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`；
- 两个命名参数不能拥有同一个名称。

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。

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。

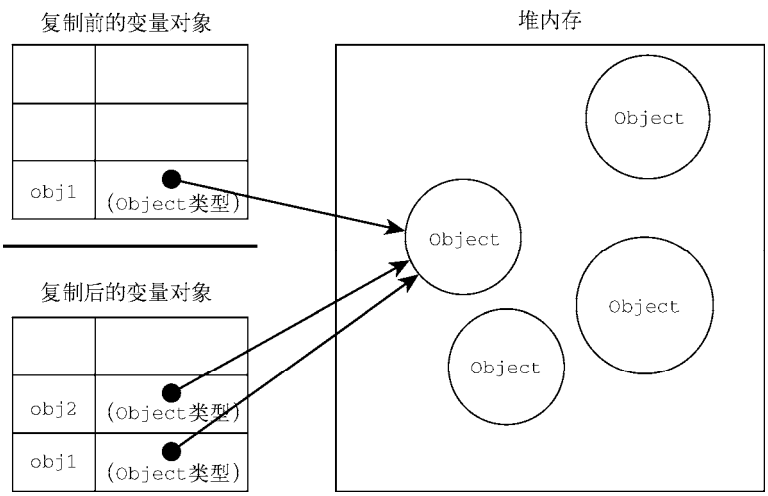

图 4-2

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;
```