128 = 384
alert(view.getUint16(0, true)); // 384

// 按大端字节序写入Uint16
view.setUint16(0, 0x0004);

// 缓冲内容（为方便阅读，人为加了空格）
// 0x0  0x0  0x0  0x4
// 0000 0000 0000 0100

alert(view.getUint8(0)); // 0
alert(view.getUint8(1)); // 4

// 按小端字节序写入Uint16
view.setUint16(0, 0x0002, true);

// 缓冲内容（为方便阅读，人为加了空格）
// 0x0  0x2  0x0  0x0
// 0000 0010 0000 0000

alert(view.getUint8(0)); // 2
alert(view.getUint8(1)); // 0
```

3. **边界情形**

DataView完成读、写操作的前提是必须有充足的缓冲区，否则就会抛出RangeError：

```
const buf = new ArrayBuffer(6);
const view = new DataView(buf);

// 尝试读取部分超出缓冲范围的值
view.getInt32(4);
// RangeError

// 尝试读取超出缓冲范围的值
view.getInt32(8);
// RangeError

// 尝试读取超出缓冲范围的值
view.getInt32(-1);
// RangeError

// 尝试写入超出缓冲范围的值
view.setInt32(4, 123);
// RangeError
```

`DataView`在写入缓冲里会尽最大努力把一个值转换为适当的类型，后备为0。如果无法转换，则抛出错误：

```
const buf = new ArrayBuffer(1);
const view = new DataView(buf);

view.setInt8(0, 1.5);
alert(view.getInt8(0)); // 1

view.setInt8(0, [4]);
alert(view.getInt8(0)); // 4

view.setInt8(0, 'f');
alert(view.getInt8(0)); // 0

view.setInt8(0, Symbol());
// TypeError
```

6.3.4 定型数组

定型数组是另一种形式的`ArrayBuffer`视图。虽然概念上与`DataView`接近，但定型数组的区别在于，它特定于一种`ElementType`且遵循系统原生的字节序。相应地，定型数组提供了适用面更广的API和更高的性能。设计定型数组的目的就是提高与WebGL等原生库交换二进制数据的效率。由于定型数组的二进制表示对操作系统而言是一种容易使用的格式，JavaScript引擎可以重度优化算术运算、按位运算和其他对定型数组的常见操作，因此使用它们速度极快。

创建定型数组的方式包括读取已有的缓冲、使用自有缓冲、填充可迭代结构，以及填充基于任意类型的定型数组。另外，通过`<ElementType>.from()`和`<ElementType>.of()`也可以创建定型数组：

```
// 创建一个12字节的缓冲
const buf = new ArrayBuffer(12);
// 创建一个引用该缓冲的Int32Array
const ints = new Int32Array(buf);
// 这个定型数组知道自己的每个元素需要4字节
// 因此长度为3
alert(ints.length); // 3

// 创建一个长度为6的Int32Array
const ints2 = new Int32Array(6);
// 每个数值使用4字节，因此ArrayBuffer是24字节
alert(ints2.length);              // 6
// 类似DataView，定型数组也有一个指向关联缓冲的引用
alert(ints2.buffer.byteLength);  // 24

// 创建一个包含[2, 4, 6, 8]的Int32Array
const ints3 = new Int32Array([2, 4, 6, 8]);
alert(ints3.length);             // 4
alert(ints3.buffer.byteLength); // 16
alert(ints3[2]);                 // 6

// 通过复制ints3的值创建一个Int16Array
const ints4 = new Int16Array(ints3);
// 这个新类型数组会分配自己的缓冲
// 对应索引的每个值会相应地转换为新格式
alert(ints4.length);             // 4
alert(ints4.buffer.byteLength); // 8
alert(ints4[2]);                 // 6

// 基于普通数组来创建一个Int16Array
const ints5 = Int16Array.from([3, 5, 7, 9]);
alert(ints5.length);             // 4
alert(ints5.buffer.byteLength); // 8
alert(ints5[2]);                 // 7

// 基于传入的参数创建一个Float32Array
const floats = Float32Array.of(3.14, 2.718, 1.618);
alert(floats.length);             // 3
alert(floats.buffer.byteLength); // 12
alert(floats[2]);                 // 1.6180000305175781
```

定型数组的构造函数和实例都有一个`BYTES_PER_ELEMENT`属性，返回该类型数组中每个元素的大小：

```
alert(Int16Array.BYTES_PER_ELEMENT);  // 2
alert(Int32Array.BYTES_PER_ELEMENT);  // 4

const ints = new Int32Array(1),
      floats = new Float64Array(1);

alert(ints.BYTES_PER_ELEMENT);        // 4
alert(floats.BYTES_PER_ELEMENT);      // 8
```

如果定型数组没有用任何值初始化，则其关联的缓冲会以0填充：

```
const ints = new Int32Array(4);
alert(ints[0]);  // 0
alert(ints[1]);  // 0
alert(ints[2]);  // 0
alert(ints[3]);  // 0
```

1. **定型数组行为**

   从很多方面看，定型数组与普通数组都很相似。定型数组支持如下操作符、方法和属性：

   - `[]`
   - `copyWithin()`
   - `entries()`
   - `every()`
   - `fill()`
   - `filter()`
   - `find()`
   - `findIndex()`
   - `forEach()`
   - `indexOf()`
   - `join()`
   - `keys()`
   - `lastIndexOf()`
   - `length`
   - `map()`
   - `reduce()`
   - `reduceRight()`
   - `reverse()`
   - `slice()`
   - `some()`
   - `sort()`
   - `toLocaleString()`
   - `toString()`
   - `values()`

   其中，返回新数组的方法也会返回包含同样元素类型（element type）的新定型数组：

   ```
   const ints = new Int16Array([1, 2, 3]);
   const doubleints = ints.map(x => 2*x);
   alert(doubleints instanceof Int16Array); // true
   ```

   定型数组有一个`Symbol.iterator`符号属性，因此可以通过`for..of`循环和扩展操作符来操作：

```
const ints = new Int16Array([1, 2, 3]);
for (const int of ints) {
  alert(int);
}
// 1
// 2
// 3

alert(Math.max(...ints)); // 3
```

2. **合并、复制和修改定型数组**

定型数组同样使用数组缓冲来存储数据，而数组缓冲无法调整大小。因此，下列方法不适用于定型数组：

- concat()
- pop()
- push()
- shift()
- splice()
- unshift()

不过，定型数组也提供了两个新方法，可以快速向外或向内复制数据：set()和subarray()。

set()从提供的数组或定型数组中把值复制到当前定型数组中指定的索引位置：

```
// 创建长度为8的int16数组
const container = new Int16Array(8);
// 把定型数组复制为前4个值
// 偏移量默认为索引0
container.set(Int8Array.of(1, 2, 3, 4));
console.log(container);  // [1,2,3,4,0,0,0,0]
// 把普通数组复制为后4个值
// 偏移量4表示从索引4开始插入
container.set([5,6,7,8], 4);
console.log(container);  // [1,2,3,4,5,6,7,8]

// 溢出会抛出错误
container.set([5,6,7,8], 7);
// RangeError
```

subarray()执行与set()相反的操作，它会基于从原始定型数组中复制的值返回一个新定型数组。复制值时的开始索引和结束索引是可选的：

```
const source = Int16Array.of(2, 4, 6, 8);

// 把整个数组复制为一个同类型的新数组
```

```
const fullCopy = source.subarray();
console.log(fullCopy);  // [2, 4, 6, 8]

// 从索引2开始复制数组
const halfCopy = source.subarray(2);
console.log(halfCopy);  // [6, 8]

// 从索引1开始复制到索引3
const partialCopy = source.subarray(1, 3);
console.log(partialCopy);  // [4, 6]
```

定型数组没有原生的拼接能力，但使用定型数组API提供的很多工具可以手动构建：

```
// 第一个参数是应该返回的数组类型
// 其余参数是应该拼接在一起的定型数组
function typedArrayConcat(typedArrayConstructor, ...typedArrays) {
  // 计算所有数组中包含的元素总数
  const numElements = typedArrays.reduce((x,y) => (x.length || x) +
y.length);

  // 按照提供的类型创建一个数组，为所有元素留出空间
  const resultArray = new typedArrayConstructor(numElements);

  // 依次转移数组
  let currentOffset = 0;
  typedArrays.map(x => {
    resultArray.set(x, currentOffset);
    currentOffset += x.length;
  });

  return resultArray;
}

const concatArray = typedArrayConcat(Int32Array,
                                      Int8Array.of(1, 2, 3),
                                      Int16Array.of(4, 5, 6),
                                      Float32Array.of(7, 8, 9));
console.log(concatArray);  // [1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(concatArray instanceof Int32Array); // true
```

3. **下溢和上溢**

定型数组中值的下溢和上溢不会影响到其他索引，但仍然需要考虑数组的元素应该是什么类型。定型数组对于可以存储的每个索引只接受一个相关位，而不考虑它们对实际数值的影响。以下代码演示了如何处理下溢和上溢：

```
// 长度为2的有符号整数数组
// 每个索引保存一个二补数形式的有符号整数
// 范围是-128 (-1 * 2^7) ~127 (2^7 - 1)
```

```
const ints = new Int8Array(2);

// 长度为2的无符号整数数组
// 每个索引保存一个无符号整数
// 范围是0~255 (2^7 - 1)
const unsignedInts = new Uint8Array(2);

// 上溢的位不会影响相邻索引
// 索引只取最低有效位上的8位
unsignedInts[1] = 256;      // 0x100
console.log(unsignedInts);  // [0, 0]
unsignedInts[1] = 511;      // 0x1FF
console.log(unsignedInts);  // [0, 255]

// 下溢的位会被转换为其无符号的等价值
// 0xFF是以二补数形式表示的-1（截取到8位），
// 但255是一个无符号整数
unsignedInts[1] = -1        // 0xFF (truncated to 8 bits)
console.log(unsignedInts);  // [0, 255]

// 上溢自动变成二补数形式
// 0x80是无符号整数的128，是二补数形式的-128
ints[1] = 128;         // 0x80
console.log(ints);     // [0, -128]

// 下溢自动变成二补数形式
// 0xFF是无符号整数的255，是二补数形式的-1
ints[1] = 255;         // 0xFF
console.log(ints);     // [0, -1]
```

除了8种元素类型，还有一种“夹板”数组类型：`Uint8ClampedArray`，不允许任何方向溢出。超出最大值255的值会被向下舍入为255，而小于最小值0的值会被向上舍入为0。

```
const clampedInts = new Uint8ClampedArray([-1, 0, 255, 256]);
console.log(clampedInts); // [0, 0, 255, 255]
```

按照JavaScript之父Brendan Eich的说法：“`Uint8ClampedArray`完全是HTML5`canvas`元素的历史留存。除非真的做跟`canvas`相关的开发，否则不要使用它。”

6.4 Map

ECMAScript 6以前，在JavaScript中实现“键/值”式存储可以使用`Object`来方便高效地完成，也就是使用对象属性作为键，再使用属性来引用值。但这种实现并非没有问题，为此TC39委员会专门为“键/值”存储定义了一个规范。

作为ECMAScript 6的新增特性，`Map`是一种新的集合类型，为这门语言带来了真正的键/值存储机制。`Map`的大多数特性都可以通过`Object`类型实现，但二者之间还是存在一些细微的差异。具体实践中使用哪一个，还是值得细细甄别。

6.4.1 基本API

使用new关键字和Map构造函数可以创建一个空映射：

```
const m = new Map();
```

如果想在创建的同时初始化实例，可以给Map构造函数传入一个可迭代对象，需要包含键/值对数组。可迭代对象中的每个键/值对都会按照迭代顺序插入到新映射实例中：

```
// 使用嵌套数组初始化映射
const m1 = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);
alert(m1.size); // 3

// 使用自定义迭代器初始化映射
const m2 = new Map({
  [Symbol.iterator]: function*() {
    yield ["key1", "val1"];
    yield ["key2", "val2"];
    yield ["key3", "val3"];
  }
});
alert(m2.size); // 3

// 映射期待的键/值对，无论是否提供
const m3 = new Map([[]]);
alert(m3.has(undefined));  // true
alert(m3.get(undefined));  // undefined
```

初始化之后，可以使用set()方法再添加键/值对。另外，可以使用get()和has()进行查询，可以通过size属性获取映射中的键/值对的数量，还可以使用delete()和clear()删除值。

```
const m = new Map();

alert(m.has("firstName"));  // false
alert(m.get("firstName"));  // undefined
alert(m.size);              // 0

m.set("firstName", "Matt")
 .set("lastName", "Frisbie");

alert(m.has("firstName")); // true
alert(m.get("firstName")); // Matt
alert(m.size);             // 2
```

```
m.delete("firstName");     // 只删除这一个键/值对

alert(m.has("firstName")); // false
alert(m.has("lastName"));  // true
alert(m.size);             // 1

m.clear(); // 清除这个映射实例中的所有键/值对

alert(m.has("firstName")); // false
alert(m.has("lastName"));  // false
alert(m.size);             // 0
```

set()方法返回映射实例，因此可以把多个操作连缀起来，包括初始化声明：

```
const m = new Map().set("key1", "val1");

m.set("key2", "val2")
 .set("key3", "val3");

alert(m.size); // 3
```

与Object只能使用数值、字符串或符号作为键不同，Map可以使用任何JavaScript数据类型作为键。Map内部使用SameValueZero比较操作（ECMAScript规范内部定义，语言中不能使用），基本上相当于使用严格对象相等的标准来检查键的匹配性。与Object类似，映射的值是没有限制的。

```
const m = new Map();

const functionKey = function() {};
const symbolKey = Symbol();
const objectKey = new Object();

m.set(functionKey, "functionValue");
m.set(symbolKey, "symbolValue");
m.set(objectKey, "objectValue");

alert(m.get(functionKey));  // functionValue
alert(m.get(symbolKey));    // symbolValue
alert(m.get(objectKey));    // objectValue

// SameValueZero比较意味着独立实例不冲突
alert(m.get(function() {})); // undefined
```

与严格相等一样，在映射中用作键和值的对象及其他“集合”类型，在自己的内容或属性被修改时仍然保持不变：

```
const m = new Map();
```

```
const objKey = {},
      objVal = {},
      arrKey = [],
      arrVal = [];

m.set(objKey, objVal);
m.set(arrKey, arrVal);

objKey.foo = "foo";
objVal.bar = "bar";
arrKey.push("foo");
arrVal.push("bar");

console.log(m.get(objKey)); // {bar: "bar"}
console.log(m.get(arrKey)); // ["bar"]
```

SameValueZero比较也可能导致意想不到的冲突：

```
const m = new Map();

const a = 0/"", // NaN
      b = 0/"", // NaN
      pz = +0,
      nz = -0;

alert(a === b);   // false
alert(pz === nz); // true

m.set(a, "foo");
m.set(pz, "bar");

alert(m.get(b));  // foo
alert(m.get(nz)); // bar
```

> **注意** SameValueZero是ECMAScript规范新增的相等性比较算法。关于ECMAScript的相等性比较，可以参考MDN文档中的文章“Equality Comparisons and Sameness”。

6.4.2 顺序与迭代

与`Object`类型的一个主要差异是，`Map`实例会维护键值对的插入顺序，因此可以根据插入顺序执行迭代操作。

映射实例可以提供一个迭代器（`Iterator`），能以插入顺序生成`[key, value]`形式的数组。可以通过`entries()`方法（或者`Symbol.iterator`属性，它引用`entries()`）取得这个迭代器：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);
```

```
alert(m.entries === m[Symbol.iterator]); // true

for (let pair of m.entries()) {
  alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]

for (let pair of m[Symbol.iterator]()) {
  alert(pair);
}
// [key1,val1]
// [key2,val2]
// [key3,val3]
```

因为`entries()`是默认迭代器，所以可以直接对映射实例使用扩展操作，把映射转换为数组：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

console.log([...m]); // [[key1,val1],[key2,val2],[key3,val3]]
```

如果不使用迭代器，而是使用回调方式，则可以调用映射的`forEach()`方法并传入回调，依次迭代每个键/值对。传入的回调接收可选的第二个参数，这个参数用于重写回调内部`this`的值：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
]);

m.forEach((val, key) => alert(`${key} -> ${val}`));
// key1 -> val1
// key2 -> val2
// key3 -> val3
```

`keys()`和`values()`分别返回以插入顺序生成键和值的迭代器：

```
const m = new Map([
  ["key1", "val1"],
  ["key2", "val2"],
  ["key3", "val3"]
```

```
]);

for (let key of m.keys()) {
  alert(key);
}
// key1
// key2
// key3

for (let key of m.values()) {
  alert(key);
}
// value1
// value2
// value3
```

键和值在迭代器遍历时是可以修改的，但映射内部的引用则无法修改。当然，这并不妨碍修改作为键或值的对象内部的属性，因为这样并不影响它们在映射实例中的身份：

```
const m1 = new Map([
  ["key1", "val1"]
]);

// 作为键的字符串原始值是不能修改的
for (let key of m1.keys()) {
  key = "newKey";
  alert(key);              // newKey
  alert(m1.get("key1"));  // val1
}

const keyObj = {id: 1};

const m = new Map([
  [keyObj, "val1"]
]);

// 修改了作为键的对象的属性，但对象在映射内部仍然引用相同的值
for (let key of m.keys()) {
  key.id = "newKey";
  alert(key);             // {id: "newKey"}
  alert(m.get(keyObj));  // val1
}
alert(keyObj);           // {id: "newKey"}
```

6.4.3 选择Object还是Map

对于多数Web开发任务来说，选择Object还是Map只是个人偏好问题，影响不大。不过，对于在乎内存和性能的开发者来说，对象和映射之间确实存在显著的差别。

1. **内存占用**

Object和Map的工程级实现在不同浏览器间存在明显差异，但存储单个键/值对所占用的内存数量都会随键的数量线性增加。批量添加或删除键/值对则取决于各浏览器对该类型内存分配的工程实现。不同浏览器的情况不同，但给定固定大小的内存，Map大约可以比Object多存储50%的键/值对。

2. **插入性能**

   向Object和Map中插入新键/值对的消耗大致相当，不过插入Map在所有浏览器中一般会稍微快一点儿。对这两个类型来说，插入速度并不会随着键/值对数量而线性增加。如果代码涉及大量插入操作，那么显然Map的性能更佳。

3. **查找速度**

   与插入不同，从大型Object和Map中查找键/值对的性能差异极小，但如果只包含少量键/值对，则Object有时候速度更快。在把Object当成数组使用的情况下（比如使用连续整数作为属性），浏览器引擎可以进行优化，在内存中使用更高效的布局。这对Map来说是不可能的。对这两个类型而言，查找速度不会随着键/值对数量增加而线性增加。如果代码涉及大量查找操作，那么某些情况下可能选择Object更好一些。

4. **删除性能**

   使用delete删除Object属性的性能一直以来饱受诟病，目前在很多浏览器中仍然如此。为此，出现了一些伪删除对象属性的操作，包括把属性值设置为undefined或null。但很多时候，这都是一种讨厌的或不适宜的折中。而对大多数浏览器引擎来说，Map的delete()操作都比插入和查找更快。如果代码涉及大量删除操作，那么毫无疑问应该选择Map。

6.5 WeakMap

ECMAScript 6新增的“弱映射”（WeakMap）是一种新的集合类型，为这门语言带来了增强的键/值对存储机制。WeakMap是Map的“兄弟”类型，其API也是Map的子集。WeakMap中的“weak”（弱），描述的是JavaScript垃圾回收程序对待“弱映射”中键的方式。

6.5.1 基本API

可以使用new关键字实例化一个空的WeakMap：

```
const wm = new WeakMap();
```

弱映射中的键只能是Object或者继承自Object的类型，尝试使用非对象设置键会抛出TypeError。值的类型没有限制。

如果想在初始化时填充弱映射，则构造函数可以接收一个可迭代对象，其中需要包含键/值对数组。可迭代对象中的每个键/值都会按照迭代顺序插入新实例中：

```
const key1 = {id: 1},
      key2 = {id: 2},
      key3 = {id: 3};
// 使用嵌套数组初始化弱映射
const wm1 = new WeakMap([
```

```
  [key1, "val1"],
  [key2, "val2"],
  [key3, "val3"]
]);
alert(wm.get(key1)); // val2
alert(wm.get(key2)); // val2
alert(wm.get(key3)); // val3

// 初始化是全有或全无的操作
// 只要有一个键无效就会抛出错误，导致整个初始化失败
const wm2 = new WeakMap([
  [key1, "val1"],
  ["BADKEY", "val2"],
  [key3, "val3"]
]);
// TypeError: Invalid value used as WeakMap key
typeof wm2;
// ReferenceError: wm2 is not defined

// 原始值可以先包装成对象再用作键
const stringKey = new String("key1");
const wm3 = new WeakMap([
  stringKey, "val1"
]);
alert(wm3.get(stringKey)); // "val1"
```

初始化之后可以使用`set()`再添加键/值对，可以使用`get()`和`has()`查询，还可以使用`delete()`删除：

```
const wm = new WeakMap();

const key1 = {id: 1},
      key2 = {id: 2};

alert(wm.has(key1)); // false
alert(wm.get(key1)); // undefined

wm.set(key1, "Matt")
  .set(key2, "Frisbie");

alert(wm.has(key1)); // true
alert(wm.get(key1)); // Matt

wm.delete(key1);     // 只删除这一个键/值对

alert(wm.has(key1)); // false
alert(wm.has(key2)); // true
```

`set()`方法返回弱映射实例，因此可以把多个操作连缀起来，包括初始化声明：

```
const key1 = {id: 1},
      key2 = {id: 2},
      key3 = {id: 3};

const wm = new WeakMap().set(key1, "val1");

wm.set(key2, "val2")
  .set(key3, "val3");

alert(wm.get(key1)); // val1
alert(wm.get(key2)); // val2
alert(wm.get(key3)); // val3
```

6.5.2 弱键

WeakMap中“weak”表示弱映射的键是“弱弱地拿着”的。意思就是，这些键不属于正式的引用，不会阻止垃圾回收。但要注意的是，弱映射中值的引用可**不是**“弱弱地拿着”的。只要键存在，键/值对就会存在于映射中，并被当作对值的引用，因此就不会被当作垃圾回收。

来看下面的例子：

```
const wm = new WeakMap();

wm.set({}, "val");
```

set()方法初始化了一个新对象并将它用作一个字符串的键。因为没有指向这个对象的其他引用，所以当这行代码执行完成后，这个对象键就会被当作垃圾回收。然后，这个键/值对就从弱映射中消失了，使其成为一个空映射。在这个例子中，因为值也没有被引用，所以这对键/值被破坏以后，值本身也会成为垃圾回收的目标。

再看一个稍微不同的例子：

```
const wm = new WeakMap();

const container = {
  key: {}
};

wm.set(container.key, "val");

function removeReference() {
  container.key = null;
}
```

这一次，container对象维护着一个对弱映射键的引用，因此这个对象键不会成为垃圾回收的目标。不过，如果调用了removeReference()，就会摧毁键对象的最后一个引用，垃圾回收程序就可以把这个键/值对清理掉。

6.5.3 不可迭代键

因为WeakMap中的键/值对任何时候都可能被销毁，所以没必要提供迭代其键/值对的能力。当然，也用不着像clear()这样一次性销毁所有键/值的方法。WeakMap 确实没有这个方法。因为不可能迭代，所以也不可能在不知道对象引用的情况下从弱映射中取得值。即便代码可以访问WeakMap实例，也没办法看到其中的内容。

WeakMap实例之所以限制只能用对象作为键，是为了保证只有通过键对象的引用才能取得值。如果允许原始值，那就没办法区分初始化时使用的字符串字面量和初始化之后使用的一个相等的字符串了。

6.5.4 使用弱映射

WeakMap实例与现有JavaScript对象有着很大不同，可能一时不容易说清楚应该怎么使用它。这个问题没有唯一的答案，但已经出现了很多相关策略。

1. **私有变量**

   弱映射造就了在JavaScript中实现真正私有变量的一种新方式。前提很明确：私有变量会存储在弱映射中，以对象实例为键，以私有成员的字典为值。

   下面是一个示例实现：

```
const wm = new WeakMap();

class User {
  constructor(id) {
    this.idProperty = Symbol('id');
    this.setId(id);
  }

  setPrivate(property, value) {
    const privateMembers = wm.get(this) || {};
    privateMembers[property] = value;
    wm.set(this, privateMembers);
  }

  getPrivate(property) {
    return wm.get(this)[property];
  }

  setId(id) {
    this.setPrivate(this.idProperty, id);
  }

  getId() {
    return this.getPrivate(this.idProperty);
  }
}

const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456
```

```
// 并不是真正私有的
alert(wm.get(user)[user.idProperty]); // 456
```

慧眼独具的读者会发现，对于上面的实现，外部代码只需要拿到对象实例的引用和弱映射，就可以取得“私有”变量了。为了避免这种访问，可以用一个闭包把WeakMap包装起来，这样就可以把弱映射与外界完全隔离开了：

```
const User = (() => {
  const wm = new WeakMap();

  class User {
    constructor(id) {
      this.idProperty = Symbol('id');
      this.setId(id);
    }

    setPrivate(property, value) {
      const privateMembers = wm.get(this) || {};
      privateMembers[property] = value;
      wm.set(this, privateMembers);
    }

    getPrivate(property) {
      return wm.get(this)[property];
    }

    setId(id) {
      this.setPrivate(this.idProperty, id);
    }

    getId(id) {
      return this.getPrivate(this.idProperty);
    }
  }
  return User;
})();

const user = new User(123);
alert(user.getId()); // 123
user.setId(456);
alert(user.getId()); // 456
```

这样，拿不到弱映射中的健，也就无法取得弱映射中对应的值。虽然这防止了前面提到的访问，但整个代码也完全陷入了ES6之前的闭包私有变量模式。

2. **DOM节点元数据**

因为WeakMap实例不会妨碍垃圾回收，所以非常适合保存关联元数据。来看下面这个例子，其中使用了常规的Map：

```
const m = new Map();

const loginButton = document.querySelector('#login');

// 给这个节点关联一些元数据
m.set(loginButton, {disabled: true});
```

假设在上面的代码执行后，页面被JavaScript改变了，原来的登录按钮从DOM树中被删掉了。但由于映射中还保存着按钮的引用，所以对应的DOM节点仍然会逗留在内存中，除非明确将其从映射中删除或者等到映射本身被销毁。

如果这里使用的是弱映射，如以下代码所示，那么当节点从DOM树中被删除后，垃圾回收程序就可以立即释放其内存（假设没有其他地方引用这个对象）：

```
const wm = new WeakMap();

const loginButton = document.querySelector('#login');

// 给这个节点关联一些元数据
wm.set(loginButton, {disabled: true});
```

6.6 Set

ECMAScript 6新增的Set是一种新集合类型，为这门语言带来集合数据结构。Set在很多方面都像是加强的Map，这是因为它们的大多数API和行为都是共有的。

6.6.1 基本API

使用new关键字和Set构造函数可以创建一个空集合：

```
const m = new Set();
```

如果想在创建的同时初始化实例，则可以给Set构造函数传入一个可迭代对象，其中需要包含插入到新集合实例中的元素：

```
// 使用数组初始化集合
const s1 = new Set(["val1", "val2", "val3"]);

alert(s1.size); // 3

// 使用自定义迭代器初始化集合
const s2 = new Set({
  [Symbol.iterator]: function*() {
    yield "val1";
    yield "val2";
```

```
    yield "val3";
  }
});
alert(s2.size); // 3
```

初始化之后，可以使用add()增加值，使用has()查询，通过size取得元素数量，以及使用delete()和clear()删除元素：

```
const s = new Set();

alert(s.has("Matt"));    // false
alert(s.size);           // 0

s.add("Matt")
 .add("Frisbie");

alert(s.has("Matt"));    // true
alert(s.size);           // 2

s.delete("Matt");

alert(s.has("Matt"));    // false
alert(s.has("Frisbie")); // true
alert(s.size);           // 1

s.clear(); // 销毁集合实例中的所有值

alert(s.has("Matt"));    // false
alert(s.has("Frisbie")); // false
alert(s.size);           // 0
```

add()返回集合的实例，所以可以将多个添加操作连缀起来，包括初始化：

```
const s = new Set().add("val1");

s.add("val2")
 .add("val3");

alert(s.size); // 3
```

与Map类似，Set可以包含任何JavaScript数据类型作为值。集合也使用SameValueZero操作（ECMAScript内部定义，无法在语言中使用），基本上相当于使用严格对象相等的标准来检查值的匹配性。

```
const s = new Set();

const functionVal = function() {};
const symbolVal = Symbol();
```

```
const objectVal = new Object();

s.add(functionVal);
s.add(symbolVal);
s.add(objectVal);

alert(s.has(functionVal));   // true
alert(s.has(symbolVal));     // true
alert(s.has(objectVal));     // true

// SameValueZero检查意味着独立的实例不会冲突
alert(s.has(function() {})); // false
```

与严格相等一样，用作值的对象和其他"集合"类型在自己的内容或属性被修改时也不会改变：

```
const s = new Set();

const objVal = {},
      arrVal = [];

s.add(objVal);
s.add(arrVal);

objVal.bar = "bar";
arrVal.push("bar");

alert(s.has(objVal)); // true
alert(s.has(arrVal)); // true
```

`add()`和`delete()`操作是幂等的。`delete()`返回一个布尔值，表示集合中是否存在要删除的值：

```
const s = new Set();

s.add('foo');
alert(s.size); // 1
s.add('foo');
alert(s.size); // 1

// 集合里有这个值
alert(s.delete('foo')); // true

// 集合里没有这个值
alert(s.delete('foo')); // false
```

6.6.2 顺序与迭代

`Set`会维护值插入时的顺序，因此支持按顺序迭代。

集合实例可以提供一个迭代器（Iterator），能以插入顺序生成集合内容。可以通过values()方法及其别名方法keys()（或者Symbol.iterator属性，它引用values()）取得这个迭代器：

```
const s = new Set(["val1", "val2", "val3"]);

alert(s.values === s[Symbol.iterator]); // true
alert(s.keys === s[Symbol.iterator]);   // true

for (let value of s.values()) {
  alert(value);
}
// val1
// val2
// val3

for (let value of s[Symbol.iterator]()) {
  alert(value);
}
// val1
// val2
// val3
```

因为values()是默认迭代器，所以可以直接对集合实例使用扩展操作，把集合转换为数组：

```
const s = new Set(["val1", "val2", "val3"]);

console.log([...s]); // ["val1", "val2", "val3"]
```

集合的entries()方法返回一个迭代器，可以按照插入顺序产生包含两个元素的数组，这两个元素是集合中每个值的重复出现：

```
const s = new Set(["val1", "val2", "val3"]);

for (let pair of s.entries()) {
  console.log(pair);
}
// ["val1", "val1"]
// ["val2", "val2"]
// ["val3", "val3"]
```

如果不使用迭代器，而是使用回调方式，则可以调用集合的forEach()方法并传入回调，依次迭代每个键/值对。传入的回调接收可选的第二个参数，这个参数用于重写回调内部this的值：

```
const s = new Set(["val1", "val2", "val3"]);

s.forEach((val, dupVal) => alert(`${val} -> ${dupVal}`));
```

```
// val1 -> val1
// val2 -> val2
// val3 -> val3
```

修改集合中值的属性不会影响其作为集合值的身份：

```
const s1 = new Set(["val1"]);

// 字符串原始值作为值不会被修改
for (let value of s1.values()) {
  value = "newVal";
  alert(value);           // newVal
  alert(s1.has("val1")); // true
}

const valObj = {id: 1};

const s2 = new Set([valObj]);

// 修改值对象的属性，但对象仍然存在于集合中
for (let value of s2.values()) {
  value.id = "newVal";
  alert(value);           // {id: "newVal"}
  alert(s2.has(valObj));  // true
}
alert(valObj);            // {id: "newVal"}
```

6.6.3 定义正式集合操作

从各方面来看，`Set`跟`Map`都很相似，只是API稍有调整。唯一需要强调的就是集合的API只支持自引用操作。很多开发者都喜欢使用`Set`操作，但需要手动实现：或者是子类化`Set`，或者是定义一个实用函数库。要把两种方式合二为一，可以在子类上实现静态方法，然后在实例方法中使用这些静态方法。在实现这些操作时，需要考虑几个地方。

- 某些`Set`操作是有关联性的，因此最好让实现的方法能支持处理任意多个集合实例。
- `Set`保留插入顺序，所有方法返回的集合必须保证顺序。
- 尽可能高效地使用内存。扩展操作符的语法很简洁，但尽可能避免集合和数组间的相互转换能够节省对象初始化成本。
- 不要修改已有的集合实例。`union(a, b)`或`a.union(b)`应该返回包含结果的新集合实例。

```
class XSet extends Set {
  union(...sets) {
    return XSet.union(this, ...sets)
  }

  intersection(...sets) {
    return XSet.intersection(this, ...sets);
  }
```

```
  difference(set) {
    return XSet.difference(this, set);
  }

  symmetricDifference(set) {
    return XSet.symmetricDifference(this, set);
  }

  cartesianProduct(set) {
    return XSet.cartesianProduct(this, set);
  }

  powerSet() {
    return XSet.powerSet(this);
  }

  // 返回两个或更多集合的并集
  static union(a, ...bSets) {
    const unionSet = new XSet(a);
    for (const b of bSets) {
      for (const bValue of b) {
        unionSet.add(bValue);
      }
    }
    return unionSet;
  }

  // 返回两个或更多集合的交集
  static intersection(a, ...bSets) {
    const intersectionSet = new XSet(a);
    for (const aValue of intersectionSet) {
      for (const b of bSets) {
        if (!b.has(aValue)) {
          intersectionSet.delete(aValue);
        }
      }
    }
    return intersectionSet;
  }

  // 返回两个集合的差集
  static difference(a, b) {
    const differenceSet = new XSet(a);
    for (const bValue of b) {
      if (a.has(bValue)) {
        differenceSet.delete(bValue);
      }
    }
    return differenceSet;
  }

  // 返回两个集合的对称差集
  static symmetricDifference(a, b) {
```

```
    // 按照定义，对称差集可以表达为
    return a.union(b).difference(a.intersection(b));
  }

  // 返回两个集合（数组对形式）的笛卡儿积
  // 必须返回数组集合，因为笛卡儿积可能包含相同值的对
  static cartesianProduct(a, b) {
    const cartesianProductSet = new XSet();
    for (const aValue of a) {
      for (const bValue of b) {
        cartesianProductSet.add([aValue, bValue]);
      }
    }
    return cartesianProductSet;
  }

  // 返回一个集合的幂集
  static powerSet(a) {
    const powerSet = new XSet().add(new XSet());
    for (const aValue of a) {
      for (const set of new XSet(powerSet)) {
        powerSet.add(new XSet(set).add(aValue));
      }
    }
    return powerSet;
  }
}
```

6.7 WeakSet

ECMAScript 6新增的“弱集合”（WeakSet）是一种新的集合类型，为这门语言带来了集合数据结构。WeakSet是Set的“兄弟”类型，其API也是Set的子集。WeakSet中的“weak”（弱），描述的是JavaScript垃圾回收程序对待“弱集合”中值的方式。

6.7.1 基本API

可以使用new关键字实例化一个空的WeakSet：

```
const ws = new WeakSet();
```

弱集合中的值只能是Object或者继承自Object的类型，尝试使用非对象设置值会抛出TypeError。

如果想在初始化时填充弱集合，则构造函数可以接收一个可迭代对象，其中需要包含有效的值。可迭代对象中的每个值都会按照迭代顺序插入到新实例中：

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};
// 使用数组初始化弱集合
```

```
const ws1 = new WeakSet([val1, val2, val3]);

alert(ws1.has(val1)); // true
alert(ws1.has(val2)); // true
alert(ws1.has(val3)); // true

// 初始化是全有或全无的操作
// 只要有一个值无效就会抛出错误，导致整个初始化失败
const ws2 = new WeakSet([val1, "BADVAL", val3]);
// TypeError: Invalid value used in WeakSet
typeof ws2;
// ReferenceError: ws2 is not defined

// 原始值可以先包装成对象再用作值
const stringVal = new String("val1");
const ws3 = new WeakSet([stringVal]);
alert(ws3.has(stringVal)); // true
```

初始化之后可以使用`add()`再添加新值，可以使用`has()`查询，还可以使用`delete()`删除：

```
const ws = new WeakSet();

const val1 = {id: 1},
      val2 = {id: 2};

alert(ws.has(val1)); // false

ws.add(val1)
  .add(val2);

alert(ws.has(val1)); // true
alert(ws.has(val2)); // true

ws.delete(val1);     // 只删除这一个值

alert(ws.has(val1)); // false
alert(ws.has(val2)); // true
```

`add()`方法返回弱集合实例，因此可以把多个操作连缀起来，包括初始化声明：

```
const val1 = {id: 1},
      val2 = {id: 2},
      val3 = {id: 3};

const ws = new WeakSet().add(val1);

ws.add(val2)
  .add(val3);
```

```
alert(ws.has(val1)); // true
alert(ws.has(val2)); // true
alert(ws.has(val3)); // true
```

6.7.2 弱值

WeakSet中"weak"表示弱集合的值是"弱弱地拿着"的。意思就是，这些值不属于正式的引用，不会阻止垃圾回收。

来看下面的例子：

```
const ws = new WeakSet();

ws.add({});
```

add()方法初始化了一个新对象，并将它用作一个值。因为没有指向这个对象的其他引用，所以当这行代码执行完成后，这个对象值就会被当作垃圾回收。然后，这个值就从弱集合中消失了，使其成为一个空集合。

再看一个稍微不同的例子：

```
const ws = new WeakSet();

const container = {
  val: {}
};

ws.add(container.val);

function removeReference() {
  container.val = null;
}
```

这一次，container对象维护着一个对弱集合值的引用，因此这个对象值不会成为垃圾回收的目标。不过，如果调用了removeReference()，就会摧毁值对象的最后一个引用，垃圾回收程序就可以把这个值清理掉。

6.7.3 不可迭代值

因为WeakSet中的值任何时候都可能被销毁，所以没必要提供迭代其值的能力。当然，也用不着像clear()这样一次性销毁所有值的方法。WeakSet 确实没有这个方法。因为不可能迭代，所以也不可能在不知道对象引用的情况下从弱集合中取得值。即便代码可以访问WeakSet实例，也没办法看到其中的内容。

WeakSet之所以限制只能用对象作为值，是为了保证只有通过值对象的引用才能取得值。如果允许原始值，那就没办法区分初始化时使用的字符串字面量和初始化之后使用的一个相等的字符串了。

6.7.4 使用弱集合

相比于WeakMap实例，WeakSet实例的用处没有那么大。不过，弱集合在给对象打标签时还是有价值的。

来看下面的例子，这里使用了一个普通Set：

```
const disabledElements = new Set();

const loginButton = document.querySelector('#login');

// 通过加入对应集合，给这个节点打上“禁用”标签
disabledElements.add(loginButton);
```

这样，通过查询元素在不在disabledElements中，就可以知道它是不是被禁用了。不过，假如元素从DOM树中被删除了，它的引用却仍然保存在Set中，因此垃圾回收程序也不能回收它。

为了让垃圾回收程序回收元素的内存，可以在这里使用WeakSet：

```
const disabledElements = new WeakSet();

const loginButton = document.querySelector('#login');

// 通过加入对应集合，给这个节点打上“禁用”标签
disabledElements.add(loginButton);
```

这样，只要WeakSet中任何元素从DOM树中被删除，垃圾回收程序就可以忽略其存在，而立即释放其内存（假设没有其他地方引用这个对象）。

6.8 迭代与扩展操作

ECMAScript 6新增的迭代器和扩展操作符对集合引用类型特别有用。这些新特性让集合类型之间相互操作、复制和修改变得异常方便。

> **注意** 第7章会更详细地介绍迭代器和生成器。

如本章前面所示，有4种原生集合类型定义了默认迭代器：

- Array
- 所有定型数组
- Map
- Set

很简单，这意味着上述所有类型都支持顺序迭代，都可以传入for-of循环：

```
let iterableThings = [
  Array.of(1, 2),
  typedArr = Int16Array.of(3, 4),
  new Map([[5, 6], [7, 8]]),
  new Set([9, 10])
];

for (const iterableThing of iterableThings) {
```

```
  for (const x of iterableThing) {
    console.log(x);
  }
}

// 1
// 2
// 3
// 4
// [5, 6]
// [7, 8]
// 9
// 10
```

这也意味着所有这些类型都兼容扩展操作符。扩展操作符在对可迭代对象执行浅复制时特别有用，只需简单的语法就可以复制整个对象：

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1];

console.log(arr1);          // [1, 2, 3]
console.log(arr2);          // [1, 2, 3]
console.log(arr1 === arr2); // false
```

对于期待可迭代对象的构造函数，只要传入一个可迭代对象就可以实现复制：

```
let map1 = new Map([[1, 2], [3, 4]]);
let map2 = new Map(map1);

console.log(map1); // Map {1 => 2, 3 => 4}
console.log(map2); // Map {1 => 2, 3 => 4}
```

当然，也可以构建数组的部分元素：

```
let arr1 = [1, 2, 3];
let arr2 = [0, ...arr1, 4, 5];

console.log(arr2); // [0, 1, 2, 3, 4, 5]
```

浅复制意味着只会复制对象引用：

```
let arr1 = [{}];
let arr2 = [...arr1];
```

```
arr1[0].foo = 'bar';
console.log(arr2[0]); // { foo: 'bar' }
```

上面的这些类型都支持多种构建方法，比如`Array.of()`和`Array.from()`静态方法。在与扩展操作符一起使用时，可以非常方便地实现互操作：

```
let arr1 = [1, 2, 3];

// 把数组复制到定型数组
let typedArr1 = Int16Array.of(...arr1);
let typedArr2 = Int16Array.from(arr1);
console.log(typedArr1);   // Int16Array [1, 2, 3]
console.log(typedArr2);   // Int16Array [1, 2, 3]

// 把数组复制到映射
let map = new Map(arr1.map((x) => [x, 'val' + x]));
console.log(map);   // Map {1 => 'val 1', 2 => 'val 2', 3 => 'val 3'}

// 把数组复制到集合
let set = new Set(typedArr2);
console.log(set);   // Set {1, 2, 3}

// 把集合复制回数组
let arr2 = [...set];
console.log(arr2);  // [1, 2, 3]
```

6.9 小结

JavaScript中的对象是引用值，可以通过几种内置引用类型创建特定类型的对象。

- 引用类型与传统面向对象编程语言中的类相似，但实现不同。
- `Object`类型是一个基础类型，所有引用类型都从它继承了基本的行为。
- `Array`类型表示一组有序的值，并提供了操作和转换值的能力。
- 定型数组包含一套不同的引用类型，用于管理数值在内存中的类型。
- `Date`类型提供了关于日期和时间的信息，包括当前日期和时间以及计算。
- `RegExp`类型是ECMAScript支持的正则表达式的接口，提供了大多数基本正则表达式以及一些高级正则表达式的能力。

JavaScript比较独特的一点是，函数其实是`Function`类型的实例，这意味着函数也是对象。由于函数是对象，因此也就具有能够增强自身行为的方法。

因为原始值包装类型的存在，所以JavaScript中的原始值可以拥有类似对象的行为。有3种原始值包装类型：`Boolean`、`Number`和`String`。它们都具有如下特点。

- 每种包装类型都映射到同名的原始类型。
- 在以读模式访问原始值时，后台会实例化一个原始值包装对象，通过这个对象可以操作数据。
- 涉及原始值的语句只要一执行完毕，包装对象就会立即销毁。

JavaScript还有两个在一开始执行代码时就存在的内置对象：Global和Math。其中，Global对象可以在大多数ECMAScript实现中访问。不过浏览器将Global实现为window对象。所有全局变量和函数都是Global对象的属性。Math对象包含辅助完成复杂数学计算的属性和方法。

ECMAScript 6新增了一批引用类型：Map、WeakMap、Set和WeakSet。这些类型为组织应用程序数据和简化内存管理提供了新能力。

第 7 章　迭代器与生成器

> **本章内容**
>
> - 理解迭代
> - 迭代器模式
> - 生成器

迭代的英文“iteration”源自拉丁文itero，意思是“重复”或“再来”。在软件开发领域，“迭代”的意思是按照顺序反复多次执行一段程序，通常会有明确的终止条件。ECMAScript 6规范新增了两个高级特性：迭代器和生成器。使用这两个特性，能够更清晰、高效、方便地实现迭代。

7.1　理解迭代

在JavaScript中，计数循环就是一种最简单的迭代：

```
for (let i = 1; i <= 10; ++i) {
  console.log(i);
}
```

循环是迭代机制的基础，这是因为它可以指定迭代的次数，以及每次迭代要执行什么操作。每次循环都会在下一次迭代开始之前完成，而每次迭代的顺序都是事先定义好的。

迭代会在一个有序集合上进行。（“有序”可以理解为集合中所有项都可以按照既定的顺序被遍历到，特别是开始和结束项有明确的定义。）数组是JavaScript中有序集合的最典型例子。

```
let collection = ['foo', 'bar', 'baz'];

for (let index = 0; index < collection.length; ++index) {
  console.log(collection[index]);
}
```

因为数组有已知的长度，且数组每一项都可以通过索引获取，所以整个数组可以通过递增索引来遍历。

由于如下原因，通过这种循环来执行例程并不理想。

- **迭代之前需要事先知道如何使用数据结构。**数组中的每一项都只能先通过引用取得数组对象，然后再通过[]操作符取得特定索引位置上的项。这种情况并不适用于所有数据结构。
- **遍历顺序并不是数据结构固有的。**通过递增索引来访问数据是特定于数组类型的方式，并不适用于其他具有隐式顺序的数据结构。

ES5新增了`Array.prototype.forEach()`方法，向通用迭代需求迈进了一步（但仍然不够理想）：

```
let collection = ['foo', 'bar', 'baz'];

collection.forEach((item) => console.log(item));
// foo
// bar
// baz
```

这个方法解决了单独记录索引和通过数组对象取得值的问题。不过，没有办法标识迭代何时终止。因此这个方法只适用于数组，而且回调结构也比较笨拙。

在ECMAScript较早的版本中，执行迭代必须使用循环或其他辅助结构。随着代码量增加，代码会变得越发混乱。很多语言都通过原生语言结构解决了这个问题，开发者无须事先知道如何迭代就能实现迭代操作。这个解决方案就是**迭代器模式**。Python、Java、C++，还有其他很多语言都对这个模式提供了完备的支持。JavaScript在ECMAScript 6以后也支持了迭代器模式。

7.2 迭代器模式

迭代器模式（特别是在ECMAScript这个语境下）描述了一个方案，即可以把有些结构称为“可迭代对象”（iterable），因为它们实现了正式的`Iterable`接口，而且可以通过迭代器`Iterator`消费。

可迭代对象是一种抽象的说法。基本上，可以把可迭代对象理解成数组或集合这样的集合类型的对象。它们包含的元素都是有限的，而且都具有无歧义的遍历顺序：

```
// 数组的元素是有限的
// 递增索引可以按序访问每个元素
let arr = [3, 1, 4];

// 集合的元素是有限的
// 可以按插入顺序访问每个元素
let set = new Set().add(3).add(1).add(4);
```

不过，可迭代对象不一定是集合对象，也可以是仅仅具有类似数组行为的其他数据结构，比如本章开头提到的计数循环。该循环中生成的值是暂时性的，但循环本身是在执行迭代。计数循环和数组都具有可迭代对象的行为。

> **注意** 临时性可迭代对象可以实现为生成器，本章后面会讨论。

任何实现`Iterable`接口的数据结构都可以被实现`Iterator`接口的结构“消费”（consume）。**迭代器**（iterator）是按需创建的一次性对象。每个迭代器都会关联一个**可迭代对象**，而迭代器会暴露迭代其关联可迭代对象的API。迭代器无须了解与其关联的可迭代对象的结构，只需要知道如何取得连续的值。这种概念上的分离正是`Iterable`和`Iterator`的强大之处。

7.2.1 可迭代协议

实现`Iterable`接口（可迭代协议）要求同时具备两种能力：支持迭代的自我识别能力和创建实现`Iterator`接口的对象的能力。在ECMAScript中，这意味着必须暴露一个属性作为“默认迭代器”，而且这个属性必须使用特

殊的`Symbol.iterator`作为键。这个默认迭代器属性必须引用一个迭代器工厂函数，调用这个工厂函数必须返回一个新迭代器。

很多内置类型都实现了`Iterable`接口：

- 字符串
- 数组
- 映射
- 集合
- `arguments`对象
- `NodeList`等DOM集合类型

检查是否存在默认迭代器属性可以暴露这个工厂函数：

```
let num = 1;
let obj = {};

// 这两种类型没有实现迭代器工厂函数
console.log(num[Symbol.iterator]); // undefined
console.log(obj[Symbol.iterator]); // undefined

let str = 'abc';
let arr = ['a', 'b', 'c'];
let map = new Map().set('a', 1).set('b', 2).set('c', 3);
let set = new Set().add('a').add('b').add('c');
let els = document.querySelectorAll('div');

// 这些类型都实现了迭代器工厂函数
console.log(str[Symbol.iterator]); // f values() { [native code] }
console.log(arr[Symbol.iterator]); // f values() { [native code] }
console.log(map[Symbol.iterator]); // f values() { [native code] }
console.log(set[Symbol.iterator]); // f values() { [native code] }
console.log(els[Symbol.iterator]); // f values() { [native code] }

// 调用这个工厂函数会生成一个迭代器
console.log(str[Symbol.iterator]()); // StringIterator {}
console.log(arr[Symbol.iterator]()); // ArrayIterator {}
console.log(map[Symbol.iterator]()); // MapIterator {}
console.log(set[Symbol.iterator]()); // SetIterator {}
console.log(els[Symbol.iterator]()); // ArrayIterator {}
```

实际写代码过程中，不需要显式调用这个工厂函数来生成迭代器。实现可迭代协议的所有类型都会自动兼容接收可迭代对象的任何语言特性。接收可迭代对象的原生语言特性包括：

- `for-of`循环
- 数组解构
- 扩展操作符
- `Array.from()`
- 创建集合
- 创建映射

- `Promise.all()`接收由期约组成的可迭代对象
- `Promise.race()`接收由期约组成的可迭代对象
- `yield*`操作符，在生成器中使用

这些原生语言结构会在后台调用提供的可迭代对象的这个工厂函数，从而创建一个迭代器：

```
let arr = ['foo', 'bar', 'baz'];

// for-of循环
for (let el of arr) {
  console.log(el);
}
// foo
// bar
// baz

// 数组解构
let [a, b, c] = arr;
console.log(a, b, c); // foo, bar, baz

// 扩展操作符
let arr2 = [...arr];
console.log(arr2); // ['foo', 'bar', 'baz']

// Array.from()
let arr3 = Array.from(arr);
console.log(arr3); // ['foo', 'bar', 'baz']

// Set构造函数
let set = new Set(arr);
console.log(set); // Set(3) {'foo', 'bar', 'baz'}

// Map构造函数
let pairs = arr.map((x, i) => [x, i]);
console.log(pairs); // [['foo', 0], ['bar', 1], ['baz', 2]]
let map = new Map(pairs);
console.log(map); // Map(3) { 'foo'=>0, 'bar'=>1, 'baz'=>2 }
```

如果对象原型链上的父类实现了`Iterable`接口，那这个对象也就实现了这个接口：

```
class FooArray extends Array {}
let fooArr = new FooArray('foo', 'bar', 'baz');

for (let el of fooArr) {
  console.log(el);
}
// foo
// bar
// baz
```

7.2.2 迭代器协议

迭代器是一种一次性使用的对象，用于迭代与其关联的可迭代对象。迭代器API使用`next()`方法在可迭代对象中遍历数据。每次成功调用`next()`，都会返回一个`IteratorResult`对象，其中包含迭代器返回的下一个值。若不调用`next()`，则无法知道迭代器的当前位置。

`next()`方法返回的迭代器对象`IteratorReault`包含两个属性：`done`和`value`。`done`是一个布尔值，表示是否还可以再次调用`next()`取得下一个值；`value`包含可迭代对象的下一个值（`done`为`false`），或者`undefined`（`done`为`true`）。`done: true`状态称为“耗尽”。可以通过以下简单的数组来演示：

```
// 可迭代对象
let arr = ['foo', 'bar'];

// 迭代器工厂函数
console.log(arr[Symbol.iterator]); // f values() { [native code] }

// 迭代器
let iter = arr[Symbol.iterator]();
console.log(iter); // ArrayIterator {}

// 执行迭代
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: true, value: undefined }
```

这里通过创建迭代器并调用`next()`方法按顺序迭代了数组，直至不再产生新值。迭代器并不知道怎么从可迭代对象中取得下一个值，也不知道可迭代对象有多大。只要迭代器到达`done: true`状态，后续调用`next()`就一直返回同样的值了：

```
let arr = ['foo'];
let iter = arr[Symbol.iterator]();
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
```

每个迭代器都表示对可迭代对象的一次性有序遍历。不同迭代器的实例相互之间没有联系，只会独立地遍历可迭代对象：

```
let arr = ['foo', 'bar'];
let iter1 = arr[Symbol.iterator]();
let iter2 = arr[Symbol.iterator]();

console.log(iter1.next()); // { done: false, value: 'foo' }
console.log(iter2.next()); // { done: false, value: 'foo' }
console.log(iter2.next()); // { done: false, value: 'bar' }
console.log(iter1.next()); // { done: false, value: 'bar' }
```

迭代器并不与可迭代对象某个时刻的快照绑定，而仅仅是使用游标来记录遍历可迭代对象的历程。如果可迭代对象在迭代期间被修改了，那么迭代器也会反映相应的变化：

```
let arr = ['foo', 'baz'];
let iter = arr[Symbol.iterator]();

console.log(iter.next()); // { done: false, value: 'foo' }

// 在数组中间插入值
arr.splice(1, 0, 'bar');

console.log(iter.next()); // { done: false, value: 'bar' }
console.log(iter.next()); // { done: false, value: 'baz' }
console.log(iter.next()); // { done: true, value: undefined }
```

> **注意** 迭代器维护着一个指向可迭代对象的引用，因此迭代器会阻止垃圾回收程序回收可迭代对象。

“迭代器”的概念有时候容易模糊，因为它可以指通用的迭代，也可以指接口，还可以指正式的迭代器类型。下面的例子比较了一个显式的迭代器实现和一个原生的迭代器实现。

```
// 这个类实现了可迭代接口（Iterable）
// 调用默认的迭代器工厂函数会返回
// 一个实现迭代器接口（Iterator）的迭代器对象
class Foo {
  [Symbol.iterator]() {
    return {
      next() {
        return { done: false, value: 'foo' };
      }
    }
  }
}
let f = new Foo();

// 打印出实现了迭代器接口的对象
console.log(f[Symbol.iterator]()); // { next: f() {} }

// Array类型实现了可迭代接口（Iterable）
// 调用Array类型的默认迭代器工厂函数
// 会创建一个ArrayIterator的实例
let a = new Array();

// 打印出ArrayIterator的实例
console.log(a[Symbol.iterator]()); // Array Iterator {}
```

7.2.3 自定义迭代器

与Iterable接口类似，任何实现Iterator接口的对象都可以作为迭代器使用。下面这个例子中的Counter类只能被迭代一定的次数：

```
class Counter {
  // Counter的实例应该迭代limit次
  constructor(limit) {
    this.count = 1;
    this.limit = limit;
  }

  next() {
    if (this.count <= this.limit) {
      return { done: false, value: this.count++ };
    } else {
      return { done: true, value: undefined };
    }
  }
  [Symbol.iterator]() {
    return this;
  }
}

let counter = new Counter(3);

for (let i of counter) {
  console.log(i);
}
// 1
// 2
// 3
```

这个类实现了Iterator接口，但不理想。这是因为它的每个实例只能被迭代一次：

```
for (let i of counter) { console.log(i); }
// 1
// 2
// 3

for (let i of counter) { console.log(i); }
// (nothing logged)
```

为了让一个可迭代对象能够创建多个迭代器，必须每创建一个迭代器就对应一个新计数器。为此，可以把计数器变量放到闭包里，然后通过闭包返回迭代器：

```
class Counter {
  constructor(limit) {
    this.limit = limit;
  }
```

```
  [Symbol.iterator]() {
    let count = 1,
        limit = this.limit;
    return {
      next() {
        if (count <= limit) {
          return { done: false, value: count++ };
        } else {
          return { done: true, value: undefined };
        }
      }
    };
  }
}

let counter = new Counter(3);

for (let i of counter) { console.log(i); }
// 1
// 2
// 3

for (let i of counter) { console.log(i); }
// 1
// 2
// 3
```

每个以这种方式创建的迭代器也实现了`Iterable`接口。`Symbol.iterator`属性引用的工厂函数会返回相同的迭代器：

```
let arr = ['foo', 'bar', 'baz'];
let iter1 = arr[Symbol.iterator]();

console.log(iter1[Symbol.iterator]);  // f values() { [native code] }

let iter2 = iter1[Symbol.iterator]();

console.log(iter1 === iter2);         // true
```

因为每个迭代器也实现了`Iterable`接口，所以它们可以用在任何期待可迭代对象的地方，比如`for-of`循环：

```
let arr = [3, 1, 4];
let iter = arr[Symbol.iterator]();

for (let item of arr ) { console.log(item); }
// 3
// 1
// 4
```

```
for (let item of iter ) { console.log(item); }
// 3
// 1
// 4
```

7.2.4 提前终止迭代器

可选的`return()`方法用于指定在迭代器提前关闭时执行的逻辑。执行迭代的结构在想让迭代器知道它不想遍历到可迭代对象耗尽时，就可以“关闭”迭代器。可能的情况包括：

- `for-of`循环通过`break`、`continue`、`return`或`throw`提前退出；
- 解构操作并未消费所有值。

`return()`方法必须返回一个有效的`IteratorResult`对象。简单情况下，可以只返回`{ done: true }`。因为这个返回值只会用在生成器的上下文中，所以本章后面再讨论这种情况。

如下面的代码所示，内置语言结构在发现还有更多值可以迭代，但不会消费这些值时，会自动调用`return()`方法。

```
class Counter {
  constructor(limit) {
    this.limit = limit;
  }

  [Symbol.iterator]() {
    let count = 1,
      limit = this.limit;
    return {
      next() {
        if (count <= limit) {
          return { done: false, value: count++ };
        } else {
          return { done: true };
        }
      },
      return() {
        console.log('Exiting early');
        return { done: true };
      }
    };
  }
}


let counter1 = new Counter(5);

for (let i of counter1) {
  if (i > 2) {
    break;
  }
```

```
  console.log(i);
}
// 1
// 2
// 提前退出


let counter2 = new Counter(5);

try {
  for (let i of counter2) {
    if (i > 2) {
      throw 'err';
    }
    console.log(i);
  }
} catch(e) {}
// 1
// 2
// 提前退出


let counter3 = new Counter(5);

let [a, b] = counter3;
// 提前退出
```

如果迭代器没有关闭，则还可以继续从上次离开的地方继续迭代。比如，数组的迭代器就是不能关闭的：

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

for (let i of iter) {
  console.log(i);
  if (i > 2) {
    break
  }
}
// 1
// 2
// 3

for (let i of iter) {
  console.log(i);
}
// 4
// 5
```

因为`return()`方法是可选的，所以并非所有迭代器都是可关闭的。要知道某个迭代器是否可关闭，可以测试这个迭代器实例的`return`属性是不是函数对象。不过，仅仅给一个不可关闭的迭代器增加这个方法**并不能**让它变

成可关闭的。这是因为调用return()不会强制迭代器进入关闭状态。即便如此，return()方法还是会被调用。

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();

iter.return = function() {
  console.log('Exiting early');
  return { done: true };
};

for (let i of iter) {
  console.log(i);
  if (i > 2) {
    break
  }
}
// 1
// 2
// 3
// 提前退出

for (let i of iter) {
  console.log(i);
}
// 4
// 5
```

7.3 生成器

生成器是ECMAScript 6新增的一个极为灵活的结构，拥有在一个函数块内暂停和恢复代码执行的能力。这种新能力具有深远的影响，比如，使用生成器可以自定义迭代器和实现协程。

7.3.1 生成器基础

生成器的形式是一个函数，函数名称前面加一个星号（*）表示它是一个生成器。只要是可以定义函数的地方，就可以定义生成器。

```
// 生成器函数声明
function* generatorFn() {}

// 生成器函数表达式
let generatorFn = function* () {}

// 作为对象字面量方法的生成器函数
let foo = {
  * generatorFn() {}
}

// 作为类实例方法的生成器函数
```

```
class Foo {
  * generatorFn() {}
}

// 作为类静态方法的生成器函数
class Bar {
  static * generatorFn() {}
}
```

> **注意** 箭头函数不能用来定义生成器函数。

标识生成器函数的星号不受两侧空格的影响：

```
// 等价的生成器函数：
function* generatorFnA() {}
function *generatorFnB() {}
function * generatorFnC() {}

// 等价的生成器方法：
class Foo {
  *generatorFnD() {}
  * generatorFnE() {}
}
```

调用生成器函数会产生一个**生成器对象**。生成器对象一开始处于暂停执行（suspended）的状态。与迭代器相似，生成器对象也实现了`Iterator`接口，因此具有`next()`方法。调用这个方法会让生成器开始或恢复执行。

```
function* generatorFn() {}

const g = generatorFn();

console.log(g);       // generatorFn {<suspended>}
console.log(g.next);  // f next() { [native code] }
```

`next()`方法的返回值类似于迭代器，有一个`done`属性和一个`value`属性。函数体为空的生成器函数中间不会停留，调用一次`next()`就会让生成器到达`done: true`状态。

```
function* generatorFn() {}

let generatorObject = generatorFn();

console.log(generatorObject);         // generatorFn {<suspended>}
console.log(generatorObject.next());  // { done: true, value: undefined }
```

`value`属性是生成器函数的返回值，默认值为`undefined`，可以通过生成器函数的返回值指定：

```
function* generatorFn() {
  return 'foo';
}

let generatorObject = generatorFn();

console.log(generatorObject);          // generatorFn {<suspended>}
console.log(generatorObject.next());  // { done: true, value: 'foo' }
```

生成器函数只会在初次调用next()方法后开始执行，如下所示：

```
function* generatorFn() {
  console.log('foobar');
}

// 初次调用生成器函数并不会打印日志
let generatorObject = generatorFn();

generatorObject.next();  // foobar
```

生成器对象实现了Iterable接口，它们默认的迭代器是自引用的：

```
function* generatorFn() {}

console.log(generatorFn);
// f* generatorFn() {}
console.log(generatorFn()[Symbol.iterator]);
// f [Symbol.iterator]() {native code}
console.log(generatorFn());
// generatorFn {<suspended>}
console.log(generatorFn()[Symbol.iterator]());
// generatorFn {<suspended>}

const g = generatorFn();

console.log(g === g[Symbol.iterator]());
// true
```

7.3.2 通过yield中断执行

yield关键字可以让生成器停止和开始执行，也是生成器最有用的地方。生成器函数在遇到yield关键字之前会正常执行。遇到这个关键字后，执行会停止，函数作用域的状态会被保留。停止执行的生成器函数只能通过在生成器对象上调用next()方法来恢复执行：

```
function* generatorFn() {
  yield;
```

```
}

let generatorObject = generatorFn();

console.log(generatorObject.next());  // { done: false, value: undefined }
console.log(generatorObject.next());  // { done: true, value: undefined }
```

此时的yield关键字有点像函数的中间返回语句，它生成的值会出现在next()方法返回的对象里。通过yield关键字退出的生成器函数会处在done: false状态；通过return关键字退出的生成器函数会处于done: true状态。

```
function* generatorFn() {
  yield 'foo';
  yield 'bar';
  return 'baz';
}

let generatorObject = generatorFn();

console.log(generatorObject.next());  // { done: false, value: 'foo' }
console.log(generatorObject.next());  // { done: false, value: 'bar' }
console.log(generatorObject.next());  // { done: true, value: 'baz' }
```

生成器函数内部的执行流程会针对每个生成器对象区分作用域。在一个生成器对象上调用next()不会影响其他生成器：

```
function* generatorFn() {
  yield 'foo';
  yield 'bar';
  return 'baz';
}

let generatorObject1 = generatorFn();
let generatorObject2 = generatorFn();


console.log(generatorObject1.next()); // { done: false, value: 'foo' }
console.log(generatorObject2.next()); // { done: false, value: 'foo' }
console.log(generatorObject2.next()); // { done: false, value: 'bar' }
console.log(generatorObject1.next()); // { done: false, value: 'bar' }
```

yield关键字只能在生成器函数内部使用，用在其他地方会抛出错误。类似函数的return关键字，yield关键字必须直接位于生成器函数定义中，出现在嵌套的非生成器函数中会抛出语法错误：

```
// 有效
function* validGeneratorFn() {
  yield;
```

```
}

// 无效
function* invalidGeneratorFnA() {
  function a() {
    yield;
  }
}

// 无效
function* invalidGeneratorFnB() {
  const b = () => {
    yield;
  }
}

// 无效
function* invalidGeneratorFnC() {
  (() => {
    yield;
  })();
}
```

1. **生成器对象作为可迭代对象**

   在生成器对象上显式调用`next()`方法的用处并不大。其实，如果把生成器对象当成可迭代对象，那么使用起来会更方便：

   ```
   function* generatorFn() {
     yield 1;
     yield 2;
     yield 3;
   }

   for (const x of generatorFn()) {
     console.log(x);
   }
   // 1
   // 2
   // 3
   ```

   在需要自定义迭代对象时，这样使用生成器对象会特别有用。比如，我们需要定义一个可迭代对象，而它会产生一个迭代器，这个迭代器会执行指定的次数。使用生成器，可以通过一个简单的循环来实现：

   ```
   function* nTimes(n) {
     while(n--) {
       yield;
     }
   }
   ```

```
for (let _ of nTimes(3)) {
  console.log('foo');
}
// foo
// foo
// foo
```

传给生成器的函数可以控制迭代循环的次数。在n为0时，while条件为假，循环退出，生成器函数返回。

2. **使用yield实现输入和输出**

除了可以作为函数的中间返回语句使用，yield关键字还可以作为函数的中间参数使用。上一次让生成器函数暂停的yield关键字会接收到传给next()方法的第一个值。这里有个地方不太好理解——第一次调用next()传入的值不会被使用，因为这一次调用是为了开始执行生成器函数：

```
function* generatorFn(initial) {
  console.log(initial);
  console.log(yield);
  console.log(yield);
}

let generatorObject = generatorFn('foo');

generatorObject.next('bar');  // foo
generatorObject.next('baz');  // baz
generatorObject.next('qux');  // qux
```

yield关键字可以同时用于输入和输出，如下例所示：

```
function* generatorFn() {
  return yield 'foo';
}

let generatorObject = generatorFn();

console.log(generatorObject.next());       // { done: false, value: 'foo' }
console.log(generatorObject.next('bar'));  // { done: true, value: 'bar' }
```

因为函数必须对整个表达式求值才能确定要返回的值，所以它在遇到yield关键字时暂停执行并计算出要产生的值："foo"。下一次调用next()传入了"bar"，作为交给同一个yield的值。然后这个值被确定为本次生成器函数要返回的值。

yield关键字并非只能使用一次。比如，以下代码就定义了一个无穷计数生成器函数：

```
```

```
function* generatorFn() {
  for (let i = 0;;++i) {
    yield i;
  }
}

let generatorObject = generatorFn();

console.log(generatorObject.next().value);  // 0
console.log(generatorObject.next().value);  // 1
console.log(generatorObject.next().value);  // 2
console.log(generatorObject.next().value);  // 3
console.log(generatorObject.next().value);  // 4
console.log(generatorObject.next().value);  // 5
...
```

假设我们想定义一个生成器函数，它会根据配置的值迭代相应次数并产生迭代的索引。初始化一个新数组可以实现这个需求，但不用数组也可以实现同样的行为：

```
function* nTimes(n) {
  for (let i = 0; i < n; ++i) {
    yield i;
  }
}

for (let x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

另外，使用`while`循环也可以，而且代码稍微简洁一点：

```
function* nTimes(n) {
  let i = 0;
  while(n--) {
    yield i++;
  }
}

for (let x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

这样使用生成器也可以实现范围和填充数组：

```
function* range(start, end) {
  while(end > start) {
    yield start++;
  }
}

for (const x of range(4, 7)) {
  console.log(x);
}
// 4
// 5
// 6

function* zeroes(n) {
  while(n--) {
    yield 0;
  }
}

console.log(Array.from(zeroes(8))); // [0, 0, 0, 0, 0, 0, 0, 0]
```

3. **产生可迭代对象**

可以使用星号增强`yield`的行为，让它能够迭代一个可迭代对象，从而一次产出一个值：

```
// 等价的generatorFn：
// function* generatorFn() {
//   for (const x of [1, 2, 3]) {
//     yield x;
//   }
// }
function* generatorFn() {
  yield* [1, 2, 3];
}

let generatorObject = generatorFn();

for (const x of generatorFn()) {
  console.log(x);
}
// 1
// 2
// 3
```

与生成器函数的星号类似，`yield`星号两侧的空格不影响其行为：

```
function* generatorFn() {
  yield* [1, 2];
  yield *[3, 4];
  yield * [5, 6];
}

for (const x of generatorFn()) {
  console.log(x);
}
// 1
// 2
// 3
// 4
// 5
// 6
```

因为`yield*`实际上只是将一个可迭代对象序列化为一连串可以单独产出的值，所以这跟把`yield`放到一个循环里没什么不同。下面两个生成器函数的行为是等价的：

```
function* generatorFnA() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

for (const x of generatorFnA()) {
  console.log(x);
}
// 1
// 2
// 3

function* generatorFnB() {
  yield* [1, 2, 3];
}

for (const x of generatorFnB()) {
  console.log(x);
}
// 1
// 2
// 3
```

`yield*`的值是关联迭代器返回`done: true`时的`value`属性。对于普通迭代器来说，这个值是`undefined`：

```
function* generatorFn() {
  console.log('iter value:', yield* [1, 2, 3]);
```

```
}

for (const x of generatorFn()) {
  console.log('value:', x);
}
// value: 1
// value: 2
// value: 3
// iter value: undefined
```

对于生成器函数产生的迭代器来说，这个值就是生成器函数返回的值：

```
function* innerGeneratorFn() {
  yield 'foo';
  return 'bar';
}
function* outerGeneratorFn(genObj) {
  console.log('iter value:', yield* innerGeneratorFn());
}

for (const x of outerGeneratorFn()) {
  console.log('value:', x);
}
// value: foo
// iter value: bar
```

4. **使用`yield*`实现递归算法**

`yield*`最有用的地方是实现递归操作，此时生成器可以产生自身。看下面的例子：

```
function* nTimes(n) {
  if (n > 0) {
    yield* nTimes(n - 1);
    yield n - 1;
  }
}

for (const x of nTimes(3)) {
  console.log(x);
}
// 0
// 1
// 2
```

在这个例子中，每个生成器首先都会从新创建的生成器对象产出每个值，然后再产出一个整数。结果就是生成器函数会递归地减少计数器值，并实例化另一个生成器对象。从最顶层来看，这就相当于创建一个可迭代对象并返回递增的整数。

使用递归生成器结构和`yield*`可以优雅地表达递归算法。下面是一个图的实现，用于生成一个随机的双向图：

```
class Node {
  constructor(id) {
    this.id = id;
    this.neighbors = new Set();
  }

  connect(node) {
    if (node !== this) {
      this.neighbors.add(node);
      node.neighbors.add(this);
    }
  }
}

class RandomGraph {
  constructor(size) {
    this.nodes = new Set();

    // 创建节点
    for (let i = 0; i < size; ++i) {
      this.nodes.add(new Node(i));
    }

    // 随机连接节点
    const threshold = 1 / size;
    for (const x of this.nodes) {
      for (const y of this.nodes) {
        if (Math.random() < threshold) {
          x.connect(y);
        }
      }
    }
  }

  // 这个方法仅用于调试
  print() {
    for (const node of this.nodes) {
      const ids = [...node.neighbors]
                      .map((n) => n.id)
                      .join(',');

      console.log('${node.id}: ${ids}');
    }
  }
}

const g = new RandomGraph(6);

g.print();
```

```
// 示例输出：
// 0: 2,3,5
// 1: 2,3,4,5
// 2: 1,3
// 3: 0,1,2,4
// 4: 2,3
// 5: 0,4
```

图数据结构非常适合递归遍历，而递归生成器恰好非常合用。为此，生成器函数必须接收一个可迭代对象，产出该对象中的每一个值，并且对每个值进行递归。这个实现可以用来测试某个图是否连通，即是否没有不可到达的节点。只要从一个节点开始，然后尽力访问每个节点就可以了。结果就得到了一个非常简洁的深度优先遍历：

```
class Node {
  constructor(id) {
    ...
  }

  connect(node) {
    ...
  }
}

class RandomGraph {
  constructor(size) {
    ...
  }

  print() {
    ...
  }

  isConnected() {
    const visitedNodes = new Set();

    function* traverse(nodes) {
      for (const node of nodes) {
        if (!visitedNodes.has(node)) {
          yield node;
          yield* traverse(node.neighbors);
        }
      }
    }

    // 取得集合中的第一个节点
    const firstNode = this.nodes[Symbol.iterator]().next().value;

    // 使用递归生成器迭代每个节点
    for (const node of traverse([firstNode])) {
      visitedNodes.add(node);
    }
```

```
      return visitedNodes.size === this.nodes.size;
    }
  }
```

7.3.3 生成器作为默认迭代器

因为生成器对象实现了`Iterable`接口，而且生成器函数和默认迭代器被调用之后都产生迭代器，所以生成器格外适合作为默认迭代器。下面是一个简单的例子，这个类的默认迭代器可以用一行代码产出类的内容：

```
class Foo {
  constructor() {
    this.values = [1, 2, 3];
  }
  * [Symbol.iterator]() {
    yield* this.values;
  }
}

const f = new Foo();
for (const x of f) {
  console.log(x);
}
// 1
// 2
// 3
```

这里，`for-of`循环调用了默认迭代器（它恰好又是一个生成器函数）并产生了一个生成器对象。这个生成器对象是可迭代的，所以完全可以在迭代中使用。

7.3.4 提前终止生成器

与迭代器类似，生成器也支持"可关闭"的概念。一个实现`Iterator`接口的对象一定有`next()`方法，还有一个可选的`return()`方法用于提前终止迭代器。生成器对象除了有这两个方法，还有第三个方法：`throw()`。

```
function* generatorFn() {}

const g = generatorFn();

console.log(g);         // generatorFn {<suspended>}
console.log(g.next);    // f next() { [native code] }
console.log(g.return);  // f return() { [native code] }
console.log(g.throw);   // f throw() { [native code] }
```

`return()`和`throw()`方法都可以用于强制生成器进入关闭状态。

1. **`return()`**

return()方法会强制生成器进入关闭状态。提供给return()方法的值，就是终止迭代器对象的值：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

console.log(g);             // generatorFn {<suspended>}
console.log(g.return(4));   // { done: true, value: 4 }
console.log(g);             // generatorFn {<closed>}
```

与迭代器不同，所有生成器对象都有return()方法，只要通过它进入关闭状态，就无法恢复了。后续调用next()会显示done: true状态，而提供的任何返回值都不会被存储或传播：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

console.log(g.next());      // { done: false, value: 1 }
console.log(g.return(4));   // { done: true, value: 4 }
console.log(g.next());      // { done: true, value: undefined }
console.log(g.next());      // { done: true, value: undefined }
console.log(g.next());      // { done: true, value: undefined }
```

for-of循环等内置语言结构会忽略状态为done: true的IteratorObject内部返回的值。

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

for (const x of g) {
  if (x > 1) {
    g.return(4);
  }
  console.log(x);
}
```

```
// 1
// 2
```

2. **throw()**

throw()方法会在暂停的时候将一个提供的错误注入到生成器对象中。如果错误未被处理，生成器就会关闭：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    yield x;
  }
}

const g = generatorFn();

console.log(g);   // generatorFn {<suspended>}
try {
  g.throw('foo');
} catch (e) {
  console.log(e); // foo
}
console.log(g);   // generatorFn {<closed>}
```

不过，假如生成器函数**内部**处理了这个错误，那么生成器就不会关闭，而且还可以恢复执行。错误处理会跳过对应的yield，因此在这个例子中会跳过一个值。比如：

```
function* generatorFn() {
  for (const x of [1, 2, 3]) {
    try {
      yield x;
    } catch(e) {}
  }
}

const g = generatorFn();

console.log(g.next()); // { done: false, value: 1}
g.throw('foo');
console.log(g.next()); // { done: false, value: 3}
```

在这个例子中，生成器在try/catch块中的yield关键字处暂停执行。在暂停期间，throw()方法向生成器对象内部注入了一个错误：字符串"foo"。这个错误会被yield关键字抛出。因为错误是在生成器的try/catch块中抛出的，所以仍然在生成器内部被捕获。可是，由于yield抛出了那个错误，生成器就不会再产出值2。此时，生成器函数继续执行，在下一次迭代再次遇到yield关键字时产出了值3。

> **注意** 如果生成器对象还没有开始执行，那么调用`throw()`抛出的错误不会在函数内部被捕获，因为这相当于在函数块外部抛出了错误。

7.4 小结

迭代是一种所有编程语言中都可以看到的模式。ECMAScript 6正式支持迭代模式并引入了两个新的语言特性：迭代器和生成器。

迭代器是一个可以由任意对象实现的接口，支持连续获取对象产出的每一个值。任何实现`Iterable`接口的对象都有一个`Symbol.iterator`属性，这个属性引用默认迭代器。默认迭代器就像一个迭代器工厂，也就是一个函数，调用之后会产生一个实现`Iterator`接口的对象。

迭代器必须通过连续调用`next()`方法才能连续取得值，这个方法返回一个`IteratorObject`。这个对象包含一个`done`属性和一个`value`属性。前者是一个布尔值，表示是否还有更多值可以访问；后者包含迭代器返回的当前值。这个接口可以通过手动反复调用`next()`方法来消费，也可以通过原生消费者，比如`for-of`循环来自动消费。

生成器是一种特殊的函数，调用之后会返回一个生成器对象。生成器对象实现了`Iterable`接口，因此可用在任何消费可迭代对象的地方。生成器的独特之处在于支持`yield`关键字，这个关键字能够暂停执行生成器函数。使用`yield`关键字还可以通过`next()`方法接收输入和产生输出。在加上星号之后，`yield`关键字可以将跟在它后面的可迭代对象序列化为一连串值。

第 8 章 对象、类与面向对象编程

> **本章内容**
>
> - 理解对象
> - 理解对象创建过程
> - 理解继承
> - 理解类

ECMA-262将对象定义为一组属性的无序集合。严格来说，这意味着对象就是一组没有特定顺序的值。对象的每个属性或方法都由一个名称来标识，这个名称映射到一个值。正因为如此（以及其他还未讨论的原因），可以把ECMAScript的对象想象成一张散列表，其中的内容就是一组名/值对，值可以是数据或者函数。

8.1 理解对象

创建自定义对象的通常方式是创建`Object`的一个新实例，然后再给它添加属性和方法，如下例所示：

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
person.job = "Software Engineer";
person.sayName = function() {
  console.log(this.name);
};
```

这个例子创建了一个名为`person`的对象，而且有三个属性（`name`、`age`和`job`）和一个方法（`sayName()`）。`sayName()`方法会显示`this.name`的值，这个属性会解析为`person.name`。早期JavaScript开发者频繁使用这种

方式创建新对象。几年后，对象字面量变成了更流行的方式。前面的例子如果使用对象字面量则可以这样写：

```
let person = {
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};
```

这个例子中的person对象跟前面例子中的person对象是等价的，它们的属性和方法都一样。这些属性都有自己的特征，而这些特征决定了它们在JavaScript中的行为。

8.1.1 属性的类型

ECMA-262使用一些内部特性来描述属性的特征。这些特性是由为JavaScript实现引擎的规范定义的。因此，开发者不能在JavaScript中直接访问这些特性。为了将某个特性标识为内部特性，规范会用两个中括号把特性的名称括起来，比如[[Enumerable]]。

属性分两种：数据属性和访问器属性。

1. **数据属性**

   数据属性包含一个保存数据值的位置。值会从这个位置读取，也会写入到这个位置。数据属性有4个特性描述它们的行为。

   - [[Configurable]]：表示属性是否可以通过delete删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认情况下，所有直接定义在对象上的属性的这个特性都是true，如前面的例子所示。
   - [[Enumberable]]：表示属性是否可以通过for-in循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是true，如前面的例子所示。
   - [[Writable]]：表示属性的值是否可以被修改。默认情况下，所有直接定义在对象上的属性的这个特性都是true，如前面的例子所示。
   - [[Value]]：包含属性实际的值。这就是前面提到的那个读取和写入属性值的位置。这个特性的默认值为undefined。

   在像前面例子中那样将属性显式添加到对象之后，[[Configurable]]、[[Enumerable]]和[[Writable]]都会被设置为true，而[[Value]]特性会被设置为指定的值。比如：

   ```
   let person = {
     name: "Nicholas"
   };
   ```

   这里，我们创建了一个名为name的属性，并给它赋予了一个值"Nicholas"。这意味着[[Value]]特性会被设置为"Nicholas"，之后对这个值的任何修改都会保存这个位置。

要修改属性的默认特性，就必须使用Object.defineProperty()方法。这个方法接收3个参数：要给其添加属性的对象、属性的名称和一个描述符对象。最后一个参数，即描述符对象上的属性可以包含：configurable、enumerable、writable和value，跟相关特性的名称一一对应。根据要修改的特性，可以设置其中一个或多个值。比如：

```
let person = {};
Object.defineProperty(person, "name", {
  writable: false,
  value: "Nicholas"
});
console.log(person.name); // "Nicholas"
person.name = "Greg";
console.log(person.name); // "Nicholas"
```

这个例子创建了一个名为name的属性并给它赋予了一个只读的值"Nicholas"。这个属性的值就不能再修改了，在非严格模式下尝试给这个属性重新赋值会被忽略。在严格模式下，尝试修改只读属性的值会抛出错误。

类似的规则也适用于创建不可配置的属性。比如：

```
let person = {};
Object.defineProperty(person, "name", {
  configurable: false,
  value: "Nicholas"
});
console.log(person.name); // "Nicholas"
delete person.name;
console.log(person.name); // "Nicholas"
```

这个例子把configurable设置为false，意味着这个属性不能从对象上删除。非严格模式下对这个属性调用delete没有效果，严格模式下会抛出错误。此外，一个属性被定义为不可配置之后，就不能再变回可配置的了。再次调用Object.defineProperty()并修改任何非writable属性会导致错误：

```
let person = {};
Object.defineProperty(person, "name", {
  configurable: false,
  value: "Nicholas"
});

// 抛出错误
Object.defineProperty(person, "name", {
  configurable: true,
  value: "Nicholas"
});
```

因此，虽然可以对同一个属性多次调用Object.defineProperty()，但在把configurable设置为false之后就会受限制了。

在调用Object.defineProperty()时，configurable、enumerable和writable的值如果不指定，则都默认为false。多数情况下，可能都不需要Object.defineProperty()提供的这些强大的设置，但要理解JavaScript对象，就要理解这些概念。

2. **访问器属性**

访问器属性不包含数据值。相反，它们包含一个获取（getter）函数和一个设置（setter）函数，不过这两个函数不是必需的。在读取访问器属性时，会调用获取函数，这个函数的责任就是返回一个有效的值。在写入访问器属性时，会调用设置函数并传入新值，这个函数必须决定对数据做出什么修改。访问器属性有4个特性描述它们的行为。

- [[Configurable]]：表示属性是否可以通过delete删除并重新定义，是否可以修改它的特性，以及是否可以把它改为数据属性。默认情况下，所有直接定义在对象上的属性的这个特性都是true。
- [[Enumerable]]：表示属性是否可以通过for-in循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是true。
- [[Get]]：获取函数，在读取属性时调用。默认值为undefined。
- [[Set]]：设置函数，在写入属性时调用。默认值为undefined。

访问器属性是不能直接定义的，必须使用Object.defineProperty()。下面是一个例子：

```
// 定义一个对象，包含伪私有成员year_和公共成员edition
let book = {
  year_: 2017,
  edition: 1
};

Object.defineProperty(book, "year", {
  get() {
    return this.year_;
  },
  set(newValue) {
    if (newValue > 2017) {
      this.year_ = newValue;
      this.edition += newValue - 2017;
    }
  }
});
book.year = 2018;
console.log(book.edition); // 2
```

在这个例子中，对象book有两个默认属性：year_和edition。year_中的下划线常用来表示该属性并不希望在对象方法的外部被访问。另一个属性year被定义为一个访问器属性，其中获取函数简单地返回year_的值，而设置函数会做一些计算以决定正确的版本（edition）。因此，把year属性修改为2018会导致year_变成2018，edition变成2。这是访问器属性的典型使用场景，即设置一个属性值会导致一些其他变化发生。

获取函数和设置函数不一定都要定义。只定义获取函数意味着属性是只读的，尝试修改属性会被忽略。在严格模式下，尝试写入只定义了获取函数的属性会抛出错误。类似地，只有一个设置函数的属性是不能读取的，非严格模式下读取会返回`undefined`，严格模式下会抛出错误。

在不支持`Object.defineProperty()`的浏览器中没有办法修改`[[Configurable]]`或`[[Enumerable]]`。

> **注意** 在ECMAScript 5以前，开发者会使用两个非标准的访问创建访问器属性：`__defineGetter__()`和`__defineSetter__()`。这两个方法最早是Firefox引入的，后来Safari、Chrome和Opera也实现了。

8.1.2 定义多个属性

在一个对象上同时定义多个属性的可能性是非常大的。为此，ECMAScript提供了`Object.defineProperties()`方法。这个方法可以通过多个描述符一次性定义多个属性。它接收两个参数：要为之添加或修改属性的对象和另一个描述符对象，其属性与要添加或修改的属性一一对应。比如：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },

  year: {
    get() {
      return this.year_;
    },

    set(newValue) {
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});
```

这段代码在`book`对象上定义了两个数据属性`year_`和`edition`，还有一个访问器属性`year`。最终的对象跟上一节示例中的一样。唯一的区别是所有属性都是同时定义的。

8.1.3 读取属性的特性

使用`Object.getOwnPropertyDescriptor()`方法可以取得指定属性的属性描述符。这个方法接收两个参数：属性所在的对象和要取得其描述符的属性名。返回值是一个对象，对于访问器属性包含`configurable`、`enumerable`、`get`和`set`属性，对于数据属性包含`configurable`、`enumberable`、`writable`和`value`属性。比如：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },

  year: {
    get: function() {
      return this.year_;
    },

    set: function(newValue){
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});

let descriptor = Object.getOwnPropertyDescriptor(book, "year_");
console.log(descriptor.value);          // 2017
console.log(descriptor.configurable);   // false
console.log(typeof descriptor.get);     // "undefined"
let descriptor = Object.getOwnPropertyDescriptor(book, "year");
console.log(descriptor.value);          // undefined
console.log(descriptor.enumerable);     // false
console.log(typeof descriptor.get);     // "function"
```

对于数据属性`year_`，`value`等于原来的值，`configurable`是`false`，`get`是`undefined`。对于访问器属性`year`，`value`是`undefined`，`enumerable`是`false`，`get`是一个指向获取函数的指针。

ECMAScript 2017新增了`Object.getOwnPropertyDescriptors()`静态方法。这个方法实际上会在每个自有属性上调用`Object.defineProperties()`并在一个新对象中返回它们。对于前面的例子，使用这个静态方法会返回如下对象：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },
```

```
    year: {
      get: function() {
        return this.year_;
      },

      set: function(newValue){
        if (newValue > 2017) {
          this.year_ = newValue;
          this.edition += newValue - 2017;
        }
      }
    }
  });

  console.log(Object.getOwnPropertyDescriptors(book));
  // {
  //   edition: {
  //     configurable: false,
  //     enumerable: false,
  //     value: 1,
  //     writable: false
  //   },
  //   year: {
  //     configurable: false,
  //     enumerable: false,
  //     get: f(),
  //     set: f(newValue),
  //   },
  //   year_: {
  //     configurable: false,
  //     enumerable: false,
  //     value: 2019,
  //     writable: false
  //   }
  // }
```

8.1.4 合并对象

JavaScript开发者经常觉得“合并”（merge）两个对象很有用。更具体地说，就是把源对象所有的本地属性一起复制到目标对象上。有时候这种操作也被称为“混入”（mixin），因为目标对象通过混入源对象的属性得到了增强。

ECMAScript 6专门为合并对象提供了`Object.assign()`方法。这个方法接收一个目标对象和一个或多个源对象作为参数，然后将每个源对象中可枚举（`Object.propertyIsEnumerable()`返回`true`）和自有（`Object.hasOwnProperty()`返回`true`）属性复制到目标对象。以字符串和符号为键的属性会被复制。对每个符合条件的属性，这个方法会使用源对象上的`[[Get]]`取得属性的值，然后使用目标对象上的`[[Set]]`设置属性的值。

```
  let dest, src, result;

  /**
```

```
 * 简单复制
 */
dest = {};
src = { id: 'src' };

result = Object.assign(dest, src);

// Object.assign修改目标对象
// 也会返回修改后的目标对象
console.log(dest === result); // true
console.log(dest !== src);    // true
console.log(result);          // { id: src }
console.log(dest);            // { id: src }


/**
 * 多个源对象
 */
dest = {};

result = Object.assign(dest, { a: 'foo' }, { b: 'bar' });

console.log(result); // { a: foo, b: bar }


/**
 * 获取函数与设置函数
 */
dest = {
  set a(val) {
    console.log('Invoked dest setter with param ${val}');
  }
};
src = {
  get a() {
    console.log('Invoked src getter');
    return 'foo';
  }
};

Object.assign(dest, src);
// 调用src的获取方法
// 调用dest的设置方法并传入参数"foo"
// 因为这里的设置函数不执行赋值操作
// 所以实际上并没有把值转移过来
console.log(dest); // { set a(val) {...} }
```

`Object.assign()`实际上对每个源对象执行的是浅复制。如果多个源对象都有相同的属性，则使用最后一个复制的值。此外，从源对象访问器属性取得的值，比如获取函数，会作为一个静态值赋给目标对象。换句话说，不能在两个对象间转移获取函数和设置函数。

```
let dest, src, result;

/**
 * 覆盖属性
 */
dest = { id: 'dest' };

result = Object.assign(dest, { id: 'src1', a: 'foo' }, { id: 'src2', b: 'bar' });

// Object.assign会覆盖重复的属性
console.log(result); // { id: src2, a: foo, b: bar }

// 可以通过目标对象上的设置函数观察到覆盖的过程：
dest = {
  set id(x) {
    console.log(x);
  }
};

Object.assign(dest, { id: 'first' }, { id: 'second' }, { id: 'third' });
// first
// second
// third


/**
 * 对象引用
 */

dest = {};
src = { a: {} };

Object.assign(dest, src);

// 浅复制意味着只会复制对象的引用
console.log(dest);              // { a :{} }
console.log(dest.a === src.a);  // true
```

如果赋值期间出错，则操作会中止并退出，同时抛出错误。`Object.assign()`没有“回滚”之前赋值的概念，因此它是一个尽力而为、可能只会完成部分复制的方法。

```
let dest, src, result;

/**
 * 错误处理
 */
dest = {};
src = {
  a: 'foo',
  get b() {
    // Object.assign()在调用这个获取函数时会抛出错误
```

```
    throw new Error();
  },
  c: 'bar'
};

try {
  Object.assign(dest, src);
} catch(e) {}

// Object.assign()没办法回滚已经完成的修改
// 因此在抛出错误之前，目标对象上已经完成的修改会继续存在：
console.log(dest); // { a: foo }
```

8.1.5 对象标识及相等判定

在ECMAScript 6之前，有些特殊情况即使是===操作符也无能为力：

```
// 这些是===符合预期的情况
console.log(true === 1);  // false
console.log({} === {});   // false
console.log("2" === 2);   // false

// 这些情况在不同JavaScript引擎中表现不同，但仍被认为相等
console.log(+0 === -0);   // true
console.log(+0 === 0);    // true
console.log(-0 === 0);    // true

// 要确定NaN的相等性，必须使用极为讨厌的isNaN()
console.log(NaN === NaN); // false
console.log(isNaN(NaN));  // true
```

为改善这类情况，ECMAScript 6规范新增了Object.is()，这个方法与===很像，但同时也考虑到了上述边界情形。这个方法必须接收两个参数：

```
console.log(Object.is(true, 1));  // false
console.log(Object.is({}, {}));   // false
console.log(Object.is("2", 2));   // false

// 正确的0、-0、+0相等/不等判定
console.log(Object.is(+0, -0));   // false
console.log(Object.is(+0, 0));    // true
console.log(Object.is(-0, 0));    // false

// 正确的NaN相等判定
console.log(Object.is(NaN, NaN)); // true
```

要检查超过两个值，递归地利用相等性传递即可：

```
function recursivelyCheckEqual(x, ...rest) {
  return Object.is(x, rest[0]) &&
         (rest.length < 2 || recursivelyCheckEqual(...rest));
}
```

8.1.6 增强的对象语法

ECMAScript 6为定义和操作对象新增了很多极其有用的语法糖特性。这些特性都没有改变现有引擎的行为，但极大地提升了处理对象的方便程度。

本节介绍的所有对象语法同样适用于ECMAScript 6的类，本章后面会讨论。

> **注意** 相比于以往的替代方案，本节介绍的增强对象语法可以说是一骑绝尘。因此本章及本书会默认使用这些新语法特性。

1. **属性值简写**

   在给对象添加变量的时候，开发者经常会发现属性名和变量名是一样的。例如：

   ```
   let name = 'Matt';

   let person = {
     name: name
   };

   console.log(person); // { name: 'Matt' }
   ```

   为此，简写属性名语法出现了。简写属性名只要使用变量名（不用再写冒号）就会自动被解释为同名的属性键。如果没有找到同名变量，则会抛出`ReferenceError`。

   以下代码和之前的代码是等价的：

   ```
   let name = 'Matt';

   let person = {
     name
   };

   console.log(person); // { name: 'Matt' }
   ```

   代码压缩程序会在不同作用域间保留属性名，以防止找不到引用。以下面的代码为例：

   ```
   function makePerson(name) {
     return {
       name
     };
   ```

```
}

let person = makePerson('Matt');

console.log(person.name);  // Matt
```

在这里，即使参数标识符只限定于函数作用域，编译器也会保留初始的name标识符。如果使用Google Closure编译器压缩，那么函数参数会被缩短，而属性名不变：

```
function makePerson(a) {
  return {
    name: a
  };
}

var person = makePerson("Matt");

console.log(person.name); // Matt
```

2. **可计算属性**

在引入可计算属性之前，如果想使用变量的值作为属性，那么必须先声明对象，然后使用中括号语法来添加属性。换句话说，不能在对象字面量中直接动态命名属性。比如：

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';

let person = {};
person[nameKey] = 'Matt';
person[ageKey] = 27;
person[jobKey] = 'Software engineer';

console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }
```

有了可计算属性，就可以在对象字面量中完成动态属性赋值。中括号包围的对象属性键告诉运行时将其作为JavaScript表达式而不是字符串来求值：

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';

let person = {
  [nameKey]: 'Matt',
  [ageKey]: 27,
  [jobKey]: 'Software engineer'
};
```

```
console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }
```

因为被当作JavaScript表达式求值，所以可计算属性本身可以是复杂的表达式，在实例化时再求值：

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let uniqueToken = 0;

function getUniqueKey(key) {
  return '${key}_${uniqueToken++}';
}

let person = {
  [getUniqueKey(nameKey)]: 'Matt',
  [getUniqueKey(ageKey)]: 27,
  [getUniqueKey(jobKey)]: 'Software engineer'
};

console.log(person);  // { name_0: 'Matt', age_1: 27, job_2: 'Software
engineer' }
```

> **注意** 可计算属性表达式中抛出任何错误都会中断对象创建。如果计算属性的表达式有副作用，那就要小心了，因为如果表达式抛出错误，那么之前完成的计算是不能回滚的。

3. **简写方法名**

在给对象定义方法时，通常都要写一个方法名、冒号，然后再引用一个匿名函数表达式，如下所示：

```
let person = {
  sayName: function(name) {
    console.log('My name is ${name}');
  }
};

person.sayName('Matt'); // My name is Matt
```

新的简写方法的语法遵循同样的模式，但开发者要放弃给函数表达式命名（不过给作为方法的函数命名通常没什么用）。相应地，这样也可以明显缩短方法声明。

以下代码和之前的代码在行为上是等价的：

```
let person = {
  sayName(name) {
    console.log('My name is ${name}');
  }
```

```
};

person.sayName('Matt'); // My name is Matt
```

简写方法名对获取函数和设置函数也是适用的：

```
let person = {
  name_: '',
  get name() {
    return this.name_;
  },
  set name(name) {
    this.name_ = name;
  },
  sayName() {
    console.log('My name is ${this.name_}');
  }
};

person.name = 'Matt';
person.sayName(); // My name is Matt
```

简写方法名与可计算属性键相互兼容：

```
const methodKey = 'sayName';

let person = {
  [methodKey](name) {
    console.log('My name is ${name}');
  }
}

person.sayName('Matt'); // My name is Matt
```

> **注意** 简写方法名对于本章后面介绍的ECMAScript 6的类更有用。

8.1.7 对象解构

ECMAScript 6新增了对象解构语法，可以在一条语句中使用嵌套数据实现一个或多个赋值操作。简单地说，对象解构就是使用与对象匹配的结构来实现对象属性赋值。

下面的例子展示了两段等价的代码，首先是不使用对象解构的：

```
// 不使用对象解构
let person = {
  name: 'Matt',
  age: 27
```

```
};

let personName = person.name,
    personAge = person.age;

console.log(personName); // Matt
console.log(personAge);  // 27
```

然后，是使用对象解构的：

```
// 使用对象解构
let person = {
  name: 'Matt',
  age: 27
};

let { name: personName, age: personAge } = person;

console.log(personName);  // Matt
console.log(personAge);   // 27
```

使用解构，可以在一个类似对象字面量的结构中，声明多个变量，同时执行多个赋值操作。如果想让变量直接使用属性的名称，那么可以使用简写语法，比如：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, age } = person;

console.log(name);  // Matt
console.log(age);   // 27
```

解构赋值不一定与对象的属性匹配。赋值的时候可以忽略某些属性，而如果引用的属性不存在，则该变量的值就是`undefined`：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, job } = person;

console.log(name);  // Matt
console.log(job);   // undefined
```

也可以在解构赋值的同时定义默认值，这适用于前面刚提到的引用的属性不存在于源对象中的情况：

```
let person = {
  name: 'Matt',
  age: 27
};

let { name, job='Software engineer' } = person;

console.log(name); // Matt
console.log(job);  // Software engineer
```

解构在内部使用函数`ToObject()`（不能在运行时环境中直接访问）把源数据结构转换为对象。这意味着在对象解构的上下文中，原始值会被当成对象。这也意味着（根据`ToObject()`的定义），`null`和`undefined`不能被解构，否则会抛出错误。

```
let { length } = 'foobar';
console.log(length);        // 6

let { constructor: c } = 4;
console.log(c === Number);  // true

let { _ } = null;           // TypeError

let { _ } = undefined;      // TypeError
```

解构并不要求变量必须在解构表达式中声明。不过，如果是给事先声明的变量赋值，则赋值表达式必须包含在一对括号中：

```
let personName, personAge;

let person = {
  name: 'Matt',
  age: 27
};

({name: personName, age: personAge} = person);

console.log(personName, personAge); // Matt, 27
```

1. **嵌套解构**

   解构对于引用嵌套的属性或赋值目标没有限制。为此，可以通过解构来复制对象属性：

   ```
   let person = {
     name: 'Matt',
   ```

```
  age: 27,
  job: {
    title: 'Software engineer'
  }
};
let personCopy = {};


({
  name: personCopy.name,
  age: personCopy.age,
  job: personCopy.job
} = person);

// 因为一个对象的引用被赋值给personCopy，所以修改
// person.job对象的属性也会影响personCopy
person.job.title = 'Hacker'

console.log(person);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }

console.log(personCopy);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }
```

解构赋值可以使用嵌套结构，以匹配嵌套的属性：

```
let person = {
  name: 'Matt',
  age: 27,
  job: {
    title: 'Software engineer'
  }
};

// 声明title变量并将person.job.title的值赋给它
let { job: { title } } = person;

console.log(title); // Software engineer
```

在外层属性没有定义的情况下不能使用嵌套解构。无论源对象还是目标对象都一样：

```
let person = {
  job: {
    title: 'Software engineer'
  }
};
let personCopy = {};

// foo在源对象上是undefined
```

```
({
  foo: {
    bar: personCopy.bar
  }
} = person);
// TypeError: Cannot destructure property 'bar' of 'undefined' or 'null'.

// job在目标对象上是undefined
({
  job: {
    title: personCopy.job.title
  }
} = person);
// TypeError: Cannot set property 'title' of undefined
```

2. **部分解构**

需要注意的是，涉及多个属性的解构赋值是一个输出无关的顺序化操作。如果一个解构表达式涉及多个赋值，开始的赋值成功而后面的赋值出错，则整个解构赋值只会完成一部分：

```
let person = {
  name: 'Matt',
  age: 27
};

let personName, personBar, personAge;

try {
  // person.foo是undefined，因此会抛出错误
  ({name: personName, foo: { bar: personBar }, age: personAge} = person);
} catch(e) {}

console.log(personName, personBar, personAge);
// Matt, undefined, undefined
```

3. **参数上下文匹配**

在函数参数列表中也可以进行解构赋值。对参数的解构赋值不会影响`arguments`对象，但可以在函数签名中声明在函数体内使用局部变量：

```
let person = {
  name: 'Matt',
  age: 27
};

function printPerson(foo, {name, age}, bar) {
  console.log(arguments);
  console.log(name, age);
}
```

```
function printPerson2(foo, {name: personName, age: personAge}, bar) {
  console.log(arguments);
  console.log(personName, personAge);
}

printPerson('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27

printPerson2('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27
```

8.2 创建对象

虽然使用`Object`构造函数或对象字面量可以方便地创建对象，但这些方式也有明显不足：创建具有同样接口的多个对象需要重复编写很多代码。

8.2.1 概述

综观ECMAScript规范的历次发布，每个版本的特性似乎都出人意料。ECMAScript 5.1并没有正式支持面向对象的结构，比如类或继承。但是，正如接下来几节会介绍的，巧妙地运用原型式继承可以成功地模拟同样的行为。

ECMAScript 6开始正式支持类和继承。ES6的类旨在完全涵盖之前规范设计的基于原型的继承模式。不过，无论从哪方面看，ES6的类都仅仅是封装了ES5.1构造函数加原型继承的语法糖而已。

> **注意** 不要误会：采用面向对象编程模式的JavaScript代码还是应该使用ECMAScript 6的类。但不管怎么说，理解ES6类出现之前的惯例总是有益无害的。特别是ES6的类定义本身就相当于对原有结构的封装。因此，在介绍ES6的类之前，本书会循序渐进地介绍被类取代的那些底层概念。

8.2.2 工厂模式

工厂模式是一种众所周知的设计模式，广泛应用于软件工程领域，用于抽象创建特定对象的过程。（本书后面还会讨论其他设计模式及其在JavaScript中的实现。）下面的例子展示了一种按照特定接口创建对象的方式：

```
function createPerson(name, age, job) {
  let o = new Object();
  o.name = name;
  o.age = age;
  o.job = job;
  o.sayName = function() {
    console.log(this.name);
  };
  return o;
}

let person1 = createPerson("Nicholas", 29, "Software Engineer");
let person2 = createPerson("Greg", 27, "Doctor");
```

这里，函数createPerson()接收3个参数，根据这几个参数构建了一个包含Person信息的对象。可以用不同的参数多次调用这个函数，每次都会返回包含3个属性和1个方法的对象。这种工厂模式虽然可以解决创建多个类似对象的问题，但没有解决对象标识问题（即新创建的对象是什么类型）。

8.2.3 构造函数模式

前面几章提到过，ECMAScript中的构造函数是用于创建特定类型对象的。像Object和Array这样的原生构造函数，运行时可以直接在执行环境中使用。当然也可以自定义构造函数，以函数的形式为自己的对象类型定义属性和方法。

比如，前面的例子使用构造函数模式可以这样写：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg
```

在这个例子中，Person()构造函数代替了createPerson()工厂函数。实际上，Person()内部的代码跟createPerson()基本是一样的，只是有如下区别。

- 没有显式地创建对象。
- 属性和方法直接赋值给了this。
- 没有return。

另外，要注意函数名Person的首字母大写了。按照惯例，构造函数名称的首字母都是要大写的，非构造函数则以小写字母开头。这是从面向对象编程语言那里借鉴的，有助于在ECMAScript中区分构造函数和普通函数。毕竟ECMAScript的构造函数就是能创建对象的函数。

要创建Person的实例，应使用new操作符。以这种方式调用构造函数会执行如下操作。

(1) 在内存中创建一个新对象。

(2) 这个新对象内部的[[Prototype]]特性被赋值为构造函数的prototype属性。

(3) 构造函数内部的this被赋值为这个新对象（即this指向新对象）。

(4) 执行构造函数内部的代码（给新对象添加属性）。

(5) 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

上一个例子的最后，person1和person2分别保存着Person的不同实例。这两个对象都有一个constructor属性指向Person，如下所示：

```
console.log(person1.constructor == Person);  // true
console.log(person2.constructor == Person);  // true
```

constructor本来是用于标识对象类型的。不过，一般认为instanceof操作符是确定对象类型更可靠的方式。前面例子中的每个对象都是Object的实例，同是也是Person的实例，如下面调用instanceof操作符的结果所示：

```
console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

定义自定义构造函数可以确保实例被标识为特定类型，相比于工厂模式，这是一个很大的好处。在这个例子中，person1和person2之所以也被认为是Object的实例，是因为所有自定义对象都继承自Object（后面再详细讨论这一点）。

构造函数不一定要写成函数声明的形式。赋值给变量的函数表达式也可以表示构造函数：

```
let Person = function(name, age, job) {
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg

console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

在实例化时，如果不想传参数，那么构造函数后面的括号可加可不加。只要有new操作符，就可以调用相应的构造函数：

```

```

```
function Person() {
  this.name = "Jake";
  this.sayName = function() {
    console.log(this.name);
  };
}

let person1 = new Person();
let person2 = new Person;

person1.sayName();  // Jake
person2.sayName();  // Jake

console.log(person1 instanceof Object);  // true
console.log(person1 instanceof Person);  // true
console.log(person2 instanceof Object);  // true
console.log(person2 instanceof Person);  // true
```

1. **构造函数也是函数**

   构造函数与普通函数唯一的区别就是调用方式不同。除此之外，构造函数也是函数。并没有把某个函数定义为构造函数的特殊语法。任何函数只要使用new操作符调用就是构造函数，而不使用new操作符调用的函数就是普通函数。比如，前面的例子中定义的Person()可以像下面这样调用：

   ```
   // 作为构造函数
   let person = new Person("Nicholas", 29, "Software Engineer");
   person.sayName();    // "Nicholas"

   // 作为函数调用
   Person("Greg", 27, "Doctor");   // 添加到window对象
   window.sayName();    // "Greg"

   // 在另一个对象的作用域中调用
   let o = new Object();
   Person.call(o, "Kristen", 25, "Nurse");
   o.sayName();   // "Kristen"
   ```

   这个例子一开始展示了典型的构造函数调用方式，即使用new操作符创建一个新对象。然后是普通函数的调用方式，这时候没有使用new操作符调用Person()，结果会将属性和方法添加到window对象。这里要记住，在调用一个函数而没有明确设置this值的情况下（即没有作为对象的方法调用，或者没有使用call()/apply()调用），this始终指向Global对象（在浏览器中就是window对象）。因此在上面的调用之后，window对象上就有了一个sayName()方法，调用它会返回"Greg"。最后展示的调用方式是通过call()（或apply()）调用函数，同时将特定对象指定为作用域。这里的调用将对象o指定为Person()内部的this值，因此执行完函数代码后，所有属性和sayName()方法都会添加到对象o上面。

2. **构造函数的问题**

构造函数虽然有用，但也不是没有问题。构造函数的主要问题在于，其定义的方法会在每个实例上都创建一遍。因此对前面的例子而言，person1和person2都有名为sayName()的方法，但这两个方法不是同一个Function实例。我们知道，ECMAScript中的函数是对象，因此每次定义函数时，都会初始化一个对象。逻辑上讲，这个构造函数实际上是这样的：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = new Function("console.log(this.name)"); // 逻辑等价
}
```

这样理解这个构造函数可以更清楚地知道，每个Person实例都会有自己的Function实例用于显示name属性。当然了，以这种方式创建函数会带来不同的作用域链和标识符解析。但创建新Function实例的机制是一样的。因此不同实例上的函数虽然同名却不相等，如下所示：

```
console.log(person1.sayName == person2.sayName); // false
```

因为都是做一样的事，所以没必要定义两个不同的Function实例。况且，this对象可以把函数与对象的绑定推迟到运行时。

要解决这个问题，可以把函数定义转移到构造函数外部：

```
function Person(name, age, job){
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = sayName;
}

function sayName() {
  console.log(this.name);
}

let person1 = new Person("Nicholas", 29, "Software Engineer");
let person2 = new Person("Greg", 27, "Doctor");

person1.sayName();  // Nicholas
person2.sayName();  // Greg
```

在这里，sayName()被定义在了构造函数外部。在构造函数内部，sayName属性等于全局sayName()函数。因为这一次sayName属性中包含的只是一个指向外部函数的指针，所以person1和person2共享了定义在全局作用域上的sayName()函数。这样虽然解决了相同逻辑的函数重复定义的问题，但全局作用域也因此被搞乱了，因为那个函数实际上只能在一个对象上调用。如果这个对象需要多个方法，那么就要

在全局作用域中定义多个函数。这会导致自定义类型引用的代码不能很好地聚集一起。这个新问题可以通过原型模式来解决。

8.2.4 原型模式

每个函数都会创建一个prototype属性，这个属性是一个对象，包含应该由特定引用类型的实例共享的属性和方法。实际上，这个对象就是通过调用构造函数创建的对象的原型。使用原型对象的好处是，在它上面定义的属性和方法可以被对象实例共享。原来在构造函数中直接赋给对象实例的值，可以直接赋值给它们的原型，如下所示：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
person1.sayName(); // "Nicholas"

let person2 = new Person();
person2.sayName(); // "Nicholas"

console.log(person1.sayName == person2.sayName); // true
```

使用函数表达式也可以：

```
let Person = function() {};

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
person1.sayName();   // "Nicholas"

let person2 = new Person();
person2.sayName();   // "Nicholas"

console.log(person1.sayName == person2.sayName); // true
```

这里，所有属性和sayName()方法都直接添加到了Person的prototype属性上，构造函数体中什么也没有。但这样定义之后，调用构造函数创建的新对象仍然拥有相应的属性和方法。与构造函数模式不同，使用这种原型

模式定义的属性和方法是由所有实例共享的。因此person1和person2访问的都是相同的属性和相同的sayName()函数。要理解这个过程，就必须理解ECMAScript中原型的本质。

1. **理解原型**

   无论何时，只要创建一个函数，就会按照特定的规则为这个函数创建一个prototype属性（指向原型对象）。默认情况下，所有原型对象自动获得一个名为constructor的属性，指回与之关联的构造函数。对前面的例子而言，Person.prototype.constructor指向Person。然后，因构造函数而异，可能会给原型对象添加其他属性和方法。

   在自定义构造函数时，原型对象默认只会获得constructor属性，其他的所有方法都继承自Object。每次调用构造函数创建一个新实例，这个实例的内部[[Prototype]]指针就会被赋值为构造函数的原型对象。脚本中没有访问这个[[Prototype]]特性的标准方式，但Firefox、Safari和Chrome会在每个对象上暴露__proto__属性，通过这个属性可以访问对象的原型。在其他实现中，这个特性完全被隐藏了。关键在于理解这一点：实例与构造函数原型之间有直接的联系，但实例与构造函数之间没有。

   这种关系不好可视化，但可以通过下面的代码来理解原型的行为：

```
/**
 * 构造函数可以是函数表达式
 * 也可以是函数声明，因此以下两种形式都可以：
 *   function Person {}
 *   let Person = function() {}
 */
function Person() {}

/**
 * 声明之后，构造函数就有了一个
 * 与之关联的原型对象：
 */
console.log(typeof Person.prototype);
console.log(Person.prototype);
// {
//   constructor: f Person(),
//   __proto__: Object
// }

/**
 * 如前所述，构造函数有一个prototype属性
 * 引用其原型对象，而这个原型对象也有一个
 * constructor属性，引用这个构造函数
 * 换句话说，两者循环引用：
 */
console.log(Person.prototype.constructor === Person); // true

/**
 * 正常的原型链都会终止于Object的原型对象
 * Object原型的原型是null
 */
console.log(Person.prototype.__proto__ === Object.prototype);   // true
console.log(Person.prototype.__proto__.constructor === Object); // true
console.log(Person.prototype.__proto__.__proto__ === null);     // true
```

```
console.log(Person.prototype.__proto__);
// {
//   constructor: f Object(),
//   toString: ...
//   hasOwnProperty: ...
//   isPrototypeOf: ...
//   ...
// }


let person1 = new Person(),
    person2 = new Person();

/**
 * 构造函数、原型对象和实例
 * 是3个完全不同的对象：
 */
console.log(person1 !== Person);           // true
console.log(person1 !== Person.prototype); // true
console.log(Person.prototype !== Person);  // true

/**
  * 实例通过__proto__链接到原型对象，
  * 它实际上指向藏特性[[Prototype]]
  *
  * 构造函数通过prototype属性链接到原型对象
  *
  * 实例与构造函数没有直接联系，与原型对象有直接联系
  */
console.log(person1.__proto__ === Person.prototype);   // true
conosle.log(person1.__proto__.constructor === Person); // true

/**
 * 同一个构造函数创建的两个实例
 * 共享同一个原型对象：
 */
console.log(person1.__proto__ === person2.__proto__); // true

/**
 * instanceof检查实例的原型链中
 * 是否包含指定构造函数的原型：
 */
console.log(person1 instanceof Person);           // true
console.log(person1 instanceof Object);           // true
console.log(Person.prototype instanceof Object);  // true
```

对于前面例子中的`Person`构造函数和`Person.prototype`，可以通过图8-1看出各个对象之间的关系。

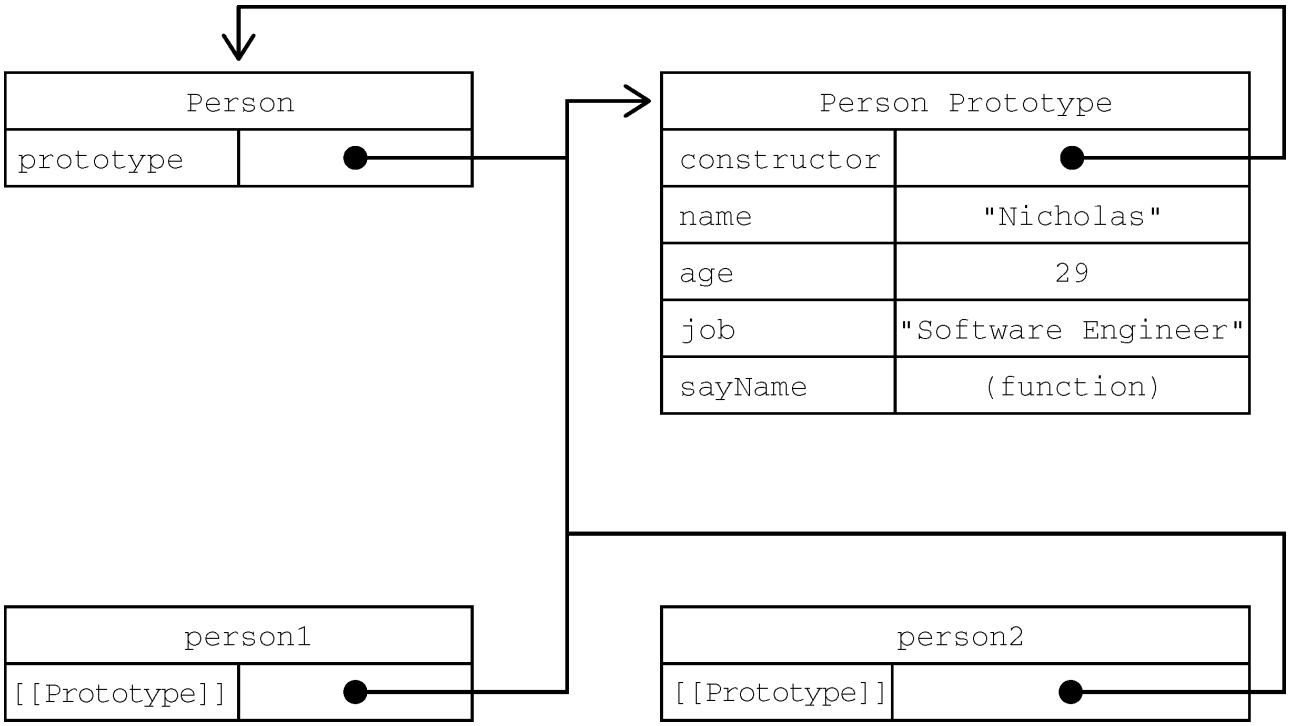


图 8-1

图8-1展示了Person构造函数、Person的原型对象和Person现有两个实例之间的关系。注意，Person.prototype指向原型对象，而Person.prototype.contructor指回Person构造函数。原型对象包含constructor属性和其他后来添加的属性。Person的两个实例person1和person2都只有一个内部属性指回Person.prototype，而且两者都与构造函数没有直接联系。另外要注意，虽然这两个实例都没有属性和方法，但person1.sayName()可以正常调用。这是由于对象属性查找机制的原因。

虽然不是所有实现都对外暴露了[[Prototype]]，但可以使用isPrototypeOf()方法确定两个对象之间的这种关系。本质上，isPrototypeOf()会在传入参数的[[Prototype]]指向调用它的对象时返回true，如下所示：

```
console.log(Person.prototype.isPrototypeOf(person1));  // true
console.log(Person.prototype.isPrototypeOf(person2));  // true
```

这里通过原型对象调用isPrototypeOf()方法检查了person1和person2。因为这两个例子内部都有链接指向Person.prototype，所以结果都返回true。

ECMAScript的Object类型有一个方法叫Object.getPrototypeOf()，返回参数的内部特性[[Prototype]]的值。例如：

```
console.log(Object.getPrototypeOf(person1) == Person.prototype);  // true
console.log(Object.getPrototypeOf(person1).name);                 //
"Nicholas"
```

第一行代码简单确认了Object.getPrototypeOf()返回的对象就是传入对象的原型对象。第二行代码则取得了原型对象上name属性的值，即"Nicholas"。使用Object.getPrototypeOf()可以方便地取得一个对象的原型，而这在通过原型实现继承时显得尤为重要（本章后面会介绍）。

Object类型还有一个setPrototypeOf()方法，可以向实例的私有特性[[Prototype]]写入一个新值。这样就可以重写一个对象的原型继承关系：

```
let biped = {
  numLegs: 2
};
let person = {
  name: 'Matt'
};

Object.setPrototypeOf(person, biped);

console.log(person.name);                                // Matt
console.log(person.numLegs);                             // 2
console.log(Object.getPrototypeOf(person) === biped);  // true
```

> **警告** Object.setPrototypeOf()可能会严重影响代码性能。Mozilla文档说得很清楚：“在所有浏览器和JavaScript引擎中，修改继承关系的影响都是微妙且深远的。这种影响并不仅是执行Object.setPrototypeOf()语句那么简单，而是会涉及所有访问了那些修改过[[Prototype]]的对象的代码。”

为避免使用Object.setPrototypeOf()可能造成的性能下降，可以通过Object.create()来创建一个新对象，同时为其指定原型：

```
let biped = {
  numLegs: 2
};
let person = Object.create(biped);
person.name = 'Matt';

console.log(person.name);                                // Matt
console.log(person.numLegs);                             // 2
console.log(Object.getPrototypeOf(person) === biped);  // true
```

2. **原型层级**

在通过对象访问属性时，会按照这个属性的名称开始搜索。搜索开始于对象实例本身。如果在这个实例上发现了给定的名称，则返回该名称对应的值。如果没有找到这个属性，则搜索会沿着指针进入原型对象，然后在原型对象上找到属性后，再返回对应的值。因此，在调用person1.sayName()时，会发生两步搜索。首先，JavaScript引擎会问：“person1实例有sayName属性吗？”答案是没有。然后，继续搜索并问：“person1的原型有sayName属性吗？”答案是有。于是就返回了保存在原型上的这个函数。在调用person2.sayName()时，会发生同样的搜索过程，而且也会返回相同的结果。这就是原型用于在多个对象实例间共享属性和方法的原理。

> **注意** 前面提到的constructor属性只存在于原型对象，因此通过实例对象也是可以访问到的。

虽然可以通过实例读取原型对象上的值，但不可能通过实例重写这些值。如果在实例上添加了一个与原型对象中同名的属性，那就会在实例上创建这个属性，这个属性会遮住原型对象上的属性。下面看一个

例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

person1.name = "Greg";
console.log(person1.name);  // "Greg", 来自实例
console.log(person2.name);  // "Nicholas", 来自原型
```

在这个例子中，person1的name属性遮蔽了原型对象上的同名属性。虽然person1.name和person2.name都返回了值，但前者返回的是"Greg"（来自实例），后者返回的是"Nicholas"（来自原型）。当console.log()访问person1.name时，会先在实例上搜索个属性。因为这个属性在实例上存在，所以就不会再搜索原型对象了。而在访问person2.name时，并没有在实例上找到这个属性，所以会继续搜索原型对象并使用定义在原型上的属性。

只要给对象实例添加一个属性，这个属性就会**遮蔽**（shadow）原型对象上的同名属性，也就是虽然不会修改它，但会屏蔽对它的访问。即使在实例上把这个属性设置为null，也不会恢复它和原型的联系。不过，使用delete操作符可以完全删除实例上的这个属性，从而让标识符解析过程能够继续搜索原型对象。

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

person1.name = "Greg";
console.log(person1.name);  // "Greg", 来自实例
console.log(person2.name);  // "Nicholas", 来自原型

delete person1.name;
console.log(person1.name);  // "Nicholas", 来自原型
```

这个修改后的例子中使用delete删除了person1.name，这个属性之前以"Greg"遮蔽了原型上的同名属性。然后原型上name属性的联系就恢复了，因此再访问person1.name时，就会返回原型对象上这个属性的值。

hasOwnProperty()方法用于确定某个属性是在实例上还是在原型对象上。这个方法是继承自Object的，会在属性存在于调用它的对象实例上时返回true，如下面的例子所示：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();
console.log(person1.hasOwnProperty("name")); // false

person1.name = "Greg";
console.log(person1.name); // "Greg", 来自实例
console.log(person1.hasOwnProperty("name")); // true

console.log(person2.name); // "Nicholas", 来自原型
console.log(person2.hasOwnProperty("name")); // false

delete person1.name;
console.log(person1.name); // "Nicholas", 来自原型
console.log(person1.hasOwnProperty("name")); // false
```

在这个例子中，通过调用hasOwnProperty()能够清楚地看到访问的是实例属性还是原型属性。调用person1.hasOwnProperty("name")只在重写person1上name属性的情况下才返回true，表明此时name是一个实例属性，不是原型属性。图8-2形象地展示了上面例子中各个步骤的状态。（为简单起见，图中省略了Person构造函数。）

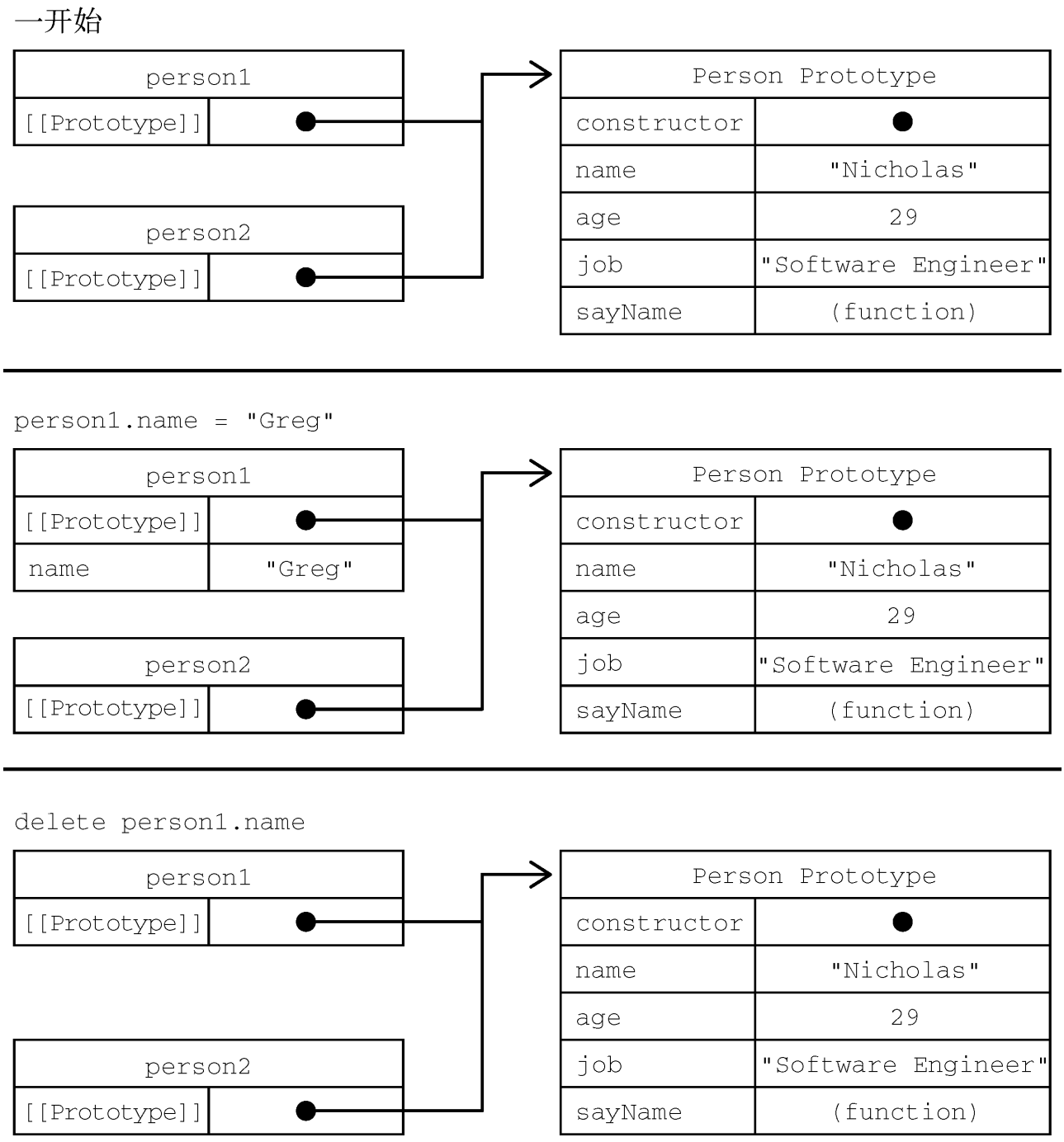


图 8-2

> **注意** ECMAScript的`Object.getOwnPropertyDescriptor()`方法只对实例属性有效。要取得原型属性的描述符，就必须直接在原型对象上调用`Object.getOwnPropertyDescriptor()`。

3. 原型和`in`操作符

有两种方式使用`in`操作符：单独使用和在`for-in`循环中使用。在单独使用时，`in`操作符会在可以通过对象访问指定属性时返回`true`，无论该属性是在实例上还是在原型上。来看下面的例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
```

```
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person1 = new Person();
let person2 = new Person();

console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true

person1.name = "Greg";
console.log(person1.name); // "Greg", 来自实例
console.log(person1.hasOwnProperty("name")); // true
console.log("name" in person1); // true

console.log(person2.name); // "Nicholas", 来自原型
console.log(person2.hasOwnProperty("name")); // false
console.log("name" in person2); // true

delete person1.name;
console.log(person1.name); // "Nicholas", 来自原型
console.log(person1.hasOwnProperty("name")); // false
console.log("name" in person1); // true
```

在上面整个例子中，`name`随时可以通过实例或通过原型访问到。因此，调用`"name" in persoon1`时始终返回`true`，无论这个属性是否在实例上。如果要确定某个属性是否存在于原型上，则可以像下面这样同时使用`hasOwnProperty()`和`in`操作符：

```
function hasPrototypeProperty(object, name){
  return !object.hasOwnProperty(name) && (name in object);
}
```

只要通过对象可以访问，`in`操作符就返回`true`，而`hasOwnProperty()`只有属性存在于实例上时才返回`true`。因此，只要`in`操作符返回`true`且`hasOwnProperty()`返回`false`，就说明该属性是一个原型属性。来看下面的例子：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let person = new Person();
console.log(hasPrototypeProperty(person, "name")); // true
```

```
person.name = "Greg";
console.log(hasPrototypeProperty(person, "name")); // false
```

在这里，name属性首先只存在于原型上，所以hasPrototypeProperty()返回true。而在实例上重写这个属性后，实例上也有了这个属性，因此hasPrototypeProperty()返回false。即便此时原型对象还有name属性，但因为实例上的属性遮蔽了它，所以不会用到。

在for-in循环中使用in操作符时，可以通过对象访问且可以被枚举的属性都会返回，包括实例属性和原型属性。遮蔽原型中不可枚举（[[Enumerable]]特性被设置为false）属性的实例属性也会在for-in循环中返回，因为默认情况下开发者定义的属性都是可枚举的。

要获得对象上所有可枚举的实例属性，可以使用Object.keys()方法。这个方法接收一个对象作为参数，返回包含该对象所有可枚举属性名称的字符串数组。比如：

```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function() {
  console.log(this.name);
};

let keys = Object.keys(Person.prototype);
console.log(keys);   // "name,age,job,sayName"
let p1 = new Person();
p1.name = "Rob";
p1.age = 31;
let p1keys = Object.keys(p1);
console.log(p1keys); // "name,age"
```

这里，keys变量保存的数组中包含"name"、"age"、"job"和"sayName"。这是正常情况下通过for-in返回的顺序。而在Person的实例上调用时，Object.keys()返回的数组中只包含"name"和"age"两个属性。

如果想列出所有实例属性，无论是否可以枚举，都可以使用Object.getOwnPropertyNames()：

```
let keys = Object.getOwnPropertyNames(Person.prototype);
console.log(keys);   // "constructor,name,age,job,sayName"
```

注意，返回的结果中包含了一个不可枚举的属性constructor。Object.keys()和Object.getOwnPropertyNames()在适当的时候都可用来代替for-in循环。

在ECMAScript 6新增符号类型之后，相应地出现了增加一个Object.getOwnPropertyNames()的兄弟方法的需求，因为以符号为键的属性没有名称的概念。因此，Object.getOwnPropertySymbols()方法就出现了，这个方法与Object.getOwnPropertyNames()类似，只是针对符号而已：

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');

let o = {
  [k1]: 'k1',
  [k2]: 'k2'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

4. **属性枚举顺序**

   `for-in`循环、`Object.keys()`、`Object.getOwnPropertyNames()`、`Object.getOwnPropertySymbols()`以及`Object.assign()`在属性枚举顺序方面有很大区别。`for-in`循环和`Object.keys()`的枚举顺序是不确定的，取决于JavaScript引擎，可能因浏览器而异。

   `Object.getOwnPropertyNames()`、`Object.getOwnPropertySymbols()`和`Object.assign()`的枚举顺序是确定性的。先以升序枚举数值键，然后以插入顺序枚举字符串和符号键。在对象字面量中定义的键以它们逗号分隔的顺序插入。

```
let k1 = Symbol('k1'),
    k2 = Symbol('k2');

let o = {
  1: 1,
  first: 'first',
  [k1]: 'sym2',
  second: 'second',
  0: 0
};

o[k2] = 'sym2';
o[3] = 3;
o.third = 'third';
o[2] = 2;

console.log(Object.getOwnPropertyNames(o));
// ["0", "1", "2", "3", "first", "second", "third"]

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(k1), Symbol(k2)]
```

8.2.5 对象迭代

在JavaScript有史以来的大部分时间内，迭代对象属性都是一个难题。ECMAScript 2017新增了两个静态方法，用于将对象内容转换为序列化的——更重要的是可迭代的——格式。这两个静态方法`Object.values()`和

`Object.entries()`接收一个对象，返回它们内容的数组。`Object.values()`返回对象值的数组，`Object.entries()`返回键/值对的数组。

下面的示例展示了这两个方法：

```
const o = {
  foo: 'bar',
  baz: 1,
  qux: {}
};

console.log(Object.values(o));
// ["bar", 1, {}]

console.log(Object.entries((o)));
// [["foo", "bar"], ["baz", 1], ["qux", {}]]
```

注意，非字符串属性会被转换为字符串输出。另外，这两个方法执行对象的浅复制：

```
const o = {
  qux: {}
};

console.log(Object.values(o)[0] === o.qux);
// true

console.log(Object.entries(o)[0][1] === o.qux);
// true
```

符号属性会被忽略：

```
const sym = Symbol();
const o = {
  [sym]: 'foo'
};

console.log(Object.values(o));
// []

console.log(Object.entries((o)));
// []
```

1. **其他原型语法**

   有读者可能注意到了，在前面的例子中，每次定义一个属性或方法都会把`Person.prototype`重写一遍。为了减少代码冗余，也为了从视觉上更好地封装原型功能，直接通过一个包含所有属性和方法的对象字面量来重写原型成为了一种常见的做法，如下面的例子所示：

```
function Person() {}

Person.prototype = {
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};
```

在这个例子中，Person.prototype被设置为等于一个通过对象字面量创建的新对象。最终结果是一样的，只有一个问题：这样重写之后，Person.prototype的constructor属性就不指向Person了。在创建函数时，也会创建它的prototype对象，同时会自动给这个原型的constructor属性赋值。而上面的写法完全重写了默认的prototype对象，因此其constructor属性也指向了完全不同的新对象（Object构造函数），不再指向原来的构造函数。虽然instanceof操作符还能可靠地返回值，但我们不能再依靠constructor属性来识别类型了，如下面的例子所示：

```
let friend = new Person();

console.log(friend instanceof Object);      // true
console.log(friend instanceof Person);      // true
console.log(friend.constructor == Person);  // false
console.log(friend.constructor == Object);  // true
```

这里，instanceof仍然对Object和Person都返回true。但constructor属性现在等于Object而不是Person了。如果constructor的值很重要，则可以像下面这样在重写原型对象时专门设置一下它的值：

```
function Person() {
}

Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};
```

这次的代码中特意包含了constructor属性，并将它设置为Person，保证了这个属性仍然包含恰当的值。

但要注意，以这种方式恢复constructor属性会创建一个[[Enumberable]]为true的属性。而原生constructor属性默认是不可枚举的。因此，如果你使用的是兼容ECMAScript的JavaScript引擎，那可能

会改为使用Object.defineProperty()方法来定义constructor属性：

```
function Person() {}

Person.prototype = {
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};

// 恢复constructor属性
Object.defineProperty(Person.prototype, "constructor", {
  enumerable: false,
  value: Person
});
```

2. **原型的动态性**

因为从原型上搜索值的过程是动态的，所以即使实例在修改原型之前已经存在，任何时候对原型对象所做的修改也会在实例上反映出来。下面是一个例子：

```
let friend = new Person();

Person.prototype.sayHi = function() {
  console.log("hi");
};

friend.sayHi();   // "hi", 没问题!
```

以上代码先创建一个Person实例并保存在friend中。然后一条语句在Person.prototype上添加了一个名为sayHi()的方法。虽然friend实例是在添加方法之前创建的，但它仍然可以访问这个方法。之所以会这样，主要原因是实例与原型之间松散的联系。在调用friend.sayHi()时，首先会从这个实例中搜索名为sayHi的属性。在没有找到的情况下，运行时会继续搜索原型对象。因为实例和原型之间的链接就是简单的指针，而不是保存的副本，所以会在原型上找到sayHi属性并返回这个属性保存的函数。

虽然随时能给原型添加属性和方法，并能够立即反映在所有对象实例上，但这跟重写整个原型是两回事。实例的[[Prototype]]指针是在调用构造函数时自动赋值的，这个指针即使把原型修改为不同的对象也不会变。重写整个原型会切断最初原型与构造函数的联系，但实例引用的仍然是最初的原型。记住，实例只有指向原型的指针，没有指向构造函数的指针。来看下面的例子：

```
function Person() {}

let friend = new Person();
Person.prototype = {
```

```
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  sayName() {
    console.log(this.name);
  }
};

friend.sayName();  // 错误
```

在这个例子中，`Person`的新实例是在重写原型对象之前创建的。在调用`friend.sayName()`的时候，会导致错误。这是因为`firend`指向的原型还是最初的原型，而这个原型上并没有`sayName`属性。图8-3展示了这里面的原因。

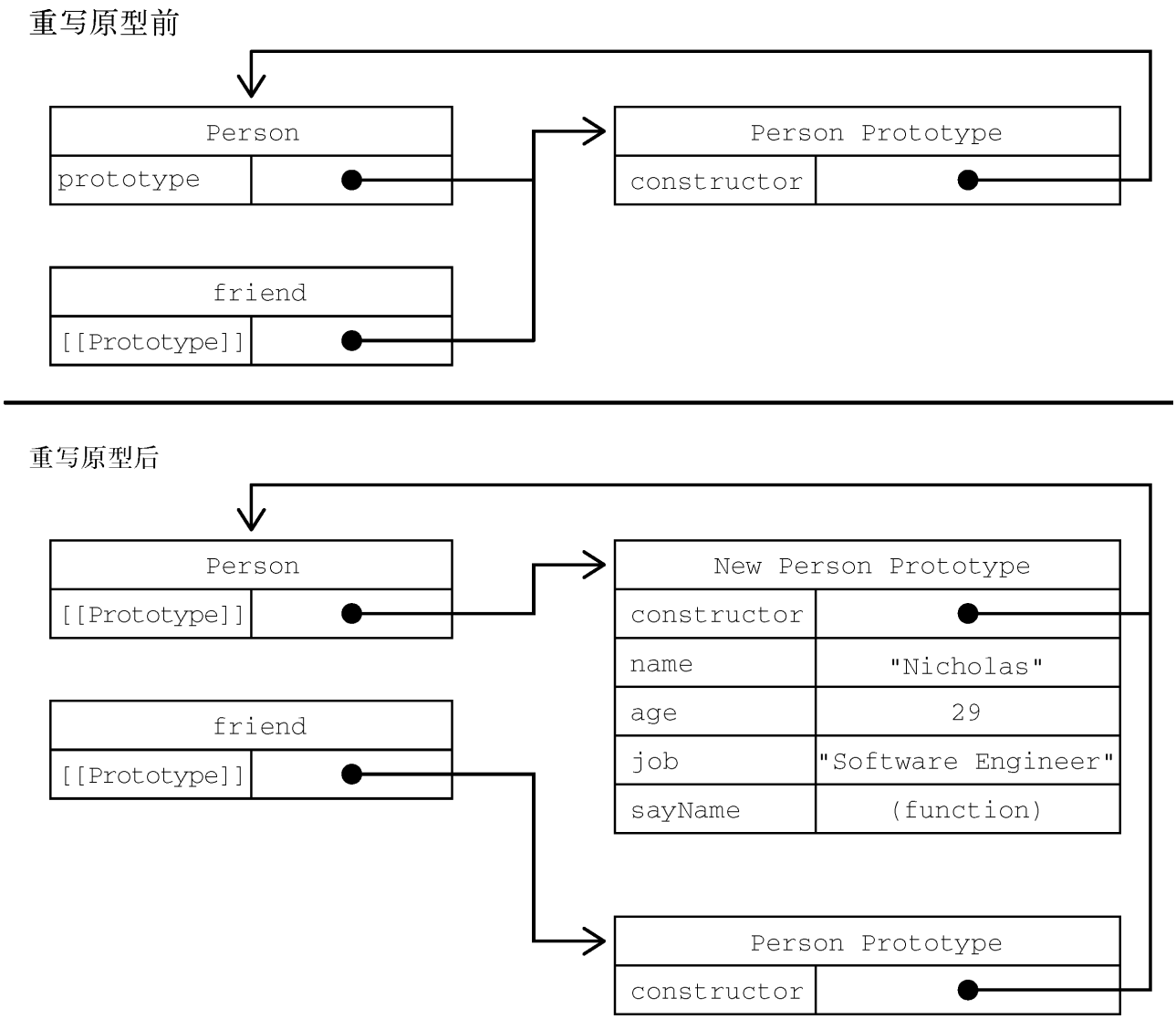


图 8-3

重写构造函数上的原型之后再创建的实例才会引用新的原型。而在此之前创建的实例仍然会引用最初的原型。

3. **原生对象原型**

原型模式之所以重要，不仅体现在自定义类型上，而且还因为它也是实现所有原生引用类型的模式。所有原生引用类型的构造函数（包括Object、Array、String等）都在原型上定义了实例方法。比如，数组实例的sort()方法就是Array.prototype上定义的，而字符串包装对象的substring()方法也是在String.prototype上定义的，如下所示：

```
console.log(typeof Array.prototype.sort);       // "function"
console.log(typeof String.prototype.substring); // "function"
```

通过原生对象的原型可以取得所有默认方法的引用，也可以给原生类型的实例定义新的方法。可以像修改自定义对象原型一样修改原生对象原型，因此随时可以添加方法。比如，下面的代码就给String原始值包装类型的实例添加了一个startsWith()方法：

```
String.prototype.startsWith = function (text) {
  return this.indexOf(text) === 0;
};

let msg = "Hello world!";
console.log(msg.startsWith("Hello"));  // true
```

如果给定字符串的开头出现了调用startsWith()方法的文本，那么该方法会返回true。因为这个方法是被定义在String.prototype上，所以当前环境下所有的字符串都可以使用这个方法。msg是个字符串，在读取它的属性时，后台会自动创建String的包装实例，从而找到并调用startsWith()方法。

> **注意** 尽管可以这么做，但并不推荐在产品环境中修改原生对象原型。这样做很可能造成误会，而且可能引发命名冲突（比如一个名称在某个浏览器实现中不存在，在另一个实现中却存在）。另外还有可能意外重写原生的方法。推荐的做法是创建一个自定义的类，继承原生类型。

4. **原型的问题**

原型模式也不是没有问题。首先，它弱化了向构造函数传递初始化参数的能力，会导致所有实例默认都取得相同的属性值。虽然这会带来不便，但还不是原型的最大问题。原型的最主要问题源自它的共享特性。

我们知道，原型上的所有属性是在实例间共享的，这对函数来说比较合适。另外包含原始值的属性也还好，如前面例子中所示，可以通过在实例上添加同名属性来简单地遮蔽原型上的属性。真正的问题来自包含引用值的属性。来看下面的例子：

```
function Person() {}

Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 29,
  job: "Software Engineer",
  friends: ["Shelby", "Court"],
  sayName() {
    console.log(this.name);
```

```
    }
};

let person1 = new Person();
let person2 = new Person();

person1.friends.push("Van");

console.log(person1.friends);  // "Shelby,Court,Van"
console.log(person2.friends);  // "Shelby,Court,Van"
console.log(person1.friends === person2.friends);  // true
```

这里，Person.prototype有一个名为friends的属性，它包含一个字符串数组。然后这里创建了两个Person的实例。person1.friends通过push方法向数组中添加了一个字符串。由于这个friends属性存在于Person.prototype而非person1上，新加的这个字符串也会在（指向同一个数组的）person2.friends上反映出来。如果这是有意在多个实例间共享数组，那没什么问题。但一般来说，不同的实例应该有属于自己的属性副本。这就是实际开发中通常不单独使用原型模式的原因。

8.3 继承

继承是面向对象编程中讨论最多的话题。很多面向对象语言都支持两种继承：接口继承和实现继承。前者只继承方法签名，后者继承实际的方法。接口继承在ECMAScript中是不可能的，因为函数没有签名。实现继承是ECMAScript唯一支持的继承方式，而这主要是通过原型链实现的。

8.3.1 原型链

ECMA-262把**原型链**定义为ECMAScript的主要继承方式。其基本思想就是通过原型继承多个引用类型的属性和方法。重温一下构造函数、原型和实例的关系：每个构造函数都有一个原型对象，原型有一个属性指回构造函数，而实例有一个内部指针指向原型。如果原型是另一个类型的实例呢？那就意味着这个原型本身有一个内部指针指向另一个原型，相应地另一个原型也有一个指针指向另一个构造函数。这样就在实例和原型之间构造了一条原型链。这就是原型链的基本构想。

实现原型链涉及如下代码模式：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}

// 继承SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function () {
```

```
    return this.subproperty;
};

let instance = new SubType();
console.log(instance.getSuperValue()); // true
```

以上代码定义了两个类型：SuperType和SubType。这两个类型分别定义了一个属性和一个方法。这两个类型的主要区别是SubType通过创建SuperType的实例并将其赋值给自己的原型SubTtype.prototype实现了对SuperType的继承。这个赋值重写了SubType最初的原型，将其替换为SuperType的实例。这意味着SuperType实例可以访问的所有属性和方法也会存在于SubType.prototype。这样实现继承之后，代码紧接着又给SubType.prototype，也就是这个SuperType的实例添加了一个新方法。最后又创建了SubType的实例并调用了它继承的getSuperValue()方法。图8-4展示了子类的实例与两个构造函数及其对应的原型之间的关系。

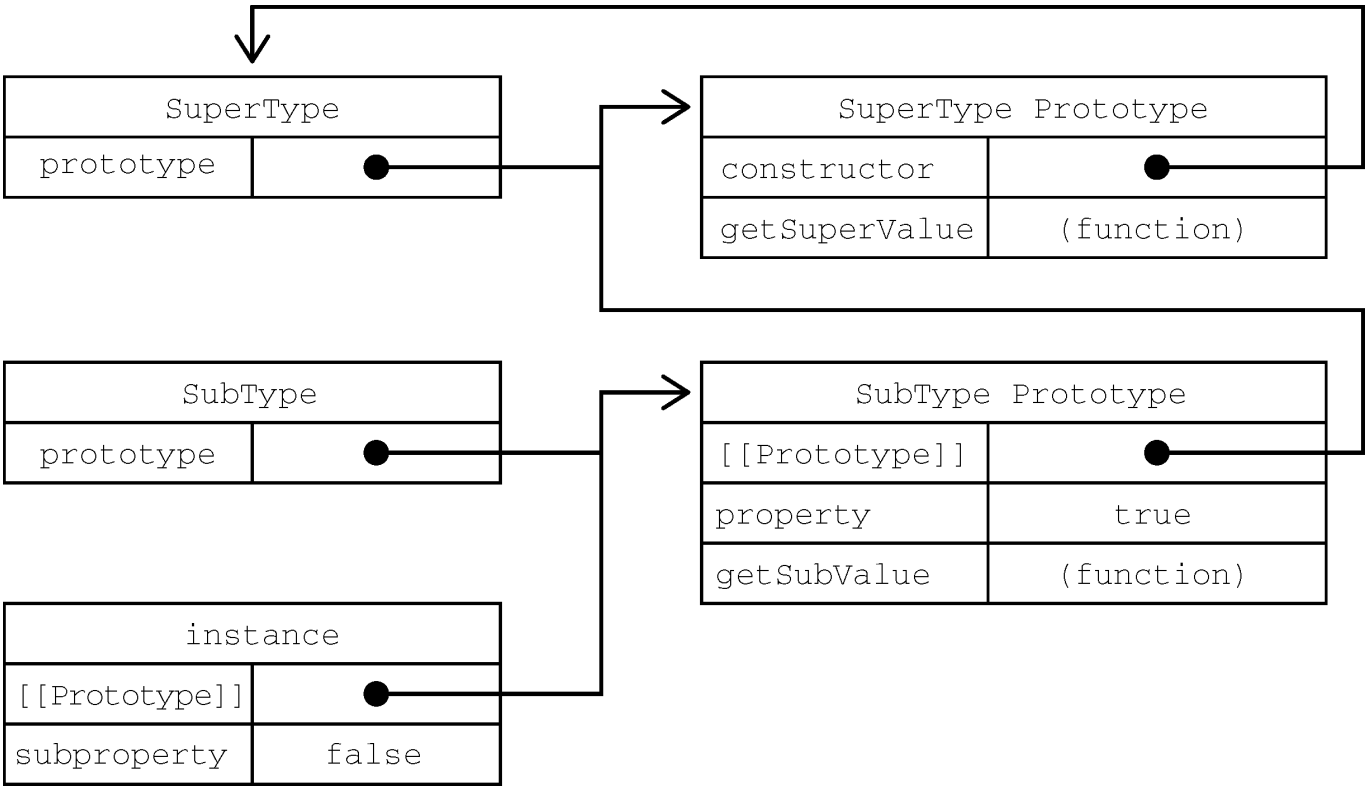


图 8-4

这个例子中实现继承的关键，是SubType没有使用默认原型，而是将其替换成了一个新的对象。这个新的对象恰好是SuperType的实例。这样一来，SubType的实例不仅能从SuperType的实例中继承属性和方法，而且还与SuperType的原型挂上了钩。于是instance（通过内部的[[Prototype]]）指向SubType.prototype，而SubType.prototype（作为SuperType的实例又通过内部的[[Prototype]]）指向SuperType.prototype。注意，getSuperValue()方法还在SuperType.prototype对象上，而property属性则在SubType.prototype上。这是因为getSuperValue()是一个原型方法，而property是一个实例属性。SubType.prototype现在是SuperType的一个实例，因此property才会存储在它上面。还要注意，由于SubType.prototype的constructor属性被重写为指向SuperType，所以instance.constructor也指向SuperType。

原型链扩展了前面描述的原型搜索机制。我们知道，在读取实例上的属性时，首先会在实例上搜索这个属性。如果没找到，则会继承搜索实例的原型。在通过原型链实现继承之后，搜索就可以继承向上，搜索原型的原型。对前面的例子而言，调用instance.getSuperValue()经过了3步搜索：instance、SubType.prototype和SuperType.prototype，最后一步才找到这个方法。对属性和方法的搜索会一直持续到原型链的末端。

1. **默认原型**

实际上，原型链中还有一环。默认情况下，所有引用类型都继承自`Object`，这也是通过原型链实现的。任何函数的默认原型都是一个`Object`的实例，这意味着这个实例有一个内部指针指向`Object.prototype`。这也是为什么自定义类型能够继承包括`toString()`、`valueOf()`在内的所有默认方法的原因。因此前面的例子还有额外一层继承关系。图8-5展示了完整的原型链。

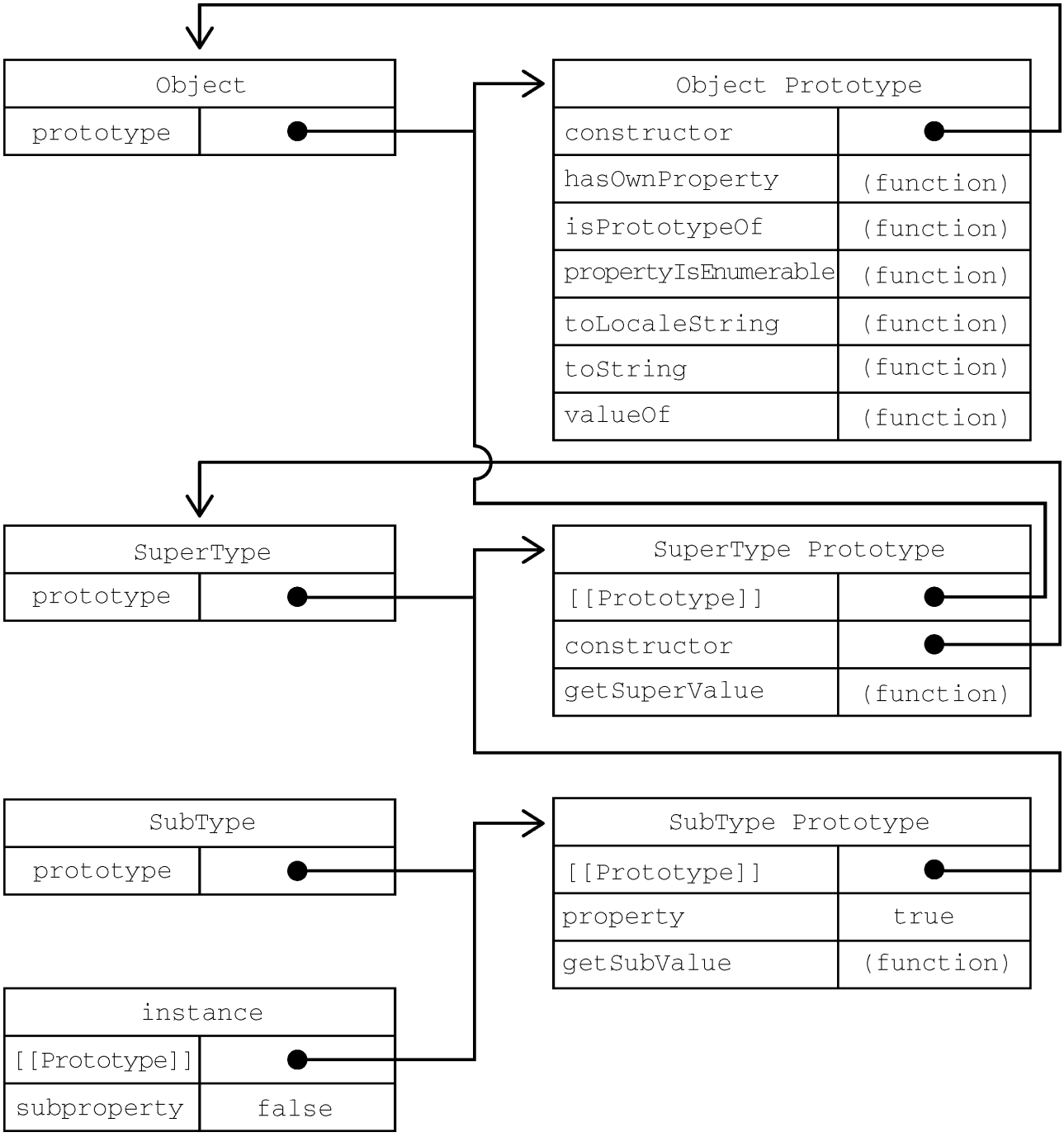


图　8-5

`SubType`继承`SuperType`，而`SuperType`继承`Object`。在调用`instance.toString()`时，实际上调用的是保存在`Object.prototype`上的方法。

2. **原型与继承关系**

原型与实例的关系可以通过两种方式来确定。第一种方式是使用`instanceof`操作符，如果一个实例的原型链中出现过相应的构造函数，则`instanceof`返回`true`。如下例所示：

```
console.log(instance instanceof Object);     // true
console.log(instance instanceof SuperType);  // true
console.log(instance instanceof SubType);    // true
```

从技术上讲，instance是Object、SuperType和SubType的实例，因为instance的原型链中包含这些构造函数的原型。结果就是instanceof对所有这些构造函数都返回true。

确定这种关系的第二种方式是使用isPrototypeOf()方法。原型链中的每个原型都可以调用这个方法，如下例所示，只要原型链中包含这个原型，这个方法就返回true：

```
console.log(Object.prototype.isPrototypeOf(instance));     // true
console.log(SuperType.prototype.isPrototypeOf(instance));  // true
console.log(SubType.prototype.isPrototypeOf(instance));    // true
```

3. **关于方法**

子类有时候需要覆盖父类的方法，或者增加父类没有的方法。为此，这些方法必须在原型赋值之后再添加到原型上。来看下面的例子：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}

// 继承SuperType
SubType.prototype = new SuperType();

// 新方法
SubType.prototype.getSubValue = function () {
  return this.subproperty;
};

// 覆盖已有的方法
SubType.prototype.getSuperValue = function () {
  return false;
};

let instance = new SubType();
console.log(instance.getSuperValue()); // false
```

在上面的代码中，加粗的部分涉及两个方法。第一个方法`getSubValue()`是`SubType`的新方法，而第二个方法`getSuperValue()`是原型链上已经存在但在这里被遮蔽的方法。后面在`SubType`实例上调用`getSuperValue()`时调用的是这个方法。而`SuperType`的实例仍然会调用最初的方法。重点在于上述两个方法都是在把原型赋值为`SuperType`的实例之后定义的。

另一个要理解的重点是，以对象字面量方式创建原型方法会破坏之前的原型链，因为这相当于重写了原型链。下面是一个例子：

```
function SuperType() {
  this.property = true;
}

SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subproperty = false;
}

// 继承SuperType
SubType.prototype = new SuperType();

// 通过对象字面量添加新方法，这会导致上一行无效
SubType.prototype = {
  getSubValue() {
    return this.subproperty;
  },

  someOtherMethod() {
    return false;
  }
};

let instance = new SubType();
console.log(instance.getSuperValue()); // 出错!
```

在这段代码中，子类的原型在被赋值为`SuperType`的实例后，又被一个对象字面量覆盖了。覆盖后的原型是一个`Object`的实例，而不再是`SuperType`的实例。因此之前的原型链就断了。`SubType`和`SuperType`之间也没有关系了。

4. **原型链的问题**

原型链虽然是实现继承的强大工具，但它也有问题。主要问题出现在原型中包含引用值的时候。前面在谈到原型的问题时也提到过，原型中包含的引用值会在所有实例间共享，这也是为什么属性通常会在构造函数中定义而不会定义在原型上的原因。在使用原型实现继承时，原型实际上变成了另一个类型的实例。这意味着原先的实例属性摇身一变成为了原型属性。下面的例子揭示了这个问题：

```
function SuperType() {
  this.colors = ["red", "blue", "green"];
}

function SubType() {}

// 继承SuperType
SubType.prototype = new SuperType();

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green,black"
```

在这个例子中，SuperType构造函数定义了一个colors属性，其中包含一个数组（引用值）。每个SuperType的实例都会有自己的colors属性，包含自己的数组。但是，当SubType通过原型继承SuperType后，SubType.prototype变成了SuperType的一个实例，因而也获得了自己的colors属性。这类似于创建了SubType.prototype.colors属性。最终结果是，SubType的所有实例都会共享这个colors属性。这一点通过instance1.colors上的修改也能反映到instance2.colors上就可以看出来。

原型链的第二个问题是，子类型在实例化时不能给父类型的构造函数传参。事实上，我们无法在不影响所有对象实例的情况下把参数传进父类的构造函数。再加上之前提到的原型中包含引用值的问题，就导致原型链基本不会被单独使用。

8.3.2 盗用构造函数

为了解决原型包含引用值导致的继承问题，一种叫作“盗用构造函数”（constructor stealing）的技术在开发社区流行起来（这种技术有时也称作“对象伪装”或“经典继承”）。基本思路很简单：在子类构造函数中调用父类构造函数。因为毕竟函数就是在特定上下文中执行代码的简单对象，所以可以使用apply()和call()方法以新创建的对象为上下文执行构造函数。来看下面的例子：

```
function SuperType() {
  this.colors = ["red", "blue", "green"];
}

function SubType() {
  // 继承SuperType
  SuperType.call(this);
}

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors); // "red,blue,green,black"

let instance2 = new SubType();
console.log(instance2.colors); // "red,blue,green"
```

示例中加粗的代码展示了盗用构造函数的调用。通过使用`call()`（或`apply()`）方法，`SuperType`构造函数在为`SubType`的实例创建的新对象的上下文中执行了。这相当于新的`SubType`对象上运行了`SuperType()`函数中的所有初始化代码。结果就是每个实例都会有自己的`colors`属性。

1. **传递参数**

   相比于使用原型链，盗用构造函数的一个优点就是可以在子类构造函数中向父类构造函数传参。来看下面的例子：

   ```
   function SuperType(name){
     this.name = name;
   }

   function SubType() {
     // 继承SuperType并传参
     SuperType.call(this, "Nicholas");

     // 实例属性
     this.age = 29;
   }

   let instance = new SubType();
   console.log(instance.name); // "Nicholas";
   console.log(instance.age);  // 29
   ```

   在这个例子中，`SuperType`构造函数接收一个参数`name`，然后将它赋值给一个属性。在`SubType`构造函数中调用`SuperType`构造函数时传入这个参数，实际上会在`SubType`的实例上定义`name`属性。为确保`SuperType`构造函数不会覆盖`SubType`定义的属性，可以在调用父类构造函数之后再给子类实例添加额外的属性。

2. **盗用构造函数的问题**

   盗用构造函数的主要缺点，也是使用构造函数模式自定义类型的问题：必须在构造函数中定义方法，因此函数不能重用。此外，子类也不能访问父类原型上定义的方法，因此所有类型只能使用构造函数模式。由于存在这些问题，盗用构造函数基本上也不能单独使用。

8.3.3 组合继承

组合继承（有时候也叫伪经典继承）综合了原型链和盗用构造函数，将两者的优点集中了起来。基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性。这样既可以把方法定义在原型上以实现重用，又可以让每个实例都有自己的属性。来看下面的例子：

```
function SuperType(name){
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
```

```
  console.log(this.name);
};

function SubType(name, age){
  // 继承属性
  SuperType.call(this, name);

  this.age = age;
}

// 继承方法
SubType.prototype = new SuperType();

SubType.prototype.sayAge = function() {
  console.log(this.age);
};

let instance1 = new SubType("Nicholas", 29);
instance1.colors.push("black");
console.log(instance1.colors);  // "red,blue,green,black"
instance1.sayName();            // "Nicholas";
instance1.sayAge();             // 29

let instance2 = new SubType("Greg", 27);
console.log(instance2.colors);  // "red,blue,green"
instance2.sayName();            // "Greg";
instance2.sayAge();             // 27
```

在这个例子中，SuperType构造函数定义了两个属性，name和colors，而它的原型上也定义了一个方法叫sayName()。SubType构造函数调用了SuperType构造函数，传入了name参数，然后又定义了自己的属性age。此外，SubType.prototype也被赋值为SuperType的实例。原型赋值之后，又在这个原型上添加了新方法sayAge()。这样，就可以创建两个SubType实例，让这两个实例都有自己的属性，包括colors，同时还共享相同的方法。

组合继承弥补了原型链和盗用构造函数的不足，是JavaScript中使用最多的继承模式。而且组合继承也保留了instanceof操作符和isPrototypeOf()方法识别合成对象的能力。

8.3.4 原型式继承

2006年，Douglas Crockford写了一篇文章：《JavaScript中的原型式继承》（“Prototypal Inheritance in JavaScript”）。这篇文章介绍了一种不涉及严格意义上构造函数的继承方法。他的出发点是即使不自定义类型也可以通过原型实现对象之间的信息共享。文章最终给出了一个函数：

```
function object(o) {
  function F() {}
  F.prototype = o;
  return new F();
}
```

这个object()函数会创建一个临时构造函数，将传入的对象赋值给这个构造函数的原型，然后返回这个临时类型的一个实例。本质上，object()是对传入的对象执行了一次浅复制。来看下面的例子：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);  // "Shelby,Court,Van,Rob,Barbie"
```

Crockford推荐的原型式继承适用于这种情况：你有一个对象，想在它的基础上再创建一个新对象。你需要把这个对象先传给object()，然后再对返回的对象进行适当修改。在这个例子中，person对象定义了另一个对象也应该共享的信息，把它传给object()之后会返回一个新对象。这个新对象的原型是person，意味着它的原型上既有原始值属性又有引用值属性。这也意味着person.friends不仅是person的属性，也会跟anotherPerson和yetAnotherPerson共享。这里实际上克隆了两个person。

ECMAScript 5通过增加Object.create()方法将原型式继承的概念规范化了。这个方法接收两个参数：作为新对象原型的对象，以及给新对象定义额外属性的对象（第二个可选）。在只有一个参数时，Object.create()与这里的object()方法效果相同：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

let yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);  // "Shelby,Court,Van,Rob,Barbie"
```

Object.create()的第二个参数与Object.defineProperties()的第二个参数一样：每个新增属性都通过各自的描述符来描述。以这种方式添加的属性会遮蔽原型对象上的同名属性。比如：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = Object.create(person, {
  name: {
    value: "Greg"
  }
});
console.log(anotherPerson.name);  // "Greg"
```

原型式继承非常适合不需要单独创建构造函数，但仍然需要在对象间共享信息的场合。但要记住，属性中包含的引用值始终会在相关对象间共享，跟使用原型模式是一样的。

8.3.5 寄生式继承

与原型式继承比较接近的一种继承方式是**寄生式继承**（parasitic inheritance），也是Crockford首倡的一种模式。寄生式继承背后的思路类似于寄生构造函数和工厂模式：创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象。基本的寄生继承模式如下：

```
function createAnother(original){
  let clone = object(original);  // 通过调用函数创建一个新对象
  clone.sayHi = function() {     // 以某种方式增强这个对象
    console.log("hi");
  };
  return clone;           // 返回这个对象
}
```

在这段代码中，`createAnother()`函数接收一个参数，就是新对象的基准对象。这个对象`original`会被传给`object()`函数，然后将返回的新对象赋值给`clone`。接着给`clone`对象添加一个新方法`sayHi()`。最后返回这个对象。可以像下面这样使用`createAnother()`函数：

```
let person = {
  name: "Nicholas",
  friends: ["Shelby", "Court", "Van"]
};

let anotherPerson = createAnother(person);
anotherPerson.sayHi();  // "hi"
```

这个例子基于`person`对象返回了一个新对象。新返回的`anotherPerson`对象具有`person`的所有属性和方法，还有一个新方法叫`sayHi()`。

寄生式继承同样适合主要关注对象，而不在乎类型和构造函数的场景。`object()`函数不是寄生式继承所必需的，任何返回新对象的函数都可以在这里使用。

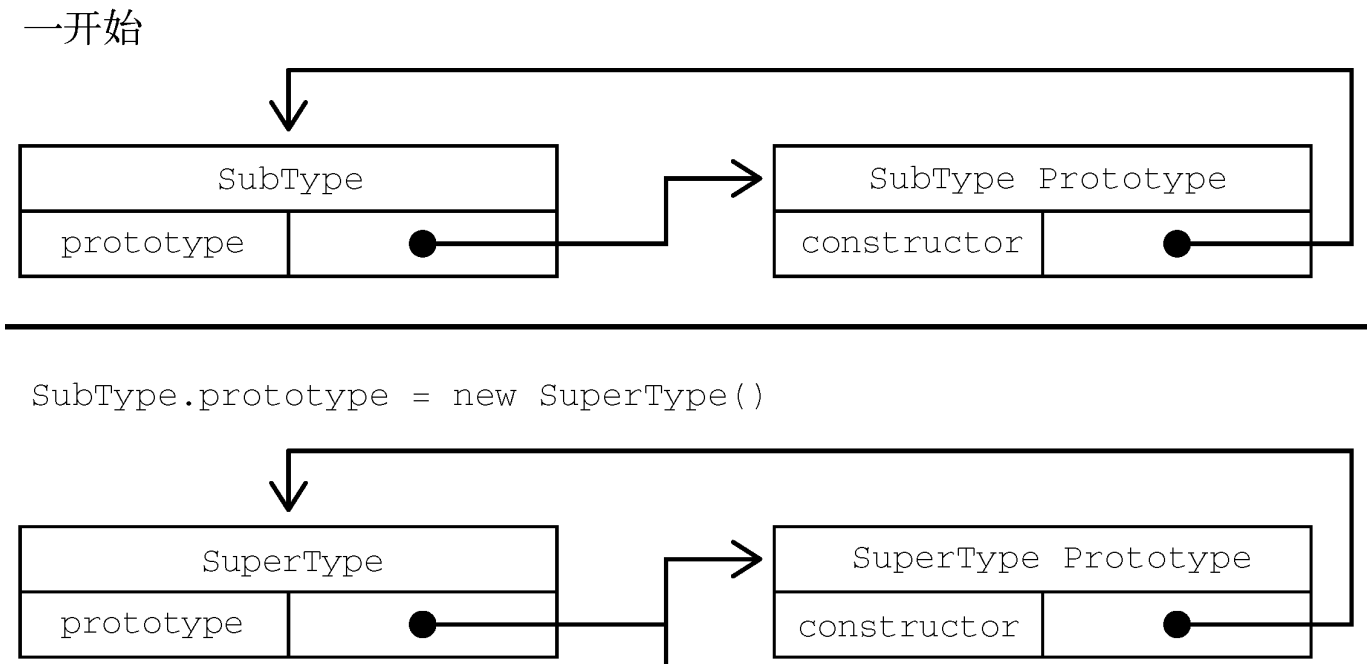
> **注意** 通过寄生式继承给对象添加函数会导致函数难以重用，与构造函数模式类似。

8.3.6 寄生式组合继承

组合继承其实也存在效率问题。最主要的效率问题就是父类构造函数始终会被调用两次：一次在是创建子类原型时调用，另一次是在子类构造函数中调用。本质上，子类原型最终是要包含超类对象的所有实例属性，子类构造函数只要在执行时重写自己的原型就行了。再来看一看这个组合继承的例子：

```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
  console.log(this.name);
};

function SubType(name, age){
  SuperType.call(this, name);   // 第二次调用SuperType()

  this.age = age;
}

SubType.prototype = new SuperType();   // 第一次调用SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function() {
  console.log(this.age);
};
```

代码中加粗的部分是调用SuperType构造函数的地方。在上面的代码执行后，SubType.prototype上会有两个属性：name和colors。它们都是SuperType的实例属性，但现在成为了SubType的原型属性。在调用SubType构造函数时，也会调用SuperType构造函数，这一次会在新对象上创建实例属性name和colors。这两个实例属性会遮蔽原型上同名的属性。图8-6展示了这个过程。



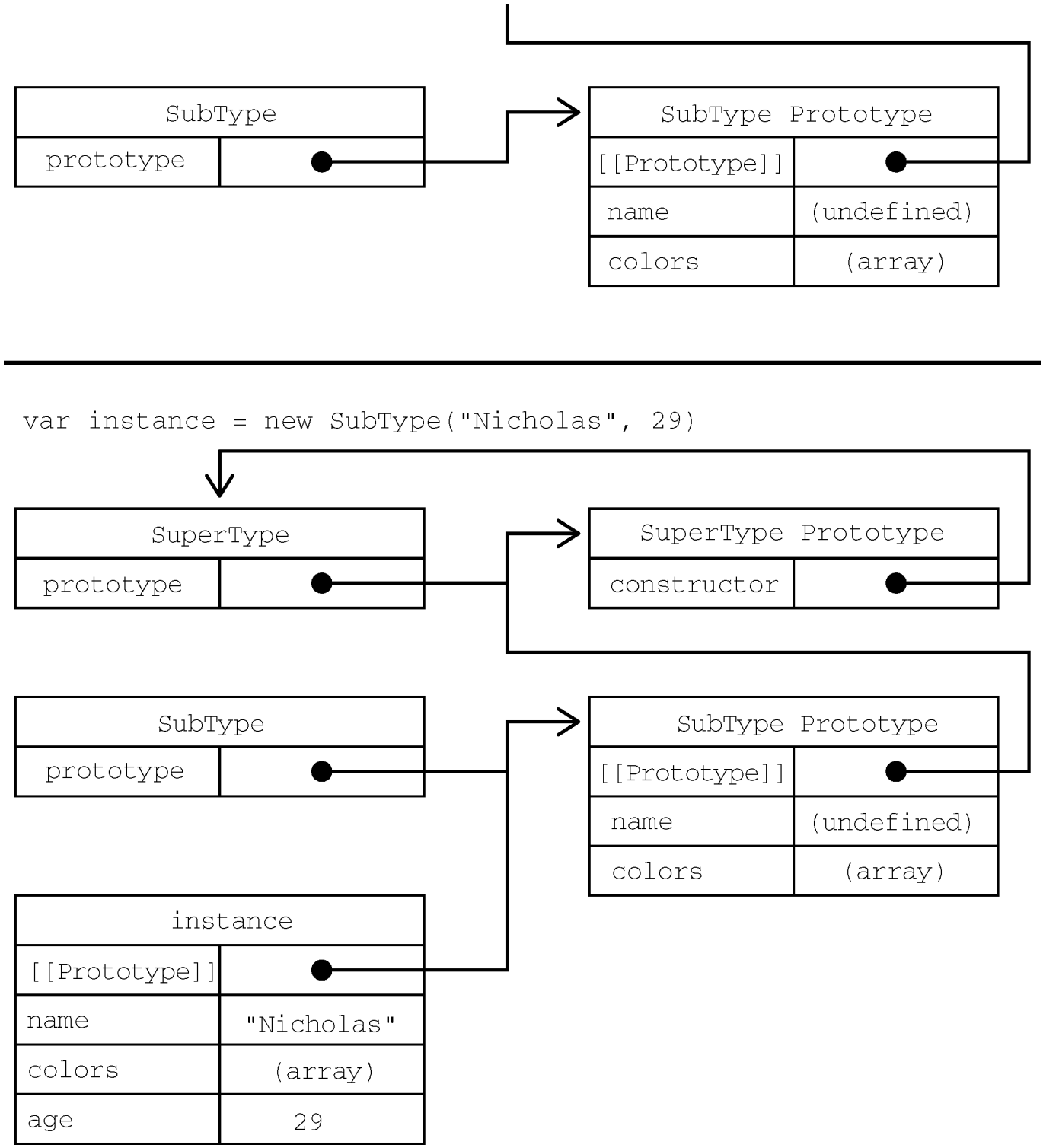


图 8-6

如图8-6所示，有两组name和colors属性：一组在实例上，另一组在SubType的原型上。这是调用两次SuperType构造函数的结果。好在有办法解决这个问题。

寄生式组合继承通过盗用构造函数继承属性，但使用混合式原型链继承方法。基本思路是不通过调用父类构造函数给子类原型赋值，而是取得父类原型的一个副本。说到底就是使用寄生式继承来继承父类原型，然后将返回的新对象赋值给子类原型。寄生式组合继承的基本模式如下所示：

```
function inheritPrototype(subType, superType) {
  let prototype = object(superType.prototype);  // 创建对象
  prototype.constructor = subType;              // 增强对象
```

```
  subType.prototype = prototype;                    // 赋值对象
}
```

这个inheritPrototype()函数实现了寄生式组合继承的核心逻辑。这个函数接收两个参数：子类构造函数和父类构造函数。在这个函数内部，第一步是创建父类原型的一个副本。然后，给返回的prototype对象设置constructor属性，解决由于重写原型导致默认constructor丢失的问题。最后将新创建的对象赋值给子类型的原型。如下例所示，调用inheritPrototype()就可以实现前面例子中的子类型原型赋值：

```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
  console.log(this.name);
};

function SubType(name, age) {
  SuperType.call(this, name);

  this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function() {
  console.log(this.age);
};
```

这里只调用了一次SuperType构造函数，避免了SubType.prototype上不必要也用不到的属性，因此可以说这个例子的效率更高。而且，原型键仍然保持不变，因此instanceof操作符和isPrototypeOf()方法正常有效。寄生式组合继承可以算是引用类型继承的最佳模式。

8.4 类

前几节深入讲解了如何只使用ECMAScript 5的特性来模拟类似于类（class-like）的行为。不难看出，各种策略都有自己的问题，也有相应的妥协。正因为如此，实现继承的代码也显得非常冗长和混乱。

为解决这些问题，ECMAScript 6新引入的class关键字具有正式定义类的能力。类（class）是ECMAScript中新的基础性语法糖结构，因此刚开始接触时可能会不太习惯。虽然ECMAScript 6类表面上看起来可以支持正式的面向对象编程，但实际上它背后使用的仍然是原型和构造函数的概念。

8.4.1 类定义

与函数类型相似，定义类也有两种主要方式：类声明和类表达式。这两种方式都使用class关键字加大括号：

```
// 类声明
class Person {}
```

```
// 类表达式
const Animal = class {};
```

与函数表达式类似，类表达式在它们被求值前也不能引用。不过，与函数定义不同的是，虽然函数声明可以提升，但类定义不能：

```
console.log(FunctionExpression);   // undefined
var FunctionExpression = function() {};
console.log(FunctionExpression);   // function() {}

console.log(FunctionDeclaration);  // FunctionDeclaration() {}
function FunctionDeclaration() {}
console.log(FunctionDeclaration);  // FunctionDeclaration() {}

console.log(ClassExpression);      // undefined
var ClassExpression = class {};
console.log(ClassExpression);      // class {}

console.log(ClassDeclaration);     // ReferenceError: ClassDeclaration is not
defined
class ClassDeclaration {}
console.log(ClassDeclaration);     // class ClassDeclaration {}
```

另一个跟函数声明不同的地方是，函数受函数作用域限制，而类受块作用域限制：

```
{
  function FunctionDeclaration() {}
  class ClassDeclaration {}
}

console.log(FunctionDeclaration); // FunctionDeclaration() {}
console.log(ClassDeclaration);    // ReferenceError: ClassDeclaration is not
defined
```

类的构成

类可以包含构造函数方法、实例方法、获取函数、设置函数和静态类方法，但这些都不是必需的。空的类定义照样有效。默认情况下，类定义中的代码都在严格模式下执行。

与函数构造函数一样，多数编程风格都建议类名的首字母要大写，以区别于通过它创建的实例（比如，通过`class Foo {}`创建实例`foo`）：

```
// 空类定义，有效
class Foo {}

// 有构造函数的类，有效
```

```
class Bar {
  constructor() {}
}

// 有获取函数的类，有效
class Baz {
  get myBaz() {}
}

// 有静态方法的类，有效
class Qux {
  static myQux() {}
}
```

类表达式的名称是可选的。在把类表达式赋值给变量后，可以通过name属性取得类表达式的名称字符串。但不能在类表达式作用域外部访问这个标识符。

```
let Person = class PersonName {
  identify() {
    console.log(Person.name, PersonName.name);
  }
}

let p = new Person();

p.identify();               // PersonName PersonName

console.log(Person.name);   // PersonName
console.log(PersonName);    // ReferenceError: PersonName is not defined
```

8.4.2 类构造函数

constructor关键字用于在类定义块内部创建类的构造函数。方法名constructor会告诉解释器在使用new操作符创建类的新实例时，应该调用这个函数。构造函数的定义不是必需的，不定义构造函数相当于将构造函数定义为空函数。

1. **实例化**

   使用new操作符实例化Person的操作等于使用new调用其构造函数。唯一可感知的不同之处就是，JavaScript解释器知道使用new和类意味着应该使用constructor函数进行实例化。

   使用new调用类的构造函数会执行如下操作。

   (1) 在内存中创建一个新对象。

   (2) 这个新对象内部的[[Prototype]]指针被赋值为构造函数的prototype属性。

   (3) 构造函数内部的this被赋值为这个新对象（即this指向新对象）。

   (4) 执行构造函数内部的代码（给新对象添加属性）。

(5) 如果构造函数返回非空对象，则返回该对象；否则，返回刚创建的新对象。

来看下面的例子：

```
class Animal {}

class Person {
  constructor() {
    console.log('person ctor');
  }
}

class Vegetable {
  constructor() {
    this.color = 'orange';
  }
}

let a = new Animal();

let p = new Person();  // person ctor

let v = new Vegetable();
console.log(v.color);  // orange
```

类实例化时传入的参数会用作构造函数的参数。如果不需要参数，则类名后面的括号也是可选的：

```
class Person {
  constructor(name) {
    console.log(arguments.length);
    this.name = name || null;
  }
}

let p1 = new Person;          // 0
console.log(p1.name);         // null

let p2 = new Person();        // 0
console.log(p2.name);         // null

let p3 = new Person('Jake');  // 1
console.log(p3.name);         // Jake
```

默认情况下，类构造函数会在执行之后返回`this`对象。构造函数返回的对象会被用作实例化的对象，如果没有什么引用新创建的`this`对象，那么这个对象会被销毁。不过，如果返回的不是`this`对象，而是其他对象，那么这个对象不会通过`instanceof`操作符检测出跟类有关联，因为这个对象的原型指针并没有被修改。

```
class Person {
  constructor(override) {
    this.foo = 'foo';
    if (override) {
      return {
        bar: 'bar'
      };
    }
  }
}

let p1 = new Person(),
    p2 = new Person(true);

console.log(p1);                    // Person{ foo: 'foo' }
console.log(p1 instanceof Person);  // true

console.log(p2);                    // { bar: 'bar' }
console.log(p2 instanceof Person);  // false
```

类构造函数与构造函数的主要区别是，调用类构造函数必须使用new操作符。而普通构造函数如果不使用new调用，那么就会以全局的this（通常是window）作为内部对象。调用类构造函数时如果忘了使用new则会抛出错误：

```
function Person() {}

class Animal {}

// 把window作为this来构建实例
let p = Person();

let a = Animal();
// TypeError: class constructor Animal cannot be invoked without 'new'
```

类构造函数没有什么特殊之处，实例化之后，它会成为普通的实例方法（但作为类构造函数，仍然要使用new调用）。因此，实例化之后可以在实例上引用它：

```
class Person {}

// 使用类创建一个新实例
let p1 = new Person();

p1.constructor();
// TypeError: Class constructor Person cannot be invoked without 'new'

// 使用对类构造函数的引用创建一个新实例
let p2 = new p1.constructor();
```

2. **把类当成特殊函数**

ECMAScript中没有正式的类这个类型。从各方面来看，ECMAScript类就是一种特殊函数。声明一个类之后，通过typeof操作符检测类标识符，表明它是一个函数：

```
class Person {}

console.log(Person);         // class Person {}
console.log(typeof Person);  // function
```

类标签符有prototype属性，而这个原型也有一个constructor属性指向类自身：

```
class Person{}

console.log(Person.prototype);                          // { constructor: f()
}
console.log(Person === Person.prototype.constructor);  // true
```

与普通构造函数一样，可以使用instanceof操作符检查构造函数原型是否存在于实例的原型链中：

```
class Person {}

let p = new Person();

console.log(p instanceof Person); // true
```

由此可知，可以使用instanceof操作符检查一个对象与类构造函数，以确定这个对象是不是类的实例。只不过此时的类构造函数要使用类标签符，比如，在前面的例子中要检查p和Person。

如前所述，类本身具有与普通构造函数一样的行为。在类的上下文中，类本身在使用new调用时就会被当成构造函数。重点在于，类中定义的constructor方法**不会**被当成构造函数，在对它使用instanceof操作符时会返回false。但是，如果在创建实例时直接将类构造函数当成普通构造函数来使用，那么instanceof操作符的返回值会反转：

```
class Person {}

let p1 = new Person();

console.log(p1.constructor === Person);         // true
console.log(p1 instanceof Person);              // true
console.log(p1 instanceof Person.constructor);  // false

let p2 = new Person.constructor();

console.log(p2.constructor === Person);         // false
```

```
console.log(p2 instanceof Person);              // false
console.log(p2 instanceof Person.constructor);  // true
```

类是JavaScript的一等公民，因此可以像其他对象或函数引用一样把类作为参数传递：

```
// 类可以像函数一样在任何地方定义，比如在数组中
let classList = [
  class {
    constructor(id) {
      this.id_ = id;
      console.log('instance ${this.id_}');
    }
  }
];

function createInstance(classDefinition, id) {
  return new classDefinition(id);
}

let foo = createInstance(classList[0], 3141);  // instance 3141
```

与立即调用函数表达式相似，类也可以立即实例化：

```
// 因为是一个类表达式，所以类名是可选的
let p = new class Foo {
  constructor(x) {
    console.log(x);
  }
}('bar');        // bar

console.log(p);  // Foo {}
```

8.4.3 实例、原型和类成员

类的语法可以非常方便地定义应该存在于实例上的成员、应该存在于原型上的成员，以及应该存在于类本身的成员。

1. **实例成员**

   每次通过new调用类标识符时，都会执行类构造函数。在这个函数内部，可以为新创建的实例（this）添加“自有”属性。至于添加什么样的属性，则没有限制。另外，在构造函数执行完毕后，仍然可以给实例继续添加新成员。

   每个实例都对应一个唯一的成员对象，这意味着所有成员都不会在原型上共享：

```
class Person {
  constructor() {
```

```
    // 这个例子先使用对象包装类型定义一个字符串
    // 为的是在下面测试两个对象的相等性
    this.name = new String('Jack');

    this.sayName = () => console.log(this.name);

    this.nicknames = ['Jake', 'J-Dog']
  }
}

let p1 = new Person(),
    p2 = new Person();

p1.sayName(); // Jack
p2.sayName(); // Jack

console.log(p1.name === p2.name);            // false
console.log(p1.sayName === p2.sayName);      // false
console.log(p1.nicknames === p2.nicknames);  // false

p1.name = p1.nicknames[0];
p2.name = p2.nicknames[1];

p1.sayName();  // Jake
p2.sayName();  // J-Dog
```

2. **原型方法与访问器**

为了在实例间共享方法，类定义语法把在类块中定义的方法作为原型方法。

```
class Person {
  constructor() {
    // 添加到this的所有内容都会存在于不同的实例上
    this.locate = () => console.log('instance');
  }

  // 在类块中定义的所有内容都会定义在类的原型上
  locate() {
    console.log('prototype');
  }
}

let p = new Person();

p.locate();                 // instance
Person.prototype.locate();  // prototype
```

可以把方法定义在类构造函数中或者类块中，但不能在类块中给原型添加原始值或对象作为成员数据：

```
class Person {
  name: 'Jake'
}
// Uncaught SyntaxError: Unexpected token
```

类方法等同于对象属性，因此可以使用字符串、符号或计算的值作为键：

```
const symbolKey = Symbol('symbolKey');

class Person {

  stringKey() {
    console.log('invoked stringKey');
  }
   [symbolKey]() {
    console.log('invoked symbolKey');
  }
   ['computed' + 'Key']() {
    console.log('invoked computedKey');
  }
}

let p = new Person();

p.stringKey();    // invoked stringKey
p[symbolKey]();   // invoked symbolKey
p.computedKey();  // invoked computedKey
```

类定义也支持获取和设置访问器。语法与行为跟普通对象一样：

```
class Person {
  set name(newName) {
    this.name_ = newName;
  }

  get name() {
    return this.name_;
  }
}

let p = new Person();
p.name = 'Jake';
console.log(p.name); // Jake
```

3. **静态类方法**

可以在类上定义静态方法。这些方法通常用于执行不特定于实例的操作，也不要求存在类的实例。与原型成员类似，每个类上只能有一个静态成员。

静态类成员在类定义中使用`static`关键字作为前缀。在静态成员中，`this`引用类自身。其他所有约定跟原型成员一样：

```
class Person {
  constructor() {
    // 添加到this的所有内容都会存在于不同的实例上
    this.locate = () => console.log('instance', this);
  }

  // 定义在类的原型对象上
  locate() {
    console.log('prototype', this);
  }

  // 定义在类本身上
  static locate() {
    console.log('class', this);
  }
}

let p = new Person();

p.locate();                 // instance, Person {}
Person.prototype.locate();  // prototype, {constructor: ... }
Person.locate();            // class, class Person {}
```

静态类方法非常适合作为实例工厂：

```
class Person {
  constructor(age) {
    this.age_ = age;
  }

  sayAge() {
    console.log(this.age_);
  }

  static create() {
    // 使用随机年龄创建并返回一个Person实例
    return new Person(Math.floor(Math.random()*100));
  }
}

console.log(Person.create()); // Person { age_: ... }
```

4. **非函数原型和类成员**

虽然类定义并不显式支持在原型或类上添加成员数据，但在类定义外部，可以手动添加：

```
class Person {
  sayName() {
    console.log('${Person.greeting} ${this.name}');
  }
}

// 在类上定义数据成员
Person.greeting = 'My name is';

// 在原型上定义数据成员
Person.prototype.name = 'Jake';

let p = new Person();
p.sayName();  // My name is Jake
```

> **注意** 类定义中之所以没有显式支持添加数据成员，是因为在共享目标（原型和类）上添加可变（可修改）数据成员是一种反模式。一般来说，对象实例应该独自拥有通过this引用的数据。

5. **迭代器与生成器方法**

类定义语法支持在原型和类本身上定义生成器方法：

```
class Person {
  // 在原型上定义生成器方法
  *createNicknameIterator() {
    yield 'Jack';
    yield 'Jake';
    yield 'J-Dog';
  }

  // 在类上定义生成器方法
  static *createJobIterator() {
    yield 'Butcher';
    yield 'Baker';
    yield 'Candlestick maker';
  }
}

let jobIter = Person.createJobIterator();
console.log(jobIter.next().value);  // Butcher
console.log(jobIter.next().value);  // Baker
console.log(jobIter.next().value);  // Candlestick maker

let p = new Person();
let nicknameIter = p.createNicknameIterator();
console.log(nicknameIter.next().value);  // Jack
console.log(nicknameIter.next().value);  // Jake
console.log(nicknameIter.next().value);  // J-Dog
```

因为支持生成器方法，所以可以通过添加一个默认的迭代器，把类实例变成可迭代对象：

```
class Person {
  constructor() {
    this.nicknames = ['Jack', 'Jake', 'J-Dog'];
  }

  *[Symbol.iterator]() {
    yield *this.nicknames.entries();
  }
}

let p = new Person();
for (let [idx, nickname] of p) {
  console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

也可以只返回迭代器实例：

```
class Person {
  constructor() {
    this.nicknames = ['Jack', 'Jake', 'J-Dog'];
  }

  [Symbol.iterator]() {
    return this.nicknames.entries();
  }
}

let p = new Person();
for (let [idx, nickname] of p) {
  console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

8.4.4 继承

本章前面花了大量篇幅讨论如何使用ES5的机制实现继承。ECMAScript 6新增特性中最出色的一个就是原生支持了类继承机制。虽然类继承使用的是新语法，但背后依旧使用的是原型链。

1. **继承基础**

ES6类支持单继承。使用extends关键字，就可以继承任何拥有[[Construct]]和原型的对象。很大程度上，这意味着不仅可以继承一个类，也可以继承普通的构造函数（保持向后兼容）：

```
class Vehicle {}

// 继承类
class Bus extends Vehicle {}

let b = new Bus();
console.log(b instanceof Bus);      // true
console.log(b instanceof Vehicle);  // true


function Person() {}

// 继承普通构造函数
class Engineer extends Person {}

let e = new Engineer();
console.log(e instanceof Engineer);  // true
console.log(e instanceof Person);    // true
```

类和原型上定义的方法都会带到派生类。this的值会反映调用相应方法的实例或者类：

```
class Vehicle {
  identifyPrototype(id) {
    console.log(id, this);
  }

  static identifyClass(id) {
    console.log(id, this);
  }
}

class Bus extends Vehicle {}

let v = new Vehicle();
let b = new Bus();

b.identifyPrototype('bus');       // bus, Bus {}
v.identifyPrototype('vehicle');   // vehicle, Vehicle {}

Bus.identifyClass('bus');         // bus, class Bus {}
Vehicle.identifyClass('vehicle'); // vehicle, class Vehicle {}
```

> **注意** extends关键字也可以在类表达式中使用，因此let Bar = class extends Foo {}是有效的语法。

2. **构造函数、HomeObject和super()**

派生类的方法可以通过super关键字引用它们的原型。这个关键字只能在派生类中使用，而且仅限于类构造函数、实例方法和静态方法内部。在类构造函数中使用super可以调用父类构造函数。

```
class Vehicle {
  constructor() {
    this.hasEngine = true;
  }
}

class Bus extends Vehicle {
  constructor() {
    // 不要在调用super()之前引用this，否则会抛出ReferenceError

    super(); // 相当于super.constructor()

    console.log(this instanceof Vehicle);  // true
    console.log(this);                     // Bus { hasEngine: true }
  }
}

new Bus();
```

在静态方法中可以通过super调用继承的类上定义的静态方法：

```
class Vehicle {
  static identify() {
    console.log('vehicle');
  }
}

class Bus extends Vehicle {
  static identify() {
    super.identify();
  }
}

Bus.identify();  // vehicle
```

> **注意** ES6给类构造函数和静态方法添加了内部特性[[HomeObject]]，这个特性是一个指针，指向定义该方法的对象。这个指针是自动赋值的，而且只能在JavaScript引擎内部访问。super始终会定义为[[HomeObject]]的原型。

在使用super时要注意几个问题。

- super只能在派生类构造函数和静态方法中使用。

```
class Vehicle {
  constructor() {
```

```
    super();
    // SyntaxError: 'super' keyword unexpected
  }
}
```

- 不能单独引用super关键字，要么用它调用构造函数，要么用它引用静态方法。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    console.log(super);
    // SyntaxError: 'super' keyword unexpected here
  }
}
```

- 调用super()会调用父类构造函数，并将返回的实例赋值给this。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    super();

    console.log(this instanceof Vehicle);
  }
}

new Bus(); // true
```

- super()的行为如同调用构造函数，如果需要给父类构造函数传参，则需要手动传入。

```
class Vehicle {
  constructor(licensePlate) {
    this.licensePlate = licensePlate;
  }
}

class Bus extends Vehicle {
  constructor(licensePlate) {
    super(licensePlate);
  }
}

console.log(new Bus('1337H4X')); // Bus { licensePlate: '1337H4X' }
```

- 如果没有定义类构造函数，在实例化派生类时会调用`super()`，而且会传入所有传给派生类的参数。

```
class Vehicle {
  constructor(licensePlate) {
    this.licensePlate = licensePlate;
  }
}

class Bus extends Vehicle {}

console.log(new Bus('1337H4X')); // Bus { licensePlate: '1337H4X' }
```

- 在类构造函数中，不能在调用`super()`之前引用`this`。

```
class Vehicle {}

class Bus extends Vehicle {
  constructor() {
    console.log(this);
  }
}

new Bus();
// ReferenceError: Must call super constructor in derived class
// before accessing 'this' or returning from derived constructor
```

- 如果在派生类中显式定义了构造函数，则要么必须在其中调用`super()`，要么必须在其中返回一个对象。

```
class Vehicle {}

class Car extends Vehicle {}

class Bus extends Vehicle {
  constructor() {
    super();
  }
}

class Van extends Vehicle {
  constructor() {
    return {};
  }
}

console.log(new Car());  // Car {}
```

```
  console.log(new Bus());  // Bus {}
  console.log(new Van());  // {}
```

3. **抽象基类**

有时候可能需要定义这样一个类，它可供其他类继承，但本身不会被实例化。虽然ECMAScript没有专门支持这种类的语法，但通过new.target也很容易实现。new.target保存通过new关键字调用的类或函数。通过在实例化时检测new.target是不是抽象基类，可以阻止对抽象基类的实例化：

```
// 抽象基类
class Vehicle {
  constructor() {
    console.log(new.target);
    if (new.target === Vehicle) {
      throw new Error('Vehicle cannot be directly instantiated');
    }
  }
}

// 派生类
class Bus extends Vehicle {}

new Bus();       // class Bus {}
new Vehicle();   // class Vehicle {}
// Error: Vehicle cannot be directly instantiated
```

另外，通过在抽象基类构造函数中进行检查，可以要求派生类必须定义某个方法。因为原型方法在调用类构造函数之前就已经存在了，所以可以通过this关键字来检查相应的方法：

```
// 抽象基类
class Vehicle {
  constructor() {
    if (new.target === Vehicle) {
      throw new Error('Vehicle cannot be directly instantiated');
    }

    if (!this.foo) {
      throw new Error('Inheriting class must define foo()');
    }

    console.log('success!');
  }
}

// 派生类
class Bus extends Vehicle {
  foo() {}
}
```

```
// 派生类
class Van extends Vehicle {}

new Bus(); // success!
new Van(); // Error: Inheriting class must define foo()
```

4. **继承内置类型**

ES6类为继承内置引用类型提供了顺畅的机制，开发者可以方便地扩展内置类型：

```
class SuperArray extends Array {
  shuffle() {
    // 洗牌算法
    for (let i = this.length - 1; i > 0; i--) {
      const j = Math.floor(Math.random() * (i + 1));
      [this[i], this[j]] = [this[j], this[i]];
    }
  }
}

let a = new SuperArray(1, 2, 3, 4, 5);

console.log(a instanceof Array);       // true
console.log(a instanceof SuperArray);  // true

console.log(a);  // [1, 2, 3, 4, 5]
a.shuffle();
console.log(a);  // [3, 1, 4, 5, 2]
```

有些内置类型的方法会返回新实例。默认情况下，返回实例的类型与原始实例的类型是一致的：

```
class SuperArray extends Array {}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);  // [1, 2, 3, 4, 5]
console.log(a2);  // [1, 3, 5]
console.log(a1 instanceof SuperArray);  // true
console.log(a2 instanceof SuperArray);  // true
```

如果想覆盖这个默认行为，则可以覆盖`Symbol.species`访问器，这个访问器决定在创建返回的实例时使用的类：

```
class SuperArray extends Array {
  static get [Symbol.species]() {
    return Array;
```

```
  }
}

let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x%2))

console.log(a1);  // [1, 2, 3, 4, 5]
console.log(a2);  // [1, 3, 5]
console.log(a1 instanceof SuperArray);  // true
console.log(a2 instanceof SuperArray);  // false
```

5. **类混入**

把不同类的行为集中到一个类是一种常见的JavaScript模式。虽然ES6没有显式支持多类继承，但通过现有特性可以轻松地模拟这种行为。

> **注意** `Object.assign()`方法是为了混入对象行为而设计的。只有在需要混入类的行为时才有必要自己实现混入表达式。如果只是需要混入多个对象的属性，那么使用`Object.assign()`就可以了。

在下面的代码片段中，`extends`关键字后面是一个JavaScript表达式。任何可以解析为一个类或一个构造函数的表达式都是有效的。这个表达式会在求值类定义时被求值：

```
class Vehicle {}

function getParentClass() {
  console.log('evaluated expression');
  return Vehicle;
}

class Bus extends getParentClass() {}
// 可求值的表达式
```

混入模式可以通过在一个表达式中连缀多个混入元素来实现，这个表达式最终会解析为一个可以被继承的类。如果`Person`类需要组合A、B、C，则需要某种机制实现B继承A，C继承B，而`Person`再继承C，从而把A、B、C组合到这个超类中。实现这种模式有不同的策略。

一个策略是定义一组“可嵌套”的函数，每个函数分别接收一个超类作为参数，而将混入类定义为这个参数的子类，并返回这个类。这些组合函数可以连缀调用，最终组合成超类表达式：

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
  foo() {
    console.log('foo');
  }
};
let BarMixin = (Superclass) => class extends Superclass {
  bar() {
```

```
    console.log('bar');
  }
};
let BazMixin = (Superclass) => class extends Superclass {
  baz() {
    console.log('baz');
  }
};

class Bus extends FooMixin(BarMixin(BazMixin(Vehicle))) {}

let b = new Bus();
b.foo();  // foo
b.bar();  // bar
b.baz();  // baz
```

通过写一个辅助函数，可以把嵌套调用展开：

```
class Vehicle {}

let FooMixin = (Superclass) => class extends Superclass {
  foo() {
    console.log('foo');
  }
};
let BarMixin = (Superclass) => class extends Superclass {
  bar() {
    console.log('bar');
  }
};
let BazMixin = (Superclass) => class extends Superclass {
  baz() {
    console.log('baz');
  }
};

function mix(BaseClass, ...Mixins) {
  return Mixins.reduce((accumulator, current) => current(accumulator),
BaseClass);
}

class Bus extends mix(Vehicle, FooMixin, BarMixin, BazMixin) {}

let b = new Bus();
b.foo();  // foo
b.bar();  // bar
b.baz();  // baz
```

> **注意** 很多JavaScript框架（特别是React）已经抛弃混入模式，转向了复合模式（把方法提取到独立的类和辅助对象中，然后把它们组合起来，但不使用继承）。这反映了那个众所周知的软件

> 设计原则：“复合胜过继承（composition over inheritance）。”这个设计原则被很多人遵循，在代码设计中能提供极大的灵活性。

8.5 小结

对象在代码执行过程中的任何时候都可以被创建和增强，具有极大的动态性，并不是严格定义的实体。下面的模式适用于创建对象。

- 工厂模式就是一个简单的函数，这个函数可以创建对象，为它添加属性和方法，然后返回这个对象。这个模式在构造函数模式出现后就很少用了。
- 使用构造函数模式可以自定义引用类型，可以使用`new`关键字像创建内置类型实例一样创建自定义类型的实例。不过，构造函数模式也有不足，主要是其成员无法重用，包括函数。考虑到函数本身是松散的、弱类型的，没有理由让函数不能在多个对象实例间共享。
- 原型模式解决了成员共享的问题，只要是添加到构造函数`prototype`上的属性和方法就可以共享。而组合构造函数和原型模式通过构造函数定义实例属性，通过原型定义共享的属性和方法。

JavaScript的继承主要通过原型链来实现。原型链涉及把构造函数的原型赋值为另一个类型的实例。这样一来，子类就可以访问父类的所有属性和方法，就像基于类的继承那样。原型链的问题是所有继承的属性和方法都会在对象实例间共享，无法做到实例私有。盗用构造函数模式通过在子类构造函数中调用父类构造函数，可以避免这个问题。这样可以让每个实例继承的属性都是私有的，但要求类型只能通过构造函数模式来定义（因为子类不能访问父类原型上的方法）。目前最流行的继承模式是组合继承，即通过原型链继承共享的属性和方法，通过盗用构造函数继承实例属性。

除上述模式之外，还有以下几种继承模式。

- 原型式继承可以无须明确定义构造函数而实现继承，本质上是对给定对象执行浅复制。这种操作的结果之后还可以再进一步增强。
- 与原型式继承紧密相关的是寄生式继承，即先基于一个对象创建一个新对象，然后再增强这个新对象，最后返回新对象。这个模式也被用在组合继承中，用于避免重复调用父类构造函数导致的浪费。
- 寄生组合继承被认为是实现基于类型继承的最有效方式。

ECMAScript 6新增的类很大程度上是基于既有原型机制的语法糖。类的语法让开发者可以优雅地定义向后兼容的类，既可以继承内置类型，也可以继承自定义类型。类有效地跨越了对象实例、对象原型和对象类之间的鸿沟。

第 9 章 代理与反射

> **本章内容**
>
> - 代理基础
> - 代码捕获器与反射方法
> - 代理模式

ECMAScript 6新增的代理和反射为开发者提供了拦截并向基本操作嵌入额外行为的能力。具体地说，可以给目标对象定义一个关联的代理对象，而这个代理对象可以作为抽象的目标对象来使用。在对目标对象的各种操作影响目标对象之前，可以在代理对象中对这些操作加以控制。

对刚刚接触这个主题的开发者而言，代理是一个比较模糊的概念，而且还夹杂着很多新术语。其实只要看几个例子，就很容易理解了。

> **注意** 在ES6之前，ECMAScript中并没有类似代理的特性。由于代理是一种新的基础性语言能力，很多转译程序都不能把代理行为转换为之前的ECMAScript代码，因为代理的行为实际上是无可替代的。为此，代理和反射只在百分之百支持它们的平台上有用。可以检测代理是否存在，不存在则提供后备代码。不过这会导致代码冗余，因此并不推荐。

9.1 代理基础

正如本章开头所介绍的，代理是目标对象的抽象。从很多方面看，代理类似C++指针，因为它可以用作目标对象的替身，但又完全独立于目标对象。目标对象既可以直接被操作，也可以通过代理来操作。但直接操作会绕过代理施予的行为。

> **注意** ECMAScript代理与C++指针有重大区别，后面会再讨论。不过作为一种有助于理解的类比，指针在概念上还是比较合适的结构。

9.1.1 创建空代理

最简单的代理是空代理，即除了作为一个抽象的目标对象，什么也不做。默认情况下，在代理对象上执行的所有操作都会无障碍地传播到目标对象。因此，在任何可以使用目标对象的地方，都可以通过同样的方式来使用与之关联的代理对象。

代理是使用`Proxy`构造函数创建的。这个构造函数接收两个参数：目标对象和处理程序对象。缺少其中任何一个参数都会抛出`TypeError`。要创建空代理，可以传一个简单的对象字面量作为处理程序对象，从而让所有操作畅通无阻地抵达目标对象。

如下面的代码所示，在代理对象上执行的任何操作实际上都会应用到目标对象。唯一可感知的不同就是代码中操作的是代理对象。

```
const target = {
  id: 'target'
};

const handler = {};

const proxy = new Proxy(target, handler);

// id属性会访问同一个值
console.log(target.id);  // target
console.log(proxy.id);   // target

// 给目标属性赋值会反映在两个对象上
// 因为两个对象访问的是同一个值
target.id = 'foo';
console.log(target.id); // foo
console.log(proxy.id);  // foo

// 给代理属性赋值会反映在两个对象上
// 因为这个赋值会转移到目标对象
proxy.id = 'bar';
console.log(target.id); // bar
console.log(proxy.id);  // bar
```

```
// hasOwnProperty()方法在两个地方
// 都会应用到目标对象
console.log(target.hasOwnProperty('id')); // true
console.log(proxy.hasOwnProperty('id'));  // true

// Proxy.prototype是undefined
// 因此不能使用instanceof操作符
console.log(target instanceof Proxy); // TypeError: Function has non-object
prototype 'undefined' in instanceof check
console.log(proxy instanceof Proxy);  // TypeError: Function has non-object
prototype 'undefined' in instanceof check

// 严格相等可以用来区分代理和目标
console.log(target === proxy); // false
```

9.1.2 定义捕获器

使用代理的主要目的是可以定义**捕获器**（trap）。捕获器就是在处理程序对象中定义的"基本操作的拦截器"。每个处理程序对象可以包含零个或多个捕获器，每个捕获器都对应一种基本操作，可以直接或间接在代理对象上调用。每次在代理对象上调用这些基本操作时，代理可以在这些操作传播到目标对象之前先调用捕获器函数，从而拦截并修改相应的行为。

> **注意** 捕获器（trap）是从操作系统中借用的概念。在操作系统中，捕获器是程序流中的一个同步中断，可以暂停程序流，转而执行一段子例程，之后再返回原始程序流。

例如，可以定义一个`get()`捕获器，在ECMAScript操作以某种形式调用`get()`时触发。下面的例子定义了一个`get()`捕获器：

```
const target = {
  foo: 'bar'
};

const handler = {
  // 捕获器在处理程序对象中以方法名为键
  get() {
    return 'handler override';
  }
};

const proxy = new Proxy(target, handler);
```

这样，当通过代理对象执行`get()`操作时，就会触发定义的`get()`捕获器。当然，`get()`不是ECMAScript对象可以调用的方法。这个操作在JavaScript代码中可以通过多种形式触发并被`get()`捕获器拦截到。`proxy[property]`、`proxy.property`或`Object.create(proxy)[property]`等操作都会触发基本的`get()`操作以获取属性。因此所有这些操作只要发生在代理对象上，就会触发`get()`捕获器。注意，只有在代理对象上执行这些操作才会触发捕获器。在目标对象上执行这些操作仍然会产生正常的行为。

```
const target = {
  foo: 'bar'
};

const handler = {
  // 捕获器在处理程序对象中以方法名为键
  get() {
    return 'handler override';
  }
};

const proxy = new Proxy(target, handler);

console.log(target.foo);                    // bar
console.log(proxy.foo);                     // handler override

console.log(target['foo']);                 // bar
console.log(proxy['foo']);                  // handler override

console.log(Object.create(target)['foo']);  // bar
console.log(Object.create(proxy)['foo']);   // handler override
```

9.1.3 捕获器参数和反射API

所有捕获器都可以访问相应的参数，基于这些参数可以重建被捕获方法的原始行为。比如，`get()`捕获器会接收到目标对象、要查询的属性和代理对象三个参数。

```
const target = {
  foo: 'bar'
};

const handler = {
  get(trapTarget, property, receiver) {
    console.log(trapTarget === target);
    console.log(property);
    console.log(receiver === proxy);
  }
};

const proxy = new Proxy(target, handler);

proxy.foo;
// true
// foo
// true
```

有了这些参数，就可以重建被捕获方法的原始行为：

```
const target = {
  foo: 'bar'
};

const handler = {
  get(trapTarget, property, receiver) {
    return trapTarget[property];
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);  // bar
console.log(target.foo); // bar
```

所有捕获器都可以基于自己的参数重建原始操作，但并非所有捕获器行为都像`get()`那么简单。因此，通过手动写码如法炮制的想法是不现实的。实际上，开发者并不需要手动重建原始行为，而是可以通过调用全局`Reflect`对象上（封装了原始行为）的同名方法来轻松重建。

处理程序对象中所有可以捕获的方法都有对应的反射（Reflect）API方法。这些方法与捕获器拦截的方法具有相同的名称和函数签名，而且也具有与被拦截方法相同的行为。因此，使用反射API也可以像下面这样定义出空代理对象：

```
const target = {
  foo: 'bar'
};

const handler = {
  get() {
    return Reflect.get(...arguments);
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);   // bar
console.log(target.foo);  // bar
```

甚至还可以写得更简洁一些：

```
const target = {
  foo: 'bar'
};

const handler = {
  get: Reflect.get
};
```

```
const proxy = new Proxy(target, handler);

console.log(proxy.foo);  // bar
console.log(target.foo); // bar
```

事实上，如果真想创建一个可以捕获所有方法，然后将每个方法转发给对应反射API的空代理，那么甚至不需要定义处理程序对象：

```
const target = {
  foo: 'bar'
};

const proxy = new Proxy(target, Reflect);

console.log(proxy.foo);   // bar
console.log(target.foo);  // bar
```

反射API为开发者准备好了样板代码，在此基础上开发者可以用最少的代码修改捕获的方法。比如，下面的代码在某个属性被访问时，会对返回的值进行一番修饰：

```
const target = {
  foo: 'bar',
  baz: 'qux'
};

const handler = {
  get(trapTarget, property, receiver) {
    let decoration = '';
    if (property === 'foo') {
      decoration = '!!!';
    }

    return Reflect.get(...arguments) + decoration;
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);   // bar!!!
console.log(target.foo);  // bar

console.log(proxy.baz);   // qux
console.log(target.baz);  // qux
```

9.1.4 捕获器不变式

使用捕获器几乎可以改变所有基本方法的行为，但也不是没有限制。根据ECMAScript规范，每个捕获的方法都知道目标对象上下文、捕获函数签名，而捕获处理程序的行为必须遵循“捕获器不变式”（trap invariant）。捕

获器不变式因方法不同而异，但通常都会防止捕获器定义出现过于反常的行为。

比如，如果目标对象有一个不可配置且不可写的数据属性，那么在捕获器返回一个与该属性不同的值时，会抛出TypeError：

```
const target = {};
Object.defineProperty(target, 'foo', {
  configurable: false,
  writable: false,
  value: 'bar'
});

const handler = {
  get() {
    return 'qux';
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);
// TypeError
```

9.1.5 可撤销代理

有时候可能需要中断代理对象与目标对象之间的联系。对于使用new Proxy()创建的普通代理来说，这种联系会在代理对象的生命周期内一直持续存在。

Proxy也暴露了revocable()方法，这个方法支持撤销代理对象与目标对象的关联。撤销代理的操作是不可逆的。而且，撤销函数（revoke()）是幂等的，调用多少次的结果都一样。撤销代理之后再调用代理会抛出TypeError。

撤销函数和代理对象是在实例化时同时生成的：

```
const target = {
  foo: 'bar'
};

const handler = {
  get() {
    return 'intercepted';
  }
};

const { proxy, revoke } = Proxy.revocable(target, handler);

console.log(proxy.foo);   // intercepted
console.log(target.foo);  // bar

revoke();
```

```
console.log(proxy.foo);   // TypeError
```

9.1.6 实用反射API

某些情况下应该优先使用反射API，这是有一些理由的。

1. **反射API与对象API**

   在使用反射API时，要记住：

   (1) 反射API并不限于捕获处理程序；

   (2) 大多数反射API方法在`Object`类型上有对应的方法。

   通常，`Object`上的方法适用于通用程序，而反射方法适用于细粒度的对象控制与操作。

2. **状态标记**

   很多反射方法返回称作"状态标记"的布尔值，表示意图执行的操作是否成功。有时候，状态标记比那些返回修改后的对象或者抛出错误（取决于方法）的反射API方法更有用。例如，可以使用反射API对下面的代码进行重构：

   ```
   // 初始代码

   const o = {};

   try {
     Object.defineProperty(o, 'foo', 'bar');
     console.log('success');
   } catch(e) {
     console.log('failure');
   }
   ```

   在定义新属性时如果发生问题，`Reflect.defineProperty()`会返回`false`，而不是抛出错误。因此使用这个反射方法可以这样重构上面的代码：

   ```
   // 重构后的代码

   const o = {};

   if(Reflect.defineProperty(o, 'foo', {value: 'bar'})) {
     console.log('success');
   } else {
     console.log('failure');
   }
   ```

   以下反射方法都会提供状态标记：

- `Reflect.defineProperty()`
- `Reflect.preventExtensions()`
- `Reflect.setPrototypeOf()`
- `Reflect.set()`
- `Reflect.deleteProperty()`

3. **用一等函数替代操作符**

   以下反射方法提供只有通过操作符才能完成的操作。

   - `Reflect.get()`：可以替代对象属性访问操作符。
   - `Reflect.set()`：可以替代`=`赋值操作符。
   - `Reflect.has()`：可以替代`in`操作符或`with()`。
   - `Reflect.deleteProperty()`：可以替代`delete`操作符。
   - `Reflect.construct()`：可以替代`new`操作符。

4. **安全地应用函数**

   在通过`apply`方法调用函数时，被调用的函数可能也定义了自己的`apply`属性（虽然可能性极小）。为绕过这个问题，可以使用定义在`Function`原型上的`apply`方法，比如：

   ```
   Function.prototype.apply.call(myFunc, thisVal, argumentList);
   ```

   这种可怕的代码完全可以使用`Reflect.apply`来避免：

   ```
   Reflect.apply(myFunc, thisVal, argumentsList);
   ```

9.1.7 代理另一个代理

代理可以拦截反射API的操作，而这意味着完全可以创建一个代理，通过它去代理另一个代理。这样就可以在一个目标对象之上构建多层拦截网：

```
const target = {
  foo: 'bar'
};

const firstProxy = new Proxy(target, {
  get() {
    console.log('first proxy');
    return Reflect.get(...arguments);
  }
});

const secondProxy = new Proxy(firstProxy, {
  get() {
    console.log('second proxy');
    return Reflect.get(...arguments);
```

```
  }
});

console.log(secondProxy.foo);
// second proxy
// first proxy
// bar
```

9.1.8 代理的问题与不足

代理是在ECMAScript现有基础之上构建起来的一套新API，因此其实现已经尽力做到最好了。很大程度上，代理作为对象的虚拟层可以正常使用。但在某些情况下，代理也不能与现在的ECMAScript机制很好地协同。

1. **代理中的`this`**

   代理潜在的一个问题来源是`this`值。我们知道，方法中的`this`通常指向调用这个方法的对象：

   ```
   const target = {
     thisValEqualsProxy() {
       return this === proxy;
     }
   }

   const proxy = new Proxy(target, {});

   console.log(target.thisValEqualsProxy());  // false
   console.log(proxy.thisValEqualsProxy());   // true
   ```

   从直觉上讲，这样完全没有问题：调用代理上的任何方法，比如`proxy.outerMethod()`，而这个方法进而又会调用另一个方法，如`this.innerMethod()`，实际上都会调用`proxy.innerMethod()`。多数情况下，这是符合预期的行为。可是，如果目标对象依赖于对象标识，那就可能碰到意料之外的问题。

   还记得第6章中通过`WeakMap`保存私有变量的例子吧，以下是它的简化版：

   ```
   const wm = new WeakMap();

   class User {
     constructor(userId) {
       wm.set(this, userId);
     }

     set id(userId) {
       wm.set(this, userId);
     }

     get id() {
       return wm.get(this);
     }
   }
   ```

由于这个实现依赖User实例的对象标识，在这个实例被代理的情况下就会出问题：

```
const user = new User(123);
console.log(user.id); // 123

const userInstanceProxy = new Proxy(user, {});
console.log(userInstanceProxy.id); // undefined
```

这是因为User实例一开始使用目标对象作为WeakMap的键，代理对象却尝试从**自身**取得这个实例。要解决这个问题，就需要重新配置代理，把代理User实例改为代理User类本身。之后再创建代理的实例就会以代理实例作为WeakMap的键了：

```
const UserClassProxy = new Proxy(User, {});
const proxyUser = new UserClassProxy(456);
console.log(proxyUser.id);
```

2. **代理与内部槽位**

代理与内置引用类型（比如Array）的实例通常可以很好地协同，但有些ECMAScript内置类型可能会依赖代理无法控制的机制，结果导致在代理上调用某些方法会出错。

一个典型的例子就是Date类型。根据ECMAScript规范，Date类型方法的执行依赖this值上的内部槽位[[NumberDate]]。代理对象上不存在这个内部槽位，而且这个内部槽位的值也不能通过普通的get()和set()操作访问到，于是代理拦截后本应转发给目标对象的方法会抛出TypeError：

```
const target = new Date();
const proxy = new Proxy(target, {});

console.log(proxy instanceof Date);  // true

proxy.getDate();  // TypeError: 'this' is not a Date object
```

9.2 代理捕获器与反射方法

代理可以捕获13种不同的基本操作。这些操作有各自不同的反射API方法、参数、关联ECMAScript操作和不变式。

正如前面示例所展示的，有几种不同的JavaScript操作会调用同一个捕获器处理程序。不过，对于在代理对象上执行的任何一种操作，只会有一个捕获处理程序被调用。不会存在重复捕获的情况。

只要在代理上调用，所有捕获器都会拦截它们对应的反射API操作。

9.2.1 get()

get()捕获器会在获取属性值的操作中被调用。对应的反射API方法为Reflect.get()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  get(target, property, receiver) {
    console.log('get()');
    return Reflect.get(...arguments)
  }
});

proxy.foo;
// get()
```

1. **返回值**

   返回值无限制。

2. **拦截的操作**

   - `proxy.property`
   - `proxy[property]`
   - `Object.create(proxy)[property]`
   - `Reflect.get(proxy, property, receiver)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `property`：引用的目标对象上的字符串键属性。
   - `receiver`：代理对象或继承代理对象的对象。

4. **捕获器不变式**

   如果`target.property`不可写且不可配置，则处理程序返回的值必须与`target.property`匹配。

   如果`target.property`不可配置且`[[Get]]`特性为`undefined`，处理程序的返回值也必须是`undefined`。

9.2.2 `set()`

`set()`捕获器会在设置属性值的操作中被调用。对应的反射API方法为`Reflect.set()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  set(target, property, value, receiver) {
    console.log('set()');
    return Reflect.set(...arguments)
  }
});
```

```
proxy.foo = 'bar';
// set()
```

1. **返回值**

   返回`true`表示成功；返回`false`表示失败，严格模式下会抛出`TypeError`。

2. **拦截的操作**

   - `proxy.property = value`
   - `proxy[property] = value`
   - `Object.create(proxy)[property] = value`
   - `Reflect.set(proxy, property, value, receiver)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `property`：引用的目标对象上的字符串键属性。
   - `value`：要赋给属性的值。
   - `receiver`：接收最初赋值的对象。

4. **捕获器不变式**

   如果`target.property`不可写且不可配置，则不能修改目标属性的值。

   如果`target.property`不可配置且`[[Set]]`特性为`undefined`，则不能修改目标属性的值。

   在严格模式下，处理程序中返回`false`会抛出`TypeError`。

9.2.3 has()

`has()`捕获器会在`in`操作符中被调用。对应的反射API方法为`Reflect.has()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  has(target, property) {
    console.log('has()');
    return Reflect.has(...arguments)
  }
});

'foo' in proxy;
// has()
```

1. **返回值**

   `has()`必须返回布尔值，表示属性是否存在。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

- `property in proxy`
- `property in Object.create(proxy)`
- `with(proxy) {(property);}`
- `Reflect.has(proxy, property)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `property`：引用的目标对象上的字符串键属性。

4. **捕获器不变式**

   如果`target.property`存在且不可配置，则处理程序必须返回`true`。

   如果`target.property`存在且目标对象不可扩展，则处理程序必须返回`true`。

9.2.4 `defineProperty()`

`defineProperty()`捕获器会在`Object.defineProperty()`中被调用。对应的反射API方法为`Reflect.defineProperty()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  defineProperty(target, property, descriptor) {
    console.log('defineProperty()');
    return Reflect.defineProperty(...arguments)
  }
});

Object.defineProperty(proxy, 'foo', { value: 'bar' });
// defineProperty()
```

1. **返回值**

   `defineProperty()`必须返回布尔值，表示属性是否成功定义。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

   - `Object.defineProperty(proxy, property, descriptor)`
   - `Reflect.defineProperty(proxy, property, descriptor)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `property`：引用的目标对象上的字符串键属性。
   - `descriptor`：包含可选的`enumerable`、`configurable`、`writable`、`value`、`get`和`set`定义的对象。

4. **捕获器不变式**

如果目标对象不可扩展，则无法定义属性。

如果目标对象有一个可配置的属性，则不能添加同名的不可配置属性。

如果目标对象有一个不可配置的属性，则不能添加同名的可配置属性。

9.2.5 getOwnPropertyDescriptor()

getOwnPropertyDescriptor()捕获器会在Object.getOwnPropertyDescriptor()中被调用。对应的反射API方法为Reflect.getOwnPropertyDescriptor()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  getOwnPropertyDescriptor(target, property) {
    console.log('getOwnPropertyDescriptor()');
    return Reflect.getOwnPropertyDescriptor(...arguments)
  }
});

Object.getOwnPropertyDescriptor(proxy, 'foo');
// getOwnPropertyDescriptor()
```

1. **返回值**

   getOwnPropertyDescriptor()必须返回对象，或者在属性不存在时返回undefined。

2. **拦截的操作**

   - Object.getOwnPropertyDescriptor(proxy, property)
   - Reflect.getOwnPropertyDescriptor(proxy, property)

3. **捕获器处理程序参数**

   - target：目标对象。
   - property：引用的目标对象上的字符串键属性。

4. **捕获器不变式**

   如果自有的target.property存在且不可配置，则处理程序必须返回一个表示该属性存在的对象。

   如果自有的target.property存在且可配置，则处理程序必须返回表示该属性可配置的对象。

   如果自有的target.property存在且target不可扩展，则处理程序必须返回一个表示该属性存在的对象。

   如果target.property不存在且target不可扩展，则处理程序必须返回undefined表示该属性不存在。

   如果target.property不存在，则处理程序不能返回表示该属性可配置的对象。

9.2.6 deleteProperty()

`deleteProperty()`捕获器会在`delete`操作符中被调用。对应的反射API方法为`Reflect.deleteProperty()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  deleteProperty(target, property) {
    console.log('deleteProperty()');
    return Reflect.deleteProperty(...arguments)
  }
});

delete proxy.foo
// deleteProperty()
```

1. **返回值**

   `deleteProperty()`必须返回布尔值，表示删除属性是否成功。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

   - `delete proxy.property`
   - `delete proxy[property]`
   - `Reflect.deleteProperty(proxy, property)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `property`：引用的目标对象上的字符串键属性。

4. **捕获器不变式**

   如果自有的`target.property`存在且不可配置，则处理程序不能删除这个属性。

9.2.7 `ownKeys()`

`ownKeys()`捕获器会在`Object.keys()`及类似方法中被调用。对应的反射API方法为`Reflect.ownKeys()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  ownKeys(target) {
    console.log('ownKeys()');
    return Reflect.ownKeys(...arguments)
  }
});

Object.keys(proxy);
// ownKeys()
```

1. **返回值**

   `ownKeys()`必须返回包含字符串或符号的可枚举对象。

2. **拦截的操作**

   - `Object.getOwnPropertyNames(proxy)`
   - `Object.getOwnPropertySymbols(proxy)`
   - `Object.keys(proxy)`
   - `Reflect.ownKeys(proxy)`

3. **捕获器处理程序参数**

   - `target`：目标对象。

4. **捕获器不变式**

   返回的可枚举对象必须包含`target`的所有不可配置的自有属性。

   如果`target`不可扩展，则返回可枚举对象必须准确地包含自有属性键。

9.2.8 `getPrototypeOf()`

`getPrototypeOf()`捕获器会在`Object.getPrototypeOf()`中被调用。对应的反射API方法为`Reflect.getPrototypeOf()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  getPrototypeOf(target) {
    console.log('getPrototypeOf()');
    return Reflect.getPrototypeOf(...arguments)
  }
});

Object.getPrototypeOf(proxy);
// getPrototypeOf()
```

1. **返回值**

   `getPrototypeOf()`必须返回对象或`null`。

2. **拦截的操作**

   - `Object.getPrototypeOf(proxy)`
   - `Reflect.getPrototypeOf(proxy)`
   - `proxy.__proto__`
   - `Object.prototype.isPrototypeOf(proxy)`
   - `proxy instanceof Object`

3. **捕获器处理程序参数**

- `target`：目标对象。

4. **捕获器不变式**

   如果`target`不可扩展，则`Object.getPrototypeOf(proxy)`唯一有效的返回值就是`Object.getPrototypeOf(target)`的返回值。

9.2.9 `setPrototypeOf()`

`setPrototypeOf()`捕获器会在`Object.setPrototypeOf()`中被调用。对应的反射API方法为`Reflect.setPrototypeOf()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  setPrototypeOf(target, prototype) {
    console.log('setPrototypeOf()');
    return Reflect.setPrototypeOf(...arguments)
  }
});

Object.setPrototypeOf(proxy, Object);
// setPrototypeOf()
```

1. **返回值**

   `setPrototypeOf()`必须返回布尔值，表示原型赋值是否成功。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

   - `Object.setPrototypeOf(proxy)`
   - `Reflect.setPrototypeOf(proxy)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `prototype`：`target`的替代原型，如果是顶级原型则为`null`。

4. **捕获器不变式**

   如果`target`不可扩展，则唯一有效的`prototype`参数就是`Object.getPrototypeOf(target)`的返回值。

9.2.10 `isExtensible()`

`isExtensible()`捕获器会在`Object.isExtensible()`中被调用。对应的反射API方法为`Reflect.isExtensible()`。

```
const myTarget = {};
```

```
const proxy = new Proxy(myTarget, {
  isExtensible(target) {
    console.log('isExtensible()');
    return Reflect.isExtensible(...arguments)
  }
});

Object.isExtensible(proxy);
// isExtensible()
```

1. **返回值**

   `isExtensible()`必须返回布尔值，表示`target`是否可扩展。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

   - `Object.isExtensible(proxy)`
   - `Reflect.isExtensible(proxy)`

3. **捕获器处理程序参数**

   - `target`：目标对象。

4. **捕获器不变式**

   如果`target`可扩展，则处理程序必须返回`true`。

   如果`target`不可扩展，则处理程序必须返回`false`。

9.2.11 `preventExtensions()`

`preventExtensions()`捕获器会在`Object.preventExtensions()`中被调用。对应的反射API方法为`Reflect.preventExtensions()`。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  preventExtensions(target) {
    console.log('preventExtensions()');
    return Reflect.preventExtensions(...arguments)
  }
});

Object.preventExtensions(proxy);
// preventExtensions()
```

1. **返回值**

   `preventExtensions()`必须返回布尔值，表示`target`是否已经不可扩展。返回非布尔值会被转型为布尔值。

2. **拦截的操作**

   - `Object.preventExtensions(proxy)`
   - `Reflect.preventExtensions(proxy)`

3. **捕获器处理程序参数**

   - `target`：目标对象。

4. **捕获器不变式**

   如果`Object.isExtensible(proxy)`是`false`，则处理程序必须返回`true`。

9.2.12 `apply()`

`apply()`捕获器会在调用函数时中被调用。对应的反射API方法为`Reflect.apply()`。

```
const myTarget = () => {};

const proxy = new Proxy(myTarget, {
  apply(target, thisArg, ...argumentsList) {
    console.log('apply()');
    return Reflect.apply(...arguments)
  }
});

proxy();
// apply()
```

1. **返回值**

   返回值无限制。

2. **拦截的操作**

   - `proxy(...argumentsList)`
   - `Function.prototype.apply(thisArg, argumentsList)`
   - `Function.prototype.call(thisArg, ...argumentsList)`
   - `Reflect.apply(target, thisArgument, argumentsList)`

3. **捕获器处理程序参数**

   - `target`：目标对象。
   - `thisArg`：调用函数时的`this`参数。
   - `argumentsList`：调用函数时的参数列表

4. **捕获器不变式**

   `target`必须是一个函数对象。

9.2.13 `construct()`

construct()捕获器会在new操作符中被调用。对应的反射API方法为Reflect.construct()。

```
const myTarget = function() {};

const proxy = new Proxy(myTarget, {
  construct(target, argumentsList, newTarget) {
    console.log('construct()');
    return Reflect.construct(...arguments)
  }
});

new proxy;
// construct()
```

1. **返回值**

   construct()必须返回一个对象。

2. **拦截的操作**

   - new proxy(...argumentsList)
   - Reflect.construct(target, argumentsList, newTarget)

3. **捕获器处理程序参数**

   - target：目标构造函数。
   - argumentsList：传给目标构造函数的参数列表。
   - newTarget：最初被调用的构造函数。

4. **捕获器不变式**

   target必须可以用作构造函数。

9.3 代理模式

使用代理可以在代码中实现一些有用的编程模式。

9.3.1 跟踪属性访问

通过捕获get、set和has等操作，可以知道对象属性什么时候被访问、被查询。把实现相应捕获器的某个对象代理放到应用中，可以监控这个对象何时在何处被访问过：

```
const user = {
  name: 'Jake'
};

const proxy = new Proxy(user, {
  get(target, property, receiver) {
    console.log('Getting ${property}');
```

```
    return Reflect.get(...arguments);
  },
  set(target, property, value, receiver) {
    console.log('Setting ${property}=${value}');

    return Reflect.set(...arguments);
  }
});

proxy.name;     // Getting name
proxy.age = 27; // Setting age=27
```

9.3.2 隐藏属性

代理的内部实现对外部代码是不可见的，因此要隐藏目标对象上的属性也轻而易举。比如：

```
const hiddenProperties = ['foo', 'bar'];
const targetObject = {
  foo: 1,
  bar: 2,
  baz: 3
};
const proxy = new Proxy(targetObject, {
  get(target, property) {
    if (hiddenProperties.includes(property)) {
      return undefined;
    } else {
      return Reflect.get(...arguments);
    }
  },
  has(target, property) {
    if (hiddenProperties.includes(property)) {
      return false;
    } else {
      return Reflect.has(...arguments);
    }
  }
});

// get()
console.log(proxy.foo);  // undefined
console.log(proxy.bar);  // undefined
console.log(proxy.baz);  // 3

// has()
console.log('foo' in proxy);  // false
console.log('bar' in proxy);  // false
console.log('baz' in proxy);  // true
```

9.3.3 属性验证

因为所有赋值操作都会触发`set()`捕获器，所以可以根据所赋的值决定是允许还是拒绝赋值：

```
const target = {
  onlyNumbersGoHere: 0
};

const proxy = new Proxy(target, {
  set(target, property, value) {
    if (typeof value !== 'Number') {
      return false;
    } else {
      return Reflect.set(...arguments);
    }
  }
});

proxy.onlyNumbersGoHere = 1;
console.log(proxy.onlyNumbersGoHere);  // 1
proxy.onlyNumbersGoHere = '2';
console.log(proxy.onlyNumbersGoHere);  // 1
```

9.3.4 函数与构造函数参数验证

跟保护和验证对象属性类似，也可对函数和构造函数参数进行审查。比如，可以让函数只接收某种类型的值：

```
function median(...nums) {
  return nums.sort()[Math.floor(nums.length / 2)];
}

const proxy = new Proxy(median, {
  apply(target, thisArg, ...argumentsList) {
    for (const arg of argumentsList) {
      if (typeof arg !== 'number') {
        throw 'Non-number argument provided';
      }
    }
    return Reflect.apply(...arguments);
  }
});

console.log(proxy(4, 7, 1));  // 4
console.log(proxy(4, '7', 1));
// Error: Non-number argument provided
```

类似地，可以要求实例化时必须给构造函数传参：

```
class User {
  constructor(id) {
```

```
    this.id_ = id;
  }
}

const proxy = new Proxy(User, {
  construct(target, argumentsList, newTarget) {
    if (argumentsList[0] === undefined) {
      throw 'User cannot be instantiated without id';
    } else {
      return Reflect.construct(...arguments);
    }
  }
});

new proxy(1);

new proxy();
// Error: User cannot be instantiated without id
```

9.3.5 数据绑定与可观察对象

通过代理可以把运行时中原本不相关的部分联系到一起。这样就可以实现各种模式，从而让不同的代码互操作。

比如，可以将被代理的类绑定到一个全局实例集合，让所有创建的实例都被添加到这个集合中：

```
const userList = [];

class User {
  constructor(name) {
    this.name_ = name;
  }
}

const proxy = new Proxy(User, {
  construct() {
    const newUser = Reflect.construct(...arguments);
    userList.push(newUser);
    return newUser;
  }
});

new proxy('John');
new proxy('Jacob');
new proxy('Jingleheimerschmidt');

console.log(userList); // [User {}, User {}, User{}]
```

另外，还可以把集合绑定到一个事件分派程序，每次插入新实例时都会发送消息：

```
const userList = [];

function emit(newValue) {
  console.log(newValue);
}

const proxy = new Proxy(userList, {
  set(target, property, value, receiver) {
    const result = Reflect.set(...arguments);
    if (result) {
      emit(Reflect.get(target, property, receiver));
    }
    return result;
  }
});

proxy.push('John');
// John
proxy.push('Jacob');
// Jacob
```

9.4 小结

代理是ECMAScript 6新增的令人兴奋和动态十足的新特性。尽管不支持向后兼容，但它开辟出了一片前所未有的JavaScript元编程及抽象的新天地。

从宏观上看，代理是真实JavaScript对象的透明抽象层。代理可以定义包含**捕获器**的处理程序对象，而这些捕获器可以拦截绝大部分JavaScript的基本操作和方法。在这个捕获器处理程序中，可以修改任何基本操作的行为，当然前提是遵从**捕获器不变式**。

与代理如影随形的反射API，则封装了一整套与捕获器拦截的操作相对应的方法。可以把反射API看作一套基本操作，这些操作是绝大部分JavaScript对象API的基础。

代理的应用场景是不可限量的。开发者使用它可以创建出各种编码模式，比如（但远远不限于）跟踪属性访问、隐藏属性、阻止修改或删除属性、函数参数验证、构造函数参数验证、数据绑定，以及可观察对象。