第1章

JavaScript 简介

JavaScript 是网络编程语言。绝大多数 网站都使用 JavaScript，所有现代网络浏览器--包括台式机、平板电脑和 手机--都包含 JavaScript 解释器，这使得 JavaScript 成为历史上使用最多的专业 编程语言。在过去十年中，Node.js 使 JavaScript 在网络浏览器之外实现了编程，Node 的巨大成功意味着 JavaScript 现在也是软件开发人员使用最多的编程语言， 。无论您是从零开始，还是已经在专业地使用 JavaScript- nally，本书都将帮助您掌握这门语言。

如果你已经熟悉了其他编程语言， ，可能会对你有所帮助。JavaScript 是一种高级、动态、解释型编程语言， ，非常适合面向对象和函数式编程风格。JavaScript 的变量 ables 是无类型的。它的语法松散地基于 Java，但在其他方面与 毫无关联。JavaScript 的一级函数来源于 Scheme，而基于原型的继承则来源于鲜为人知的语言 Self。但是，你并不需要了解 ，也不需要熟悉这些术语，就可以使用本书学习 JavaScript。

JavaScript "这个名字很容易让人产生误解。除了语法上的表面相似 ，JavaScript 与 Java 编程语言完全不同。 JavaScript早已摆脱了脚本语言的根基，成为一种强大的 、高效的通用语言，适合于严肃的软件工程和拥有庞大代码库的 项目

。

JavaScript：名称、版本和模式

从技术上讲，"Java 脚本 "是 Sun Microsystems（现甲骨文公司）授权的商标，用于描述网景公司（现 Mozilla 公司）的语言实现。网景公司向欧洲计算机制造商协会（ECMA）提交了该语言的标准化申请，但由于商标问题，该语言的标准化版本只能使用 "ECMAScript "这个别扭的名字。在实践中，大家都称这种语言为 JavaScript。本书使用 "ECMAScript "和缩写 "ES "来指代语言标准和该标准的版本。

在 2010 年代的大部分时间里，所有网络浏览器都支持 ECMAScript 标准的第 5 版。本书将 ES5 作为兼容性基线，不再讨论该语言的早期版本。ES6于2015年发布，增加了包括类和模块语法在内的主要新特性，将JavaScript从脚本语言转变为适合大规模软件工程的严肃的通用语言。自 ES6 发布以来，ECMAScript 规范已改为每年发布一次，语言的版本--ES2016、ES2017、ES2018、ES2019 和 ES2020--现在都按发布年份进行标识。

随着 JavaScript 的发展，语言设计者试图修正早期（ES5 之前）版本中的缺陷。为了保持向后兼容性，不可能删除遗留功能，无论这些功能有多大缺陷。但在 ES5 及以后的版本中，程序可以选择加入 JavaScript 的*严格模式，在*这种模式下，许多早期的语言错误都得到了纠正。选择加入的机制是 "use strict"（使用严格）指令。

§5.6.3.该节还总结了传统 JavaScript 与严格 JavaScript 之间的区别。在 ES6 及更高版本中，新语言特性的使用通常会隐式地调用严格模式。例如，如果您使用了 ES6 类关键字或创建了 ES6 模块，那么类或模块中的所有代码都会自动采用严格模式，而那些有缺陷的旧特性则无法在这些上下文中使用。本书将介绍 JavaScript 的遗留特性，但会谨慎地指出这些特性在严格模式下不可用。

每种语言都必须有一个平台或标准库，用于执行基本的输入和输出等功能，这样才能发挥作用。JavaScript 核心语言定义了处理数字、文本、数组、集合、地

图等的小型应用程序接口，但不包括任何输入或输出功能。输入和输出（以及更复杂的功能，如网络、存储和图形）由嵌入 JavaScript 的 "主机环境 "负责。

JavaScript 最初的宿主环境是网络浏览器，这仍然是 JavaScript 代码最常见的执行环境。网络浏览器环境允许 JavaScript 代码从用户的鼠标和键盘上获取输入，并在浏览器上运行。

通过 HTTP 请求。它允许 JavaScript 代码通过 HTML 和 CSS 向用户显示输出结果。

自 2010 年起，JavaScript 代码可以使用另一种主机环境。Node 并不限制 JavaScript 使用网络浏览器提供的 API，而是允许 JavaScript 访问整个操作系统，允许 JavaScript 程序读写文件、通过网络发送和接收数据，以及发出和提供 HTTP 请求。Node 是实现网络服务器的热门选择，也是编写简单实用脚本的便捷工具，可替代 shell 脚本。

本书的大部分内容都集中在 JavaScript 语言本身。第 11 章介绍 JavaScript 标准库，第 15 章介绍网络浏览器主机环境，第 16 章介绍 Node 主机环境。

本书首先介绍低级基础知识，然后在此基础上介绍更高级和更高层次的抽象概念。各章的阅读顺序大致相同。但是，学习一门新的编程语言从来都不是一个线性的过程，而描述一门语言也不是线性的：每种语言的特性都与其他特性相关，本书充满了交叉引用--有时是向后引用，有时是向前引用相关资料。本介绍性章节对语言进行了快速的初步介绍，介绍了一些关键特性，使读者更容易理解后面章节的深入论述。如果你已经是一名 Java 脚本程序员，可以跳过本章。(不过，在继续学习之前，您可能会喜欢阅读本章末尾的例 1-1）。

1.1 探索 JavaScript

在学习一门新的编程语言时，重要的是尝试书中的示例，然后修改它们并再次尝试，以测试你对这门语言的理解。为此，你需要一个 JavaScript 解释器。

要试用几行 JavaScript，最简单的方法是打开浏览器中的网页开发工具（使用 F12、Ctrl-Shift-I 或 Command-Option-I），然后选择控制台选项卡。然后，你就可以在提示符下输入代码，并在输入的同时看到结果。浏览器开发工具通常以窗格形式出现在浏览器窗口的底部或右侧，但通常可以将它们分离成单独的窗口（如图 1-1 所示），这通常非常方便。

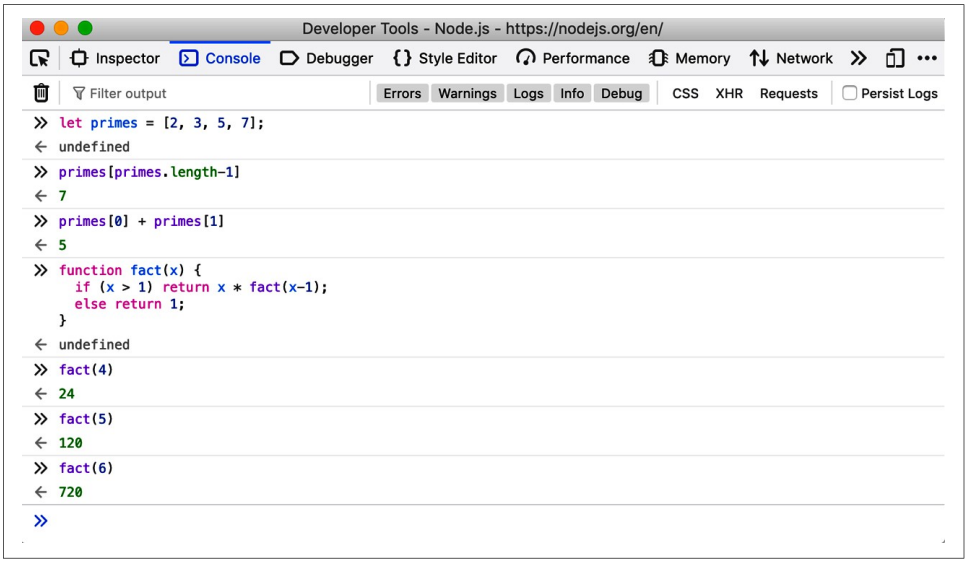

图1-1.火狐浏览器开发工具中的JavaScript 控制台

尝试 JavaScript 代码的另一种方法是从 *https:// nodejs.org* 下载并安装 Node。在系统上安装 Node 后，只需打开终端窗口并键入 **node，就**可以开始像这样的 JavaScript 交互会话：

```
$ 节点
欢迎使用 Node.js v12.13.0。
输入".help "获取更多信息。
> .help
.break有时你会被卡住，这个功能可以帮你摆脱困境
          .clearAlias for .break
.编辑器进入 编辑器模式
.exit     退出 repl
.help打印 此帮助信息
.loadLoad将 文件中的 JS 加载 到 REPL 会话中
.save将    此 REPL 会话中所有已评估的命令保存  到文件中

按 ^C 放弃当前表达式，按 ^D 退出回复。
> 让 x = 2，y = 3；未定义
> x + y
5
> (x === 2) && (y === 3)
真
> (x > 3) || (y < 3)
false
```

1.2 你好，世界

当你准备开始试验较长的代码块时，这些逐行交互式环境可能不再适用，你可能更喜欢在文本编辑器中编写代码。在文本编辑器中，您可以复制并粘贴到 Java 脚本控制台或 Node 会话中。或者，您也可以将代码保存到文件中（JavaScript 代码的传统文件扩展名为 *.js*），然后使用 Node 运行该 JavaScript 代码文件：

```
$ node snippet.js
```

如果像这样以非交互方式使用 Node，它不会自动打印出你运行的所有代码的值，所以你必须自己打印出来。您可以使用函数 `console.log()` 在终端窗口或浏览器的开发工具控制台中显示文本和其他 JavaScript 值。例如，如果您创建了一个 *hello.js* 文件，其中包含这样一行代码：

```
console.log("Hello World!");
```

并用 `node hello.js` 执行该文件，就会看到打印出来的信息 "Hello World!"。

如果您想在网络浏览器的 JavaScript 控制台中看到打印出的相同信息，请创建一个名为 *hello.html* 的新文件，并在其中写入这段文字：

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

然后使用像这样的 `file://` URL 将 *hello.html* 加载到网络浏览器中：

```
file:///Users/username/javascript/hello.html
```

打开开发工具窗口，查看控制台中的问候语。

1.3 JavaScript 之旅

本节通过代码示例快速介绍 JavaScript 语言。本章介绍之后，我们将深入学习 JavaScript 的最底层：第 2 章介绍了 JavaScript 注释、分号和 Unicode 字符集。第 3 章开始变得更加有趣：它解释了 JavaScript 变量和可以为这些变量赋值的值。

下面是一些示例代码，以说明这两章的重点：

```
// 双斜线之后的内容为英文注释。
// 仔细阅读注释：它们解释了 JavaScript 代码。

// 变量是一个值的符号名称。
// 使用 let 关键字声明变量：
let x;                          // 声明一个名为 x 的变量。
```

```
// 可以用 = 符号为变量赋值
x = 0;                          // 现在变量 x 的值为 0
x                               // => 0: 变量的值为其值。

// JavaScript 支持多种类型的值
x = 1;                          // 数字。
x = 0.01;                       // 数字可以是整数或实数。
x = "hello world";              // 带引 号 的文本字符串。
x = 'JavaScript';               // 单引号也可以给字符串定界。
x = true;                       // 布尔值。
x = false;                      // 另一个布尔值。
x = null;                       // Null 是一个特殊值，表示 "无值"。
x = undefined;                  // Undefined 是另一个特殊值，就像 null 一样。
```

JavaScript 程序可以操作的另外两种非常重要的*类型*是对象和数组。它们是第 6 章和第 7 章的主题，但它们非常重要，在学习这两章之前，你会多次看到它们：

```
// JavaScript 最重要的数据类型是对象。
// 对象是名称/值对的集合，或者是字符串到值的映射。
让 book = {                     // 对象用大括号括起来。
    主题: "JavaScript",、       // 属性 "topic "的值为 "JavaScript"。
    版: 7                       // 属性 "edition "的值为 7
};                              // 大括号标志着对象的结束。

// 使用 .或 [] 访问对象的属性：
book.topic                      // => "JavaScript"
book["edition"]                 // book.author = "Flanagan"; // 通过赋值创建
新属性book.contents = {}; // {} 是一个没有属性的空对象。

// 使用"..... "有条件地访问属性。(ES2020)：
book.contents?.ch01?.sect1 // => 未定义：book.contents 没有 ch01 属性。

// JavaScript 还支持数组（数值索引列表）：
let primes = [2, 3, 5, 7]; // 包含 4 个值的数组，以 [ 和 ]分隔。
primes[0]                       // => 2：数组的第一个元素（索引 0）。
primes.length                   primes[primes.length-1] // => 7：数组的
最后一个元素。                  // primes[4] = 9;          // 通过赋值添
加新元素。
primes[4] = 11;                 // 让 empty = [];          // [] 是一个没有元
素的空数组。                    // => 0

                    // 数组和对象可以容纳其他数组和对象：
    let points = [              // 包含 2 个元素的数组。
      {x: 0, y: 0},             // 每个元素都是一个对象。
    {x: 1, y: 1}
];
让 data = {                     // 包含 2 个属性的对象
```

```
    trial1: [[1,2], [3,4]], // 每个属性的值都是一个数组。
```

```
    试用2: [[2,3], [4,5]]    // 数组的元素是数组。
};
```

代码示例中的注释语法

你可能已经注意到，在前面的代码中，有些注释以箭头（=>）开头。这些注释显示了注释前的代码所产生的值，这也是我试图模拟交互式 JavaScript 环境的一种尝试，就像印刷书中的网络浏览器控制台一样。

这些 `// =>` 注释也是一个*断言*，我编写了一个工具来测试代码，验证它是否产生了注释中指定的值。我希望这将有助于减少书中的错误。

注释/断言有两种相关的样式。如果你看到 `// a == 42` 形式的注释，这意味着在注释前的代码运行后，变量 `a` 的值将是 42。如果出现 `// !`形式的注释，则表示注释前一行的代码抛出了异常（感叹号后的其他内容通常解释了抛出的异常类型）。

你会在全书中看到这些评论。

这里说明的在方括号内列出数组元素或在大括号内将对象属性名映射到属性值的语法称为*初始化器表达式*，它只是第 4 章的主题之一。*表达式*是 JavaScript 中的一个短语，可以通过求值产生一个值。例如，使用 `.` 和 `[]` 来指代对象属性或数组元素的值就是一个表达式。

在 JavaScript 中形成表达式最常用的方法之一就是使用*操作符*：

```
// 运算符作用于数值（操作数），产生新的数值。
// 算术运算符是最简单的运算符：
3 + 2                       // => 5：加法
3 - 2                       // => 1：减法
3 * 2                       // => 6：乘法
3 / 2                       // => 1.5：除法
points[1].x - points[0].x // => 1：更复杂的操作数也能工作
"3" + "2"                   // => "32":+ 添加数字，连接字符串

// JavaScript 定义了一些速记算术运算符
let count = 0;              // 定义变量
count++;                    // 增加变量
count--;                    // 减少变量
count += 2;                 // 添加 2：与 count = count + 2 相同;
```

```
count *= 3;                    // 乘以 3：与 count = count * 3 相同;
计数                           // => 6：变量名也是表达式。

// 等价运算符和关系运算符测试两个值是否相等、
```

```
// 不相等、小于、大于等。让 x = 2, y = 3;  // 这些 = 符号是赋值，而不是相等
测试 x === y                => false：相等
x !== y                     // => true：不等式
x < y                       // => true: 小于
x <= y                      // => true：小于或等于
x > y                       // => false：大于
x >= y                      // => false：大于或等于
"two" === "three"           // => false：两个字符串不同
"二">"三"                    // => true: "tw" 按字母顺序大于 "th"
false === (x > y)           // => true：false 等于 false

// 逻辑运算符组合或反转布尔值
(x === 2) && (y === 3)      // => true：两个比较都为真。
(x > 3) || (y < 3)          // => false：两种比较都不为真。|| 是 OR
!(x === y)                  // => true: ！反转布尔值
```

如果 JavaScript 表达式像短语，那么 JavaScript *语句就*像完整的句子。语句是第 5 章的主题。粗略地说，表达式是一种能计算值但不*做*任何事情的东西：它不会以任何方式改变程序状态。而语句没有值，但可以改变程序状态。变量声明和赋值语句已经在上面介绍过了。另一大类语句是*控制结构*，如条件语句和循环语句。我们将在下文介绍函数后再举例说明。

*函数*是一个命名和参数化的 JavaScript 代码块，你只需定义一次，然后就可以反复调用。函数要到第 8 章才正式涉及，但就像对象和数组一样，在这一章之前你会多次看到它们。下面是一些简单的示例：

```
// 函数是我们可以调用的参数化 JavaScript 代码块。
函数 plus1(x) {             // 定义一个参数为 "x"、名为 "plus1 "的函数
    return x + 1;           // 返回一个比传入值大 1 的值
}                           // 函数用大括号括起来

plus1(y)                    // => 4：y 是 3，所以这次调用返回 3+1

let square = function(x) { // 函数是值，可以赋值给变量
    return x * x;           // 计算函数值
};                          // 分号标志着赋值的结束。

square(plus1(y))            // => 16：在一个表达式中调用两个函数
```

在 ES6 及更高版本中，有一种定义函数的速记语法。这种简洁的语法使用 => 来分隔参数列表和函数体，因此以这种方式定义的函数被称为*箭头函数*。箭头函数最常用于将一个未命名的函数作为参数传递给另一个函数。前面的代码在

改写为使用箭头函数后如下所示：

```
const plus1 = x => x + 1;  // 输入 x 映射到输出 x + 1
const square = x => x * x; // 输入 x 映射到输出 x * x
```

```
plus1(y)                    // => 4：函数调用相同
square(plus1(y))            // => 16
```

当我们在对象中使用函数时，就会得到*方法*:

```
// 当函数被赋值给对象的属性时，我们会调用
// 它们是"方法"。 所有 JavaScript 对象（包括数组）都有方法：
let a = [];                 // 创建一个空数组
a.push(1,2,3);              // push() 方法将元素添加到数组中
a.reverse();                // 另一种方法：颠倒元素顺序

// 我们也可以定义自己的方法。this "关键字指的是对象
// 方法的定义对象：在本例中，就是之前的点数组。
points.dist = function() { // 定义计算点间距离的方法
    let p1 = this[0];      // 我们调用的数组的第一个元素
    let p2 = this[1];      // "this "对象的第二个元素
    let a = p2.x-p1.x; // x 坐标差 let b = p2.y-p1.y;
    // y 坐标差 return Math.sqrt(a*a + // 勾股定理
                     b*b); // Math.sqrt() 计算平方根
};
points.dist()               // => Math.sqrt(2)：两个点之间的距离
```

现在，按照承诺，我们将在这里展示一些函数，这些函数的主体展示了常见的 Java 脚本控制结构语句：

```
// JavaScript 语句包括条件和循环，使用的语法是
// C、C++、Java 和其他语言。
function abs(x) {           // 计算绝对值的函数。
    if (x >= 0) {           // if 语句...
        return x;           // 如果比较结果为真，则执行此代码。
    }                       // if 子句到此为止。
    else {                  // 可选的 else 子句执行其代码，如果
        返回 -x;            // 比较结果为假。
    }                       // 每个子句有 1 条语句时，大括号可有可无。
}                           // 注意嵌套在 if/else 中的 return 语句。
abs(-10) === abs(10)        // => true

function sum(array) {       // 计算数组元素之和
    let sum = 0;            // 以初始和为 0 开始。
    for(let x of array) { // 循环数组，将每个元素赋值给 x。
        sum += x;           // 将元素值添加到总和中。
    }                       // 循环结束。
    return sum;             // 返回总和。
}
sum(primes)                 // => 28：前 5 个素数之和 2+3+5+7+11

function factorial(n) {     // 计算阶乘的函数
    let product = 1;        // 从乘积为 1 开始
    while(n > 1) {          // 当 () 中的 expr 为真时，重复 {} 中的语句
        product *= n;       // product = product * n 的快捷方式；
```

```
    n--;                    // n = n - 1 的快捷方式
}                           // 循环结束
```

```
    return product;         // 返回产品
}
阶乘 (4)                    // => 24: 1*4*3*2

function factorial2(n) {    // 使用不同循环的另一个版本
    let i, product = 1;     // 从 1 开始
    for(i=2; i <= n; i++) // 自动将 i 从 2 递增到 n
        product *= i;       // 每次都这样做。单行循环不需要 {}
    return product;         // 返回阶乘
}
factor2(5)                  // => 120: 1*2*3*4*5
```

JavaScript 支持面向对象编程风格，但它与 "经典 "面向对象编程语言有很大不同。第 9 章详细介绍了 JavaScript 中的面向对象编程，并附有大量示例。下面是一个非常简单的示例，演示如何定义一个 JavaScript 类来表示二维几何点。作为该类实例的对象只有一个名为 distance() 的方法，用于计算点到原点的距离：

```
类 点 {                     // 按照惯例，类名大写。
    constructor(x, y) { // 用于初始化新实例的构造函数 this.x = x; // 这个关
        键字是被初始化的新对象 this.y = y; // 将函数参数存储为对象属性。
    }                       // 构造函数中不需要返回值。

    distance() {            // 计算从原点到点的距离的方法。
        return Math.sqrt( // 返回 x² + y² 的平方根。
            this.x * this.x + // 此处指的是点对象，该点对象上有
            this.y * this.y   // 距离方法被调用。
        );
    }
}

// 使用带 "new "的 Point() 构造函数创建点对象
let p = new Point(1, 1);  // 几何点 （1,1）。

// 现在使用点对象 p 的方法
p.distance()                // => Math.SQRT2
```

关于 JavaScript 基本语法和功能的入门介绍到此结束，但本书还将继续以独立章节的形式介绍该语言的其他功能：

第 10 章，模块

展示一个文件或脚本中的 JavaScript 代码如何使用其他文件或脚本中定义的 JavaScript 函数和类。

第 11 章，JavaScript 标准库

涵盖所有 JavaScript 程序都可用的内置函数和类。其中包括重要的数据结构（如映射和集合）、正则表达式（如.....

用于文本模式匹配的表达式类、用于序列化 JavaScript 数据结构的函数等等。

第 12 章，迭代器和生成器

解释 `for/of` 循环的工作原理，以及如何使用 `for/of` 使自己的类可以迭代。还包括生成器函数和 `yield` 语句。

第 13 章 异步 JavaScript

本章将深入探讨 Java 脚本中的异步编程，涵盖回调和事件、基于 Promise 的 API 以及 `async` 和 `await` 关键字。虽然 JavaScript 核心语言不是异步的，但异步 API 是 Web 浏览器和 Node 的默认设置，本章将介绍使用这些 API 的技巧。

第 14 章，元编程

介绍 JavaScript 的一些高级功能，编写供其他 JavaScript 程序员使用的代码库的程序员可能会对这些功能感兴趣。

第 15 章，网络浏览器中的 JavaScript

介绍网络浏览器的主机环境，解释网络浏览器如何输出可爱的 JavaScript 代码，并涵盖网络浏览器定义的众多 API 中最重要的部分。这是本书迄今为止篇幅最长的一章。

第 16 章，使用 Node 的服务器端 JavaScript

介绍 Node 主机环境，涵盖基本编程模型以及最需要了解的数据结构和 API。

第 17 章 JavaScript 工具和扩展

涵盖值得了解的工具和语言扩展，因为它们被广泛使用，并可能使你成为更高效的程序员。

1.4 示例：字符频率直方图字符频率直方图

本章最后将介绍一个简短但并不复杂的 JavaScript 程序。例 1-1 是一个 Node 程序，它从标准输入端读取文本，根据文本计算字符频率直方图，然后打印出直方图。您可以像这样调用该程序来分析其源代码的字符频率：

```
$ node charfreq.js < charfreq.js
T: ########### 11.22%
E:########## 10.15%
R:####### 6.68%
S:###### 6.44%
A: ###### 6.16%
N:###### 5.81%
O:##### 5.45%
I:##### 4.54%
```

```
H: #### 4.07%
C:### 3.36%
L:### 3.20%
U:### 3.08%
/:### 2.88%
```

本示例使用了大量高级 JavaScript 功能，旨在演示真实世界中的 JavaScript 程序。你不应该期望理解所有代码，但请放心，所有代码都将在后面的章节中解释。

例 1-1.用 JavaScript 计算字符频率直方图

```
/**
 * 这个 Node 程序从标准输入端读取文本，然后计算频数。
 * 的直方图，并显示该文本中字母最多的
 * 经常使用的字符。 它需要 Node 12 或更高版本才能运行。
 *
 * 在 Unix 环境中，您可以这样调用程序：
 *     node charfreq.js < corpus.txt
 */

// 该类扩展了 Map，因此 get() 方法会返回指定的
// 当键不在地图中时，用值代替空值
类 DefaultMap extends Map { 构
    造函数 (defaultValue)  {
        super();                           // 调用超类构造函数
        this.defaultValue = defaultValue; // 记住默认值
    }

    get(key) {
                    if (this.has(key)) {              // 如果键已经在地图中
                          return super.get(key);       // 从超类中返回其值。
        }
        否则 {
            return this.defaultValue;     // 否则返回默认值
        }
    }
}

// 该类计算并显示字母频率直方图
类 直方图 { 构造函数( ) {
        this.letterCounts = new DefaultMap(0); // 从字母到计数的映射
        this.totalLetters = 0;                 // 共有多少个字母
    }

                                // 该函数用文本字母更新直方图。
```

```
    add(text) {
                                // 删除文本中的空白，并转换为大写字母
        text = text.replace(/\s/g, "").toUpperCase();
```

```
        // 现在循环查看文本中的字符
        for(let character of text) {
            let count = this.letterCounts.get(character); // Get old count
            this.letterCounts.set(character, count+1);          // 递增
            this.totalLetters++;
        }
    }

                                // 将直方图转换为显示 ASCII 图形的字符串
    toString() {
        // 将地图转换为[key,value]数组的数组
        let entries = [...this.letterCounts];

        // 按计数对数组排序，然后按字母顺序排序
        entries.sort((a,b) => {              // 用于定义排序顺序的函数。
            if (a[1] === b[1]) {             // 如果计数相同
                  return a[0] < b[0] ?-1 : 1; // 按字母顺序排序。
                } else {                          // 如果计数不同
                    return b[1] - a[1];           // 按最大计数排序。
            }
        });

        // 将计数转换为百分比
        for(let entry of entries) {
            entry[1] = entry[1] / this.totalLetters*100;
        }

        // 删除小于 1%的条目
        entries = entries.filter(entry => entry[1] >= 1);

        // 现在将每个条目转换为一行文本
        let lines = entries.map(
            ([l,n]) => `${l}: ${"#".repeat(Math.round(n))} ${n.toFixed(2)}%`
        );

        // 并返回以换行符分隔的连接行。
        return lines.join("\n");
    }
}

// 这个异步（承诺返回）函数创建一个直方图对象、
// 从标准输入中异步读取文本块，并将这些文本块添加到
//直方图。当它到达数据流的末端时，就会返回这个直方图
async 函数 histogramFromStdin() {
    process.stdin.setEncoding("utf-8"); // 读取 Unicode 字符串，而不是字节
    让 histogram = new Histogram();
    for await (let chunk of process.stdin) {
        histogram.add(chunk);
    }
    返回直方图;
```

```
}
```

```
// 最后一行代码是程序的主体。
// 它从标准输入创建直方图对象，然后打印直方图。
histogramFromStdin().then(histogram => { console.log(histogram.toString()); });
```

1.5 摘要

本书自下而上地讲解 JavaScript。也就是说，我们从注释、标识符、变量和类型等低级细节开始，然后逐步深入到表达式、语句、对象和函数，最后涵盖类和模块等高级语言抽象。我非常重视本书标题中的 "*明确*"一词，在接下来的章节中，我将详细讲解这门语言，一开始可能会让人感觉有些生疏。不过，要真正掌握 JavaScript，就必须了解其中的细节，我希望你能抽出时间从头到尾读完本书。但请不要觉得第一次阅读就需要这样做。如果你发现自己在某一章节感到困惑，只需跳到下一章节。当你对整个语言有了一定的了解后，再回来掌握细节。

第2章

词汇结构

编程语言的词法结构是一组基本规则，规定了如何用该语言编写程序。它是一门语言最底层的语法：例如，它规定了变量名的样子、连接词的分隔符，以及一条程序语句与下一条程序语句的分隔方式。本章将介绍 JavaScript 的词法结构。内容包括

- 大小写敏感性、空格和换行符
- 评论
- 文字
- 标识符和保留字
- 统一码
- 可选的分号

2.1 JavaScript 程序文本

JavaScript 是一种区分大小写的语言。这意味着语言关键字、变量、函数名和其他*标识符*必须始终使用一致的字母大写字母。例如，`while` 关键字必须输入 "while"，而不是 "While" 或 "WHILE"。同样，`online`、`Online`、`OnLine` 和 `ONLINE` 是四个不同的变量名。

JavaScript会忽略程序中标记之间的空格。在大多数情况下，JavaScript也会忽略换行符（但请参见第2.6节中的例外情况）。由于可以在程序中自由使用空格和换行符，因此可以以整齐一致的方式格式化和缩进程序，使代码易于阅读和理解。

除了常规的空格字符（\u0020）之外，JavaScript 还能将制表符、各种 ASCII 控制字符以及各种 Unicode 空格字符识别为空白。JavaScript 还能将换行符、回车符和回车/换行序列识别为行结束符。

2.2 评论

JavaScript 支持两种注释样式。在//和行尾之间的任何文本都被视为注释，并被 JavaScript 忽略。字符 /* 和 */ 之间的任何文本也被视为注释；这些注释可以跨越多行，但不能嵌套。以下代码行都是合法的 JavaScript 注释：

```
// 这是一个单行注释。

/* 这也是一个注释 */ // 这里是另一个注释。

/*
 * 这是一个多行注释。在
 * 并不是语法的必要组成部分，只是看起来很酷而已！
 */
```

2.3 文字

*字面量*是直接出现在程序中的数据值。以下都是字面量：

```
12              // 数字 12
1.2             // 数字一点二
"hello world    // 一串文本
嗨              // 另一个字符串
true            // 布尔值
假              // 另一个布尔值
无效            // 没有对象
```

有关数字和字符串字面量的完整详细信息，请参阅第 3 章。

2.4 标识符和保留字

*标识符*只是一个名称。在 JavaScript 中，标识符用于命名常量、变量、属性、函数和类，并为 JavaScript 代码中的某些循环提供标签。JavaScript 标识符必须

以字母、下划线 (_) 或美元符号 ($) 开头。后面的字符可以是字母、数字、下划线或美元符号。(数字不允许作为第一个字符，以便 JavaScript 可以轻松区分标识符和数字）。这些都是合法的标识符：

```
i
my_variable_name
v13
我的天
$str
```

与其他语言一样，JavaScript 保留了某些标识符供语言本身使用。这些 "保留字 "不能作为常规标识符使用。它们将在下一节中列出。

2.4.1 保留字

以下单词是 JavaScript 语言的一部分。其中许多（如 if、while 和 for）是保留关键字，不得用作常量、变量、函数或类的名称（不过它们都可以用作对象中的属性名称）。其他关键字（如 from、of、get 和 set）只在有限的上下文中使用，语法上没有歧义，完全可以作为标识符使用。还有一些关键字（如 let）不能完全保留，以保持与旧程序的向后兼容性，因此有一些复杂的规则规定它们何时可以用作标识符，何时不能。(例如，如果在类外使用 var 声明，let 可以用作变量名，但如果在类内或使用 const 声明，let 就不能用作变量名）。最简单的方法是避免将这些词用作标识符，但 from、set 和 target 除外。

```
作为    缢      出口    获取    无效    目标    空白
异步    继续    延长    如果    的      此      虽然
等      调试器  错误    舶来品  返回    丢      与
断裂    默认    总算    于      设置    真      屈服
个案    删去    对于    实例    天电    尝试
捕捉    做      从      告诉    棒极了  typeof
类      不然    功能    新      开关    变异
```

JavaScript 还保留或限制使用某些关键字，这些关键字目前未被该语言使用，但可能会在未来版本中使用：

```
枚举实现接口 包 私有
```

由于历史原因，在某些情况下不允许使用 arguments 和 eval 作为标识符，最好完全避免使用。

2.5 统一码

JavaScript 程序是使用 Unicode 字符集编写的，因此可以在字符串和注释中使用任何 Unicode 字符。为了便于移植和编辑，通常只在标识符中使用 ASCII 字母和数字。但这只是一种编程习惯，JavaScript 语言允许使用 Unicode 字母、数字和表意文字（但

而不是表情符号）作为标识符。这意味着程序员可以使用非英语语言的数学符号和单词作为常量和变量：

```
const π = 3.14;
const sí = true;
```

2.5.1 统一字符编码转义序列

有些计算机硬件和软件无法显示、输入或正确处理全套 Unicode 字符。为了支持使用旧技术的程序员和系统，JavaScript 定义了转义序列，允许我们仅使用 ASCII 字符编写 Unicode 字符。这些 Unicode 转义字符以字符 `\u` 开头，后面有四个十六进制数字（使用大写或小写字母 A-F）或一至六个十六进制数字，并用大括号括起来。这些 Unicode 转义字符可以出现在 JavaScript 字符串字面量、正则表达式字面量和标识符中（但不能出现在语言关键字中）。例如，字符 "é "的 Unicode 转义符是 `\u00E9`；下面是包含该字符的变量名的三种不同写法：

```
let café = 1; // 使用 Unicode 字符定义变量 caf\u00e9 // => 1; 使用
转义序列访问变量 caf\u{E9}  // => 1; 相同转义序列的另一种形式
```

早期版本的 JavaScript 只支持四位数转义序列。ES6 引入了带大括号的版本，以更好地支持需要 16 位以上的 Unicode 编码点，如表情符号：

```
console.log("\u{1F600}"); // 打印一个笑脸表情符号
```

Unicode 转义字符也可能出现在注释中，但由于注释会被忽略，因此在这种情况下它们只被视为 ASCII 字符，而不会被解释为 Unicode 字符。

2.5.2 统一码规范化

如果在 JavaScript 程序中使用非 ASCII 字符，必须注意 Unicode 允许对同一字符进行多种编码。例如，字符串 "é "可以编码为单个 Unicode 字符 `\u00E9`，也可以编码为普通 ASCII "e"，后跟锐尖重音组合标记 `\u0301`。这两种编码在文本编辑器中显示时通常看起来完全一样，但它们的二进制编码却不同，这意味着

JavaScript 认为它们是不同的，这 会 导致程序非常混乱：

```
const café = 1; // 该常量名为 "caf\u{e9}"
const café = 2; // 这个常量是不同的： "cafe\u{301}"
咖啡馆 // => 1：该常数只有一个值
咖啡馆 // => 2：这个无法区分的常量具有不同的值
```

Unicode 标准定义了所有字符的首选编码，并规定了将文本转换为适合比较的规范形式的规范化程序。JavaScript 假定其解释的源代码已经过规范化处理，因此*不会*自行进行任何规范化处理。如果打算在 JavaScript 程序中使用 Unicode 字符，应确保编辑器或其他工具对源代码进行 Unicode 规范化处理，以防止出现不同但视觉上无法区分的标识符。

2.6 可选分号

与许多编程语言一样，JavaScript 使用分号（;）来分隔语句（参见第 5 章）。这对于明确代码的含义非常重要：如果没有分隔符，一条语句的结尾可能看起来像是下一条语句的开头，反之亦然。在 JavaScript 中，如果两条语句分别写在不同的行上，通常可以省略它们之间的分号。(在程序末尾，或者程序中的下一个标记是结尾大括号时，也可以省略分号：}.)许多 JavaScript 程序员（以及本书中的代码）都使用分号来明确标记语句的结尾，即使在不需要分号的地方也是如此。另一种风格是尽可能省略分号，只在少数需要分号的情况下使用。无论你选择哪种方式，都应该了解 JavaScript 中可选分号的一些细节。

请看下面的代码。由于这两条语句分别出现在不同的行上，因此第一个分号可以省略：

```
a = 3;
b = 4;
```

不过，如下所示，第一个分号是必需的：

```
a = 3; b = 4;
```

需要注意的是，JavaScript 并不是将每一个换行符都视为分号：通常只有在不添加隐式分号就无法解析代码的情况下，JavaScript 才会将换行符视为分号。更正式地说（除了稍后描述的三种例外情况），如果下一个非空格字符不能被解释为当前语句的继续，JavaScript 就会将换行符视为分号。请看下面这段代码

```
let a
a
=
```

```
3
console.log(a)
```

JavaScript 是这样解释这段代码的

```
let a; a = 3; console.log(a);
```

JavaScript 将第一个换行符视为分号，因为它无法解析没有分号的 `a a` 代码。第二个 `a` 可以单独成为语句 `a;`，但 JavaScript 不会将第二个换行符视为分号，因为它可以继续解析更长的语句 `a = 3;`。

这些语句终止规则会导致一些令人惊讶的情况。这段代码看起来像两条独立的语句，中间用换行分隔：

```
let y = x + f
(a+b).toString()
```

但是，第二行代码中的括号可以解释为第一行代码中 `f` 的函数调用，JavaScript 是这样解释代码的：

```
让 y = x + f(a+b).toString();
```

这很可能不是代码编写者的本意。为了作为两个独立的语句运行，在这种情况下需要使用明确的分号。

一般来说，如果语句以（、[、/、+或-开头，就有可能被解释为前面语句的延续。以 /、+ 和 - 开头的语句在实践中非常少见，但以 (和 [开头的语句却并不少见，至少在某些 JavaScript 编程风格中是这样。有些程序员喜欢在此类语句的开头加上一个防御性分号，这样即使前面的语句被修改，前面的分号被删除，语句也能继续正常工作：

```
让 x = 0                                // 此处省略分号
[x,x+1,x+2].forEach(console.log) // 防卫性；使该语句保持独立
```

一般来说，JavaScript 在无法将第二行解析为第一行语句的 继续时，会将换行符解释为分号，但有三种例外情况。第一个例外涉及 `return`、`throw`、`yield`、`break` 和 `continue` 语句（参见第 5 章）。这些语句通常是单独存在的，但有时后面会跟一个标识符或表达式。如果换行符出现在这些语句之后（任何其他标记之前），JavaScript 总是会将换行符解释为分号。例如，如果您写道

```
返回
true;
```

JavaScript 假设你是指

```
return; true;
```

不过，你的意思可能是

```
返回 true;
```

这意味着在 return、break 或 continue 与关键字后面的表达式之间不能插入换行符。如果你插入了换行符，你的代码很 可能会以一种难以调试的非明显方式失败。

第二个例外涉及 ++ 和 -- 操作符（第 4.8 节）。这些运算符可以是出现在表达式之前的前缀运算符，也可以是出现在表达式之后的后缀运算符。如果要将这两个运算符用作后缀运算符，它们必须与所应用的表达式位于同一行。第三个例外涉及使用简洁的 "箭头 "语法定义的函数：=> 箭头本身必须与参数列表位于同一行。

2.7 摘要

本章介绍了 JavaScript 程序的最底层编写方式。下一章将更上一层楼，介绍作为 Java 脚本程序基本计算单位的基元类型和值（数字、字符串等）。