第4章

变量、作用域和内存问题

本章内容

- 理解基本类型和引用类型的值
- 理解执行环境
- 理解垃圾收集

按照 ECMA-262 的定义，JavaScript 的变量与其他语言的变量有很大区别。JavaScript 变量松散类型的本质，决定了它只是在特定时间用于保存特定值的一个名字而已。由于不存在定义某个变量必须要保存何种数据类型值的规则，变量的值及其数据类型可以在脚本的生命周期内改变。尽管从某种角度看，这可能是一个既有趣又强大，同时又容易出问题的特性，但 JavaScript 变量实际的复杂程度还远不止如此。

4.1 基本类型和引用类型的值

ECMAScript 变量可能包含两种不同数据类型的值：基本类型值和引用类型值。**基本类型值**指的是简单的数据段，而**引用类型值**指那些可能由多个值构成的对象。

在将一个值赋给变量时，解析器必须确定这个值是基本类型值还是引用类型值。第 3 章讨论了 5 种基本数据类型：`Undefined`、`Null`、`Boolean`、`Number` 和 `String`。这 5 种基本数据类型是按值访问的，因为可以操作保存在变量中的实际的值。

引用类型的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中的位置，也就是说不能直接操作对象的内存空间。在操作对象时，实际上是在操作对象的引用而不是实际的对象。为此，引用类型的值是按引用访问的。

在很多语言中，字符串以对象的形式来表示，因此被认为是引用类型的。ECMAScript 放弃了这一传统。

4.1.1 动态的属性

定义基本类型值和引用类型值的方式是类似的：创建一个变量并为该变量赋值。但是，当这个值保存到变量中以后，对不同类型值可以执行的操作则大相径庭。对于引用类型的值，我们可以为其添加属性和方法，也可以改变和删除其属性和方法。请看下面的例子：

```
var person = new Object();
person.name = "Nicholas";
```

```
alert(person.name);      //"Nicholas"
```

DynamicPropertiesExample01.htm

以上代码创建了一个对象并将其保存在了变量 person 中。然后，我们为该对象添加了一个名为 name 的属性，并将字符串值"Nicholas"赋给了这个属性。紧接着，又通过 alert()函数访问了这个新属性。如果对象不被销毁或者这个属性不被删除，则这个属性将一直存在。

但是，我们不能给基本类型的值添加属性，尽管这样做不会导致任何错误。比如：

```
var name = "Nicholas";
name.age = 27;
alert(name.age);       //undefined
```

DynamicPropertiesExample02.htm

在这个例子中，我们为字符串 name 定义了一个名为 age 的属性，并为该属性赋值 27。但在下一行访问这个属性时，发现该属性不见了。这说明只能给引用类型值动态地添加属性，以便将来使用。

4.1.2 复制变量值

除了保存的方式不同之外，在从一个变量向另一个变量复制基本类型值和引用类型值时，也存在不同。如果从一个变量向另一个变量复制基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上。来看一个例子：

```
var num1 = 5;
var num2 = num1;
```

在此，num1 中保存的值是 5。当使用 num1 的值来初始化 num2 时，num2 中也保存了值 5。但 num2 中的 5 与 num1 中的 5 是完全独立的，该值只是 num1 中 5 的一个副本。此后，这两个变量可以参与任何操作而不会相互影响。图 4-1 形象地展示了复制基本类型值的过程。

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

当从一个变量向另一个变量复制引用类型的值时，同样也会将存储在变量对象中的值复制一份放到为新变量分配的空间中。不同的是，这个值的副本实际上是一个指针，而这个指针指向存储在堆中的一个对象。复制操作结束后，两个变量实际上将引用同一个对象。因此，改变其中一个变量，就会影响另一个变量，如下面的例子所示：

```
var obj1 = new Object();
var obj2 = obj1;
obj1.name = "Nicholas";
alert(obj2.name);  //"Nicholas"
```

首先，变量 obj1 保存了一个对象的新实例。然后，这个值被复制到了 obj2 中；换句话说，obj1 和 obj2 都指向同一个对象。这样，当为 obj1 添加 name 属性后，可以通过 obj2 来访问这个属性，因为这两个变量引用的都是同一个对象。图 4-2 展示了保存在变量对象中的变量和保存在堆中的对象之间的这种关系。

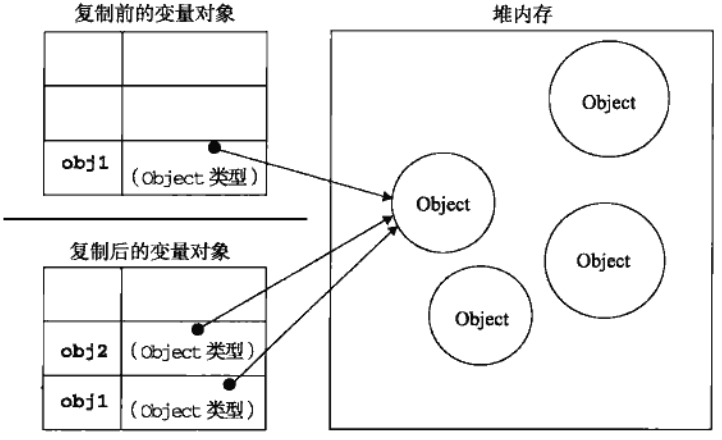

图　4-2

4.1.3　传递参数

ECMAScript 中所有函数的参数都是按值传递的。也就是说，把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。基本类型值的传递如同基本类型变量的复制一样，而引用类型值的传递，则如同引用类型变量的复制一样。有不少开发人员在这一点上可能会感到困惑，因为访问变量有按值和按引用两种方式，而参数只能按值传递。

在向参数传递基本类型的值时，被传递的值会被复制给一个局部变量（即命名参数，或者用 ECMAScript 的概念来说，就是 arguments 对象中的一个元素）。在向参数传递引用类型的值时，会把这个值在内存中的地址复制给一个局部变量，因此这个局部变量的变化会反映在函数的外部。请看下面这个例子：

```
function addTen(num) {
    num += 10;
    return num;
}
```

```
var count = 20;
var result = addTen(count);
alert(count);     //20，没有变化
alert(result);    //30
```

FunctionArgumentsExample01.htm

这里的函数 addTen()有一个参数 num，而参数实际上是函数的局部变量。在调用这个函数时，变量 count 作为参数被传递给函数，这个变量的值是 20。于是，数值 20 被复制给参数 num 以便在 addTen()中使用。在函数内部，参数 num 的值被加上了 10，但这一变化不会影响函数外部的 count 变量。参数 num 与变量 count 互不相识，它们仅仅是具有相同的值。假如 num 是按引用传递的话，那么变量 count 的值也将变成 30，从而反映函数内部的修改。当然，使用数值等基本类型值来说明按值传递参数比较简单，但如果使用对象，那问题就不怎么好理解了。再举一个例子：


```
function setName(obj) {
    obj.name = "Nicholas";
}

var person = new Object();
setName(person);
alert(person.name);     //"Nicholas"
```

FunctionArgumentsExample02.htm

以上代码中创建一个对象，并将其保存在了变量 person 中。然后，这个对象被传递到 setName()函数中之后就被复制给了 obj。在这个函数内部，obj 和 person 引用的是同一个对象。换句话说，即使这个对象是按值传递的，obj 也会按引用来访问同一个对象。于是，当在函数内部为 obj 添加 name 属性后，函数外部的 person 也将有所反映；因为 person 指向的对象在堆内存中只有一个，而且是全局对象。有很多开发人员错误地认为：在局部作用域中修改的对象会在全局作用域中反映出来，就说明参数是按引用传递的。为了证明对象是按值传递的，我们再看一看下面这个经过修改的例子：

```
function setName(obj) {
    obj.name = "Nicholas";
    obj = new Object();
    obj.name = "Greg";
}

var person = new Object();
setName(person);
alert(person.name);     //"Nicholas"
```

这个例子与前一个例子的唯一区别，就是在 setName()函数中添加了两行代码：一行代码为 obj 重新定义了一个对象，另一行代码为该对象定义了一个带有不同值的 name 属性。在把 person 传递给 setName()后，其 name 属性被设置为"Nicholas"。然后，又将一个新对象赋给变量 obj，同时将其 name 属性设置为"Greg"。如果 person 是按引用传递的，那么 person 就会自动被修改为指向其 name 属性值为"Greg"的新对象。但是，当接下来再访问 person.name 时，显示的值仍然是"Nicholas"。这说明即使在函数内部修改了参数的值，但原始的引用仍然保持未变。实际上，当在函数内部重写 obj 时，这个变量引用的就是一个局部对象了。而这个局部对象会在函数执行完毕后立即被销毁。

可以把 ECMAScript 函数的参数想象成局部变量。

4.1.4　检测类型

要检测一个变量是不是基本数据类型？第 3 章介绍的 typeof 操作符是最佳的工具。说得更具体一点，typeof 操作符是确定一个变量是字符串、数值、布尔值，还是 undefined 的最佳工具。如果变量的值是一个对象或 null，则 typeof 操作符会像下面例子中所示的那样返回"object"：

```
var s = "Nicholas";
var b = true;
var i = 22;
var u;
var n = null;
var o = new Object();

alert(typeof s);    //string
alert(typeof i);    //number
alert(typeof b);    //boolean
alert(typeof u);    //undefined
alert(typeof n);    //object
alert(typeof o);    //object
```

DeterminingTypeExample01.htm

虽然在检测基本数据类型时 typeof 是非常得力的助手，但在检测引用类型的值时，这个操作符的用处不大。通常，我们并不是想知道某个值是对象，而是想知道它是什么类型的对象。为此，ECMAScript 提供了 instanceof 操作符，其语法如下所示：

```
result = variable instanceof constructor
```

如果变量是给定引用类型（根据它的原型链来识别；第 6 章将介绍原型链）的实例，那么 instanceof 操作符就会返回 true。请看下面的例子：

```
alert(person instanceof Object);    // 变量 person 是 Object 吗?
alert(colors instanceof Array);     // 变量 colors 是 Array 吗?
alert(pattern instanceof RegExp);   // 变量 pattern 是 RegExp 吗?
```

根据规定，所有引用类型的值都是 Object 的实例。因此，在检测一个引用类型值和 Object 构造函数时，instanceof 操作符始终会返回 true。当然，如果使用 instanceof 操作符检测基本类型的值，则该操作符始终会返回 false，因为基本类型不是对象。

使用 typeof 操作符检测函数时，该操作符会返回"function"。在 Safari 5 及之前版本和 Chrome 7 及之前版本中使用 typeof 检测正则表达式时，由于规范的原因，这个操作符也返回"function"。ECMA-262 规定任何在内部实现[[Call]]方法的对象都应该在应用 typeof 操作符时返回"function"。由于上述浏览器中的正则表达式也实现了这个方法，因此对正则表达式应用 typeof 会返回"function"。在 IE 和 Firefox 中，对正则表达式应用 typeof 会返回"object"。

4.2 执行环境及作用域

执行环境（execution context，为简单起见，有时也称为“环境”）是 JavaScript 中最为重要的一个概念。执行环境定义了变量或函数有权访问的其他数据，决定了它们各自的行为。每个执行环境都有一个与之关联的变量对象（variable object），环境中定义的所有变量和函数都保存在这个对象中。虽然我们编写的代码无法访问这个对象，但解析器在处理数据时会在后台使用它。

全局执行环境是最外围的一个执行环境。根据 ECMAScript 实现所在的宿主环境不同，表示执行环境的对象也不一样。在 Web 浏览器中，全局执行环境被认为是 window 对象（第 7 章将详细讨论），因此所有全局变量和函数都是作为 window 对象的属性和方法创建的。某个执行环境中的所有代码执行完毕后，该环境被销毁，保存在其中的所有变量和函数定义也随之销毁（全局执行环境直到应用程序退出——例如关闭网页或浏览器——时才会被销毁）。

每个函数都有自己的执行环境。当执行流进入一个函数时，函数的环境就会被推入一个环境栈中。而在函数执行之后，栈将其环境弹出，把控制权返回给之前的执行环境。ECMAScript 程序中的执行流正是由这个方便的机制控制着。

当代码在一个环境中执行时，会创建变量对象的一个作用域链（scope chain）。作用域链的用途，是保证对执行环境有权访问的所有变量和函数的有序访问。作用域链的前端，始终都是当前执行的代码所在环境的变量对象。如果这个环境是函数，则将其活动对象（activation object）作为变量对象。活动对象在最开始时只包含一个变量，即 arguments 对象（这个对象在全局环境中是不存在的）。作用域链中的下一个变量对象来自包含（外部）环境，而再下一个变量对象则来自下一个包含环境。这样，一直延续到全局执行环境；全局执行环境的变量对象始终都是作用域链中的最后一个对象。

标识符解析是沿着作用域链一级一级地搜索标识符的过程。搜索过程始终从作用域链的前端开始，然后逐级地向后回溯，直至找到标识符为止（如果找不到标识符，通常会导致错误发生）。

请看下面的示例代码：

```
var color = "blue";

function changeColor(){
    if (color === "blue"){
        color = "red";
    } else {
        color = "blue";
    }
}

changeColor();

alert("Color is now " + color);
```

ExecutionContextExample01.htm

在这个简单的例子中，函数 changeColor()的作用域链包含两个对象：它自己的变量对象（其中定义着 arguments 对象）和全局环境的变量对象。可以在函数内部访问变量 color，就是因为可以在这个作用域链中找到它。

此外，在局部作用域中定义的变量可以在局部环境中与全局变量互换使用，如下面这个例子所示：

```
var color = "blue";

function changeColor(){
    var anotherColor = "red";

    function swapColors(){
        var 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，但不能访问 tempColor
    swapColors();
}

// 这里只能访问 color
changeColor();
```

以上代码共涉及 3 个执行环境：全局环境、changeColor()的局部环境和 swapColors()的局部环境。全局环境中有一个变量 color 和一个函数 changeColor()。changeColor()的局部环境中有一个名为 anotherColor 的变量和一个名为 swapColors()的函数，但它也可以访问全局环境中的变量 color。swapColors()的局部环境中有一个变量 tempColor，该变量只能在这个环境中访问到。无论全局环境还是 changeColor()的局部环境都无权访问 tempColor。然而，在 swapColors()内部则可以访问其他两个环境中的所有变量，因为那两个环境是它的父执行环境。图 4-3 形象地展示了前面这个例子的作用域链。

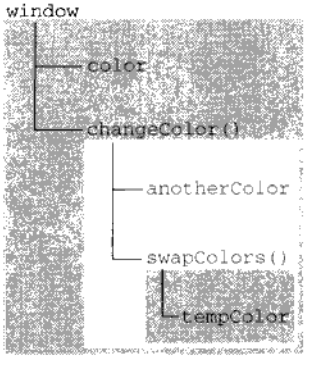

图　4-3

图 4-3 中的矩形表示特定的执行环境。其中，内部环境可以通过作用域链访问所有的外部环境，但外部环境不能访问内部环境中的任何变量和函数。这些环境之间的联系是线性、有次序的。每个环境都可以向上搜索作用域链，以查询变量和函数名；但任何环境都不能通过向下搜索作用域链而进入另一个执行环境。对于这个例子中的 swapColors()而言，其作用域链中包含 3 个对象：swapColors()的变量对象、changeColor()的变量对象和全局变量对象。swapColors()的局部环境开始时会先在自己的变量对象中搜索变量和函数名，如果搜索不到则再搜索上一级作用域链。changeColor()的作用域链

中只包含两个对象：它自己的变量对象和全局变量对象。这也就是说，它不能访问 swapColors()的环境。

函数参数也被当作变量来对待，因此其访问规则与执行环境中的其他变量相同。

4.2.1 延长作用域链

虽然执行环境的类型总共只有两种——全局和局部（函数），但还是有其他办法来延长作用域链。这么说是因为有些语句可以在作用域链的前端临时增加一个变量对象，该变量对象会在代码执行后被移除。在两种情况下会发生这种现象。具体来说，就是当执行流进入下列任何一个语句时，作用域链就会得到加长：

- try-catch 语句的 catch 块；
- with 语句。

这两个语句都会在作用域链的前端添加一个变量对象。对 with 语句来说，会将指定的对象添加到作用域链中。对 catch 语句来说，会创建一个新的变量对象，其中包含的是被抛出的错误对象的声明。下面看一个例子。

```
function buildUrl() {
    var qs = "?debug=true";

    with(location){
        var url = href + qs;
    }

    return url;
}
```

ExecutionContextExample03.htm

在此，with 语句接收的是 location 对象，因此其变量对象中就包含了 location 对象的所有属性和方法，而这个变量对象被添加到了作用域链的前端。buildUrl()函数中定义了一个变量 qs。当在 with 语句中引用变量 href 时（实际引用的是 location.href），可以在当前执行环境的变量对象中找到。当引用变量 qs 时，引用的则是在 buildUrl()中定义的那个变量，而该变量位于函数环境的变量对象中。至于 with 语句内部，则定义了一个名为 url 的变量，因而 url 就成了函数执行环境的一部分，所以可以作为函数的值被返回。

在 IE8 及之前版本的 JavaScript 实现中，存在一个与标准不一致的地方，即在 catch 语句中捕获的错误对象会被添加到执行环境的变量对象，而不是 catch 语句的变量对象中。换句话说，即使是在 catch 块的外部也可以访问到错误对象。IE9 修复了这个问题。

4.2.2　没有块级作用域

JavaScript 没有块级作用域经常会导致理解上的困惑。在其他类 C 的语言中，由花括号封闭的代码块都有自己的作用域（如果用 ECMAScript 的话来讲，就是它们自己的执行环境），因而支持根据条件来定义变量。例如，下面的代码在 JavaScript 中并不会得到想象中的结果：

```
if (true) {
    var color = "blue";
}

alert(color);    //"blue"
```

这里是在一个 if 语句中定义了变量 color。如果是在 C、C++或 Java 中，color 会在 if 语句执行完毕后被销毁。但在 JavaScript 中，if 语句中的变量声明会将变量添加到当前的执行环境（在这里是全局环境）中。在使用 for 语句时尤其要牢记这一差异，例如：

```
for (var i=0; i < 10; i++){
    doSomething(i);
}

alert(i);      //10
```

对于有块级作用域的语言来说，for 语句初始化变量的表达式所定义的变量，只会存在于循环的环境之中。而对于 JavaScript 来说，由 for 语句创建的变量 i 即使在 for 循环执行结束后，也依旧会存在于循环外部的执行环境中。

1. 声明变量

使用 var 声明的变量会自动被添加到最接近的环境中。在函数内部，最接近的环境就是函数的局部环境；在 with 语句中，最接近的环境是函数环境。如果初始化变量时没有使用 var 声明，该变量会自动被添加到全局环境。如下所示：

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

var result = add(10, 20);  //30
alert(sum);                //由于 sum 不是有效的变量，因此会导致错误
```

ExecutionContextExample04.htm

以上代码中的函数 add()定义了一个名为 sum 的局部变量，该变量包含加法操作的结果。虽然结果值从函数中返回了，但变量 sum 在函数外部是访问不到的。如果省略这个例子中的 var 关键字，那么当 add()执行完毕后，sum 也将可以访问到：

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

var result = add(10, 20);  //30
alert(sum);                //30
```

ExecutionContextExample05.htm

这个例子中的变量 sum 在被初始化赋值时没有使用 var 关键字。于是，当调用完 add()之后，添加到全局环境中的变量 sum 将继续存在；即使函数已经执行完毕，后面的代码依旧可以访问它。

在编写 JavaScript 代码的过程中，不声明而直接初始化变量是一个常见的错误做法，因为这样可能会导致意外。我们建议在初始化变量之前，一定要先声明，这样就可以避免类似问题。在严格模式下，初始化未经声明的变量会导致错误。

2. 查询标识符

当在某个环境中为了读取或写入而引用一个标识符时，必须通过搜索来确定该标识符实际代表什么。搜索过程从作用域链的前端开始，向上逐级查询与给定名字匹配的标识符。如果在局部环境中找到了该标识符，搜索过程停止，变量就绪。如果在局部环境中没有找到该变量名，则继续沿作用域链向上搜索。搜索过程将一直追溯到全局环境的变量对象。如果在全局环境中也没有找到这个标识符，则意味着该变量尚未声明。

通过下面这个示例，可以理解查询标识符的过程：

```
var color = "blue";

function getColor(){
    return color;
}

alert(getColor());  //"blue"
```

ExecutionContextExample06.htm

调用本例中的函数 getColor()时会引用变量 color。为了确定变量 color 的值，将开始一个两步的搜索过程。首先，搜索 getColor()的变量对象，查找其中是否包含一个名为 color 的标识符。在没有找到的情况下，搜索继续到下一个变量对象（全局环境的变量对象），然后在那里找到了名为 color 的标识符。因为搜索到了定义这个变量的变量对象，搜索过程宣告结束。图 4-4 形象地展示了上述搜索过程。

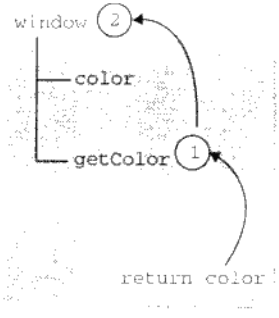

图 4-4

在这个搜索过程中，如果存在一个局部的变量的定义，则搜索会自动停止，不再进入另一个变量对象。换句话说，如果局部环境中存在着同名标识符，就不会使用位于父环境中的标识符，如下面的例子所示：

```
var color = "blue";

function getColor(){
    var color = "red";
    return color;
}

alert(getColor());  //"red"
```

ExecutionContextExample07.htm

修改后的代码在 getColor()函数中声明了一个名为 color 的局部变量。调用函数时，该变量就会被声明。而当函数中的第二行代码执行时，意味着必须找到并返回变量 color 的值。搜索过程首先从局部环境中开始，而且在这里发现了一个名为 color 的变量，其值为"red"。因为变量已经找到了，所以搜索即行停止，return 语句就使用这个局部变量，并为函数会返回"red"。也就是说，任何位于局部变量 color 的声明之后的代码，如果不使用 window.color 都无法访问全局 color 变量。如果有一个操作数是对象，而另一个不是，就会在对象上调用 valueOf()方法以取得基本类型的值，以便根据前面的规则进行比较。

变量查询也不是没有代价的。很明显，访问局部变量要比访问全局变量更快，因为不用向上搜索作用域链。JavaScript 引擎在优化标识符查询方面做得不错，因此这个差别在将来恐怕就可以忽略不计了。

4.3　垃圾收集

JavaScript 具有自动垃圾收集机制，也就是说，执行环境会负责管理代码执行过程中使用的内存。而在 C 和 C++之类的语言中，开发人员的一项基本任务就是手工跟踪内存的使用情况，这是造成许多问题的一个根源。在编写 JavaScript 程序时，开发人员不用再关心内存使用问题，所需内存的分配以及无用内存的回收完全实现了自动管理。这种垃圾收集机制的原理其实很简单：找出那些不再继续使用的变量，然后释放其占用的内存。为此，垃圾收集器会按照固定的时间间隔（或代码执行中预定的收集时间），周期性地执行这一操作。

下面我们来分析一下函数中局部变量的正常生命周期。局部变量只在函数执行的过程中存在。而在这个过程中，会为局部变量在栈（或堆）内存上分配相应的空间，以便存储它们的值。然后在函数中使用这些变量，直至函数执行结束。此时，局部变量就没有存在的必要了，因此可以释放它们的内存以供将来使用。在这种情况下，很容易判断变量是否还有存在的必要；但并非所有情况下都这么容易就能得出结论。垃圾收集器必须跟踪哪个变量有用哪个变量没用，对于不再有用的变量打上标记，以备将来收回其占用的内存。用于标识无用变量的策略可能会因实现而异，但具体到浏览器中的实现，则通常有两个策略。

4.3.1　标记清除

JavaScript 中最常用的垃圾收集方式是标记清除（mark-and-sweep）。当变量进入环境（例如，在函数中声明一个变量）时，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变

量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。

可以使用任何方式来标记变量。比如，可以通过翻转某个特殊的位来记录一个变量何时进入环境，或者使用一个“进入环境的”变量列表及一个“离开环境的”变量列表来跟踪哪个变量发生了变化。说到底，如何标记变量其实并不重要，关键在于采取什么策略。

垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾收集器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。

到 2008 年为止，IE、Firefox、Opera、Chrome 和 Safari 的 JavaScript 实现使用的都是标记清除式的垃圾收集策略（或类似的策略），只不过垃圾收集的时间间隔互有不同。

4.3.2 引用计数

另一种不太常见的垃圾收集策略叫做**引用计数**（reference counting）。引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，则这个值的引用次数就是 1。如果同一个值又被赋给另一个变量，则该值的引用次数加 1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减 1。当这个值的引用次数变成 0 时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

Netscape Navigator 3.0 是最早使用引用计数策略的浏览器，但很快它就遇到了一个严重的问题：循环引用。**循环引用**指的是对象 A 中包含一个指向对象 B 的指针，而对象 B 中也包含一个指向对象 A 的引用。请看下面这个例子：

```
function problem(){
    var objectA = new Object();
    var 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；也就是说，这两个对象的引用次数都是 2。在采用标记清除策略的实现中，由于函数执行之后，这两个对象都离开了作用域，因此这种相互引用不是个问题。但在采用引用计数策略的实现中，当函数执行完毕后，objectA 和 objectB 还将继续存在，因为它们的引用次数永远不会是 0。假如这个函数被重复多次调用，就会导致大量内存得不到回收。为此，Netscape 在 Navigator 4.0 中放弃了引用计数方式，转而采用标记清除来实现其垃圾收集机制。可是，引用计数导致的麻烦并未就此终结。

我们知道，IE 中有一部分对象并不是原生 JavaScript 对象。例如，其 BOM 和 DOM 中的对象就是使用 C++以 COM（Component Object Model，组件对象模型）对象的形式实现的，而 COM 对象的垃圾收集机制采用的就是引用计数策略。因此，即使 IE 的 JavaScript 引擎是使用标记清除策略来实现的，但 JavaScript 访问的 COM 对象依然是基于引用计数策略的。换句话说，只要在 IE 中涉及 COM 对象，就会存在循环引用的问题。下面这个简单的例子，展示了使用 COM 对象导致的循环引用问题：

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.element = element;
element.someObject = myObject;
```

这个例子在一个 DOM 元素（element）与一个原生 JavaScript 对象（myObject）之间创建了循环引用。其中，变量 myObject 有一个名为 element 的属性指向 element 对象；而变量 element 也有一个属性名叫 someObject 回指 myObject。由于存在这个循环引用，即使将例子中的 DOM 从页面中移除，它也永远不会被回收。

为了避免类似这样的循环引用问题，最好是在不使用它们的时候手工断开原生 JavaScript 对象与 DOM 元素之间的连接。例如，可以使用下面的代码消除前面例子创建的循环引用：

```
myObject.element = null;
element.someObject = null;
```

将变量设置为 null 意味着切断变量与它此前引用的值之间的连接。当垃圾收集器下次运行时，就会删除这些值并回收它们占用的内存。

为了解决上述问题，IE9 把 BOM 和 DOM 对象都转换成了真正的 JavaScript 对象。这样，就避免了两种垃圾收集算法并存导致的问题，也消除了常见的内存泄漏现象。

导致循环引用的情况不止这些，其他一些情况将在本书中陆续介绍。

4.3.3　性能问题

垃圾收集器是周期性运行的，而且如果为变量分配的内存数量很可观，那么回收工作量也是相当大的。在这种情况下，确定垃圾收集的时间间隔是一个非常重要的问题。说到垃圾收集器多长时间运行一次，不禁让人联想到 IE 因此而声名狼藉的性能问题。IE 的垃圾收集器是根据内存分配量运行的，具体一点说就是 256 个变量、4096 个对象（或数组）字面量和数组元素（slot）或者 64KB 的字符串。达到上述任何一个临界值，垃圾收集器就会运行。这种实现方式的问题在于，如果一个脚本中包含那么多变量，那么该脚本很可能会在其生命周期中一直保有那么多的变量。而这样一来，垃圾收集器就不得不频繁地运行。结果，由此引发的严重性能问题促使 IE7 重写了其垃圾收集例程。

随着 IE7 的发布，其 JavaScript 引擎的垃圾收集例程改变了工作方式：触发垃圾收集的变量分配、字面量和（或）数组元素的临界值被调整为动态修正。IE7 中的各项临界值在初始时与 IE6 相等。如果垃圾收集例程回收的内存分配量低于 15%，则变量、字面量和（或）数组元素的临界值就会加倍。如果例程回收了 85%的内存分配量，则将各种临界值重置回默认值。这一看似简单的调整，极大地提升了 IE 在运行包含大量 JavaScript 的页面时的性能。

事实上，在有的浏览器中可以触发垃圾收集过程，但我们不建议读者这样做。在 IE 中，调用 **window.CollectGarbage()** 方法会立即执行垃圾收集。在 Opera 7 及更高版本中，调用 **window.opera.collect()** 也会启动垃圾收集例程。

4.3.4 管理内存

使用具备垃圾收集机制的语言编写程序，开发人员一般不必操心内存管理的问题。但是，JavaScript 在进行内存管理及垃圾收集时面临的问题还是有点与众不同。其中最主要的一个问题，就是分配给 Web 浏览器的可用内存数量通常要比分配给桌面应用程序的少。这样做的目的主要是出于安全方面的考虑，目的是防止运行 JavaScript 的网页耗尽全部系统内存而导致系统崩溃。内存限制问题不仅会影响给变量分配内存，同时还会影响调用栈以及在一个线程中能够同时执行的语句数量。

因此，确保占用最少的内存可以让页面获得更好的性能。而优化内存占用的最佳方式，就是为执行中的代码只保存必要的数据。一旦数据不再有用，最好通过将其值设置为 null 来释放其引用——这个做法叫做解除引用（dereferencing）。这一做法适用于大多数全局变量和全局对象的属性。局部变量会在它们离开执行环境时自动被解除引用，如下面这个例子所示：


```
function createPerson(name){
    var localPerson = new Object();
    localPerson.name = name;
    return localPerson;
}

var globalPerson = createPerson("Nicholas");

// 手工解除 globalPerson 的引用

globalPerson = null;
```

在这个例子中，变量 globalPerson 取得了 createPerson()函数返回的值。在 createPerson()函数内部，我们创建了一个对象并将其赋给局部变量 localPerson，然后又为该对象添加了一个名为 name 的属性。最后，当调用这个函数时，localPerson 以函数值的形式返回并赋给全局变量 globalPerson。由于 localPerson 在 createPerson()函数执行完毕后就离开了其执行环境，因此无需我们显式地去为它解除引用。但是对于全局变量 globalPerson 而言，则需要我们在不使用它的时候手工为它解除引用，这也正是上面例子中最后一行代码的目的。

不过，解除一个值的引用并不意味着自动回收该值所占用的内存。解除引用的真正作用是让值脱离执行环境，以便垃圾收集器下次运行时将其回收。

4.4 小结

JavaScript 变量可以用来保存两种类型的值：基本类型值和引用类型值。基本类型的值源自以下 5 种基本数据类型：Undefined、Null、Boolean、Number 和 String。基本类型值和引用类型值具有以下特点：

- 基本类型值在内存中占据固定大小的空间，因此被保存在栈内存中；
- 从一个变量向另一个变量复制基本类型的值，会创建这个值的一个副本；
- 引用类型的值是对象，保存在堆内存中；
- 包含引用类型值的变量实际上包含的并不是对象本身，而是一个指向该对象的指针；
- 从一个变量向另一个变量复制引用类型的值，复制的其实是指针，因此两个变量最终都指向同一个对象；

- 确定一个值是哪种基本类型可以使用 typeof 操作符，而确定一个值是哪种引用类型可以使用 instanceof 操作符。

所有变量（包括基本类型和引用类型）都存在于一个执行环境（也称为作用域）当中，这个执行环境决定了变量的生命周期，以及哪一部分代码可以访问其中的变量。以下是关于执行环境的几点总结：

- 执行环境有全局执行环境（也称为全局环境）和函数执行环境之分；
- 每次进入一个新执行环境，都会创建一个用于搜索变量和函数的作用域链；
- 函数的局部环境不仅有权访问函数作用域中的变量，而且有权访问其包含（父）环境，乃至全局环境；
- 全局环境只能访问在全局环境中定义的变量和函数，而不能直接访问局部环境中的任何数据；
- 变量的执行环境有助于确定应该何时释放内存。

JavaScript 是一门具有自动垃圾收集机制的编程语言，开发人员不必关心内存分配和回收问题。可以对 JavaScript 的垃圾收集例程作如下总结。

- 离开作用域的值将被自动标记为可以回收，因此将在垃圾收集期间被删除。
- “标记清除”是目前主流的垃圾收集算法，这种算法的思想是给当前不使用的值加上标记，然后再回收其内存。
- 另一种垃圾收集算法是“引用计数”，这种算法的思想是跟踪记录所有值被引用的次数。JavaScript 引擎目前都不再使用这种算法；但在 IE 中访问非原生 JavaScript 对象（如 DOM 元素）时，这种算法仍然可能会导致问题。
- 当代码中存在循环引用现象时，“引用计数”算法就会导致问题。
- 解除变量的引用不仅有助于消除循环引用现象，而且对垃圾收集也有好处。为了确保有效地回收内存，应该及时解除不再使用的全局对象、全局对象属性以及循环引用变量的引用。