第 4 章　JavaScript 血液系统——数据、数据类型和变量

如今 Web 技术发展很快，有点乱花渐欲迷人眼，浅草都能没马蹄的感觉。身陷其中不免会拔剑四顾、心底茫然。搞开发，做精固然不易，但顾全实属无力。此时如果静下心来沉思繁荣下的底层技术，你会发现幸福的本源都是苦的，当然也可以反过来思考，痛苦的极限绝对是幸福的。

在编程世界中，变量曾经让很多初学者为之憔悴，但是在汇编语言中是没有变量这样的概念的。数据很抽象，数据类型更是让不少读者迷茫，但是在机器指令中是没有数据这样的概念的。计算机就是一个简单的计算工具，停息时，它是一堆廉价的废铁；工作时，它只识别开或关、敏感电流的强或弱。但是从计算机诞生之日起，它一直都是这样不知疲倦地工作着。计算机的本质就是计算，它没有新花样，无论是文字、图像、声音还是视频。所以我们可以概括说：

- 计算需要对象，于是就产生了数据。
- 计算需要效率，于是就产生了类型。
- 计算需要控制，于是就产生了变量。

4.1　从数据到类型

如果说，语法和逻辑结构是 JavaScript 语言的骨肉，那么数据可以让人联想到鲜活的血液。血流淌于整个 JavaScript 体系的每一个细胞，滋养着人的思想，孕育出新的设计和算法，同时也产生了一个个实际的应用项目，让人感到 JavaScript 语言的体温，而不是冷冰冰的代码。

4.1.1　数据的本质

提及数据，你会想到 1、2、3 等数字，也会想到 a、b、c 等字符，还有 true 和 false 等。没错，计算机程序正是通过操作这些具体可感的值来运行的。那么数据到底是什么呢？

最简单的问题可能也是最复杂的。当我们去思考这个简单问题时，必须从更深的层面去寻找答案。这正如数学领域里的公理证明，物理界的定律推论。计算机语言也必须从高级到低级，从汇编到机器指令，通过逆流而上，你才可能找到问题的本源。

程序设计的本质是使用命令来操纵数据的过程。这个命令体现为一段逻辑代码，逻辑体现的是一种算法或设计思想。因此，可以说指令或逻辑是无形的东西，而数据则是客观存在的。

数据与物质在本质上是同源的，物质存在于客观世界中，而数据存在于内存或磁盘中，物质是可感的，而数据却是可测的（如电流、磁介等）。

既然说数据是存在的，那么存在的都是有形的。如果说数据有形，你可能会惊讶。但是，如果说数据都是有类型的，你会释然。正如质量是物质存在的基础，那么类型体现了数据的形体。

说到类型，我们不妨来探究一下它的本质。物质的质量都是有单位的，如克、千克、斤、吨等。而类型正是数据的单位，通俗地说就是计算机指令一次可以操作的数据块大小。例如，双字节一次可以操作 32 位数据，单字节一次可以操作 16 位数据。如果没有各种数据类型，则计算机只有一个单位（即字节），那么当需要操作一个 4 字节大小的数据块时，就需要 4 次操作，而如果定义有双字节为单位的数据类型时，计算机只需要一次操作就可以完成任务，所以说数据类型提高了计算机的执行效率。

类型是客观存在的，它存在于硬件支持层面，而不是人的主观想象。如果你去反汇编程序的机器码，可以看到不同的数据类型，不同机器码的存在证明了类型是机器硬件级别支持的，不是通过软件实现的，更不是一个抽象的概念。我们常说的 32 位操作系统、64 位操作系统，实际上它描述了系统执行的数据存在的基本类型。数据类型的单位有大有小，小的单位可能只有 1 个字节（如布尔型数据），大的单位可能会非常大，这要看硬件是否支持。一般整型数据大小为 16 位 2 个字节、浮点型数据大小为 32 位 4 个字节等。实际上，对于布尔值来说，它只有 true 和 false 两个值，完全可以使用 1 位大小的数据类型来表示。但是，由于 CPU 在操作数据时是以字节为单位的，如果一个布尔值占据 1 个位，那么剩余的 7 个位就浪费了。

数据类型是数据的单位，而数据结构却是对数据进行的抽象。汇编语言没有、也不需要变量，这是因为开发人员可以手工分配内存，而高级语言必须定义变量，这是因为变量的内存分配是由编译器或运行时计算机自动完成的，因此可以在这个基础上发展接近人类抽象思维的数据结构，例如，面向对象就是对数据的一种抽象。

无论世事万物如何变化，能量守恒、物质依旧，计算的本质都是相同的，编程的核心也是静态的。程序设计是为了解决实际问题，而问题可以拆分成某些概念和逻辑关系。例如，结构化程序设计和面向对象程序设计就是对概念和逻辑进行描述的不同形式。在程序设计中，逻辑关系的复杂程度会随程序的规模而增加。所以说，如何对概念的抽象，如何对逻辑的描述，始终都是编程中最核心的内容。

4.1.2 柔弱的 JavaScript 语言

语言有强类型和弱类型之分。所谓弱类型语言，就是语言对于变量类型没有强制性要求。换句话说，弱类型语言就是允许在内存中某个固定区域里可以存放不同类型的数据。JavaScript 是一种弱类型语言，其柔弱性具体表现如下：

- 声明变量时，不用指定数据类型。当然这并不意味着 JavaScript 变量没有类型，而是它能够根据所赋值的类型来决定自己的类型，所以一个变量到底可以显示为什么类型，不取决于变量定义本身，取决于它所包含的数据。这一点显示了 JavaScript 的变量使用具有很强的灵活性，但是这种灵活性也制约了 JavaScript 语言做大做强。
- 当参与计算时，变量会根据具体的环境随时发生类型转换。当然这并不说明 JavaScript 语言在执行变量运算时无法无天，硬把数值当字符，或硬把字符当数值执行运算。只是它在计算之前显性或隐性转换变量的数据类型，以努力完成计算。

这说明 JavaScript 变量的用法比较灵活，所以你不用担心类型错误。例如，JavaScript 会自动调用 toString()方法把数值或布尔值转变为字符串，调用 parseFloat()或 parseInt()函数把字符串转变为数值，使用两个连续的非运算符（!）把字符串或数值转变为布尔值。

- 变量的数据类型比较简洁。JavaScript 语言只有 3 种最原始的数据类型：数值型、字符串型和布尔型。对于数值型，它不区分整数和浮点数，这在强类型语言中是难以想象的事情。复杂数据类型仅有对象和函数两种，对象是一类复合型数据，即数据集合，而数组只是对象的一种特殊形式，函数也是对象，但它只是一组可执行的代码块，而不是数据集合。此外，JavaScript 还定义了几个特殊的数据类型，如空类型（null）和未定义类型（undefined）。基本数据类型按值传送，而复杂数据类型按引用传送。

当然语言的类型强与弱仅从程度上来分析，语言的本质没有什么不同。例如，无类型的语言不用检查，甚至不区分指令和数据；弱类型语言对于数据类型的检查非常弱，仅能严格区分指令和数据；强类型语言能够严格区分不同类型数据，并在编译期进行检查。一般完全面向对象的语言都是强类型的，如 C#、Java 等。

弱类型的优点就是限制少，代码灵活，学习的门槛比较低。当然这也是以牺牲开发速度和效率为代价的，所以它成不了大气候，但是绝对好用，招人喜欢。例如，对于开发同样一个功能，如果使用 JavaScript 和强类型的 Java 来开发，两者对开发人员的要求、开发所花费的时间，以及代码执行效率都是明显不同的。在此就不再深入论述。

4.1.3　JavaScript 的基本数据类型

JavaScript 数据是非常简洁的，它只定义了 6 种基本数据类型（如表 4-1 所示）。

表 4-1　JavaScript 语言定义的 6 种基本数据类型

基本数据类型	说明
null	空、无。表示不存在，当为对象的属性赋值为 null，表示删除该属性
undefined	未定义。当声明变量却没有赋值时会显示该值。可以为变量赋值为 undefined
number	数值。最原始的数据类型，表达式计算的载体
string	字符串。最抽象的数据类型，信息传播的载体
boolean	布尔值。最机械的数据类型，逻辑运算的载体
object	对象。面向对象的基础

任何变量或值的基本类型都可以使用 typeof 运算符来获取，例如：

```
alert(typeof 1);                // 返回字符串"number"
alert(typeof "1");              // 返回字符串"string"
alert(typeof true);             // 返回字符串"boolean"
alert(typeof {});               // 返回字符串"object"
alert(typeof []);               // 返回字符串"object "
alert(typeof function(){});     // 返回字符串"function"
alert(typeof null);             // 返回字符串"object"
alert(typeof undefined);        // 返回字符串"undefined"
```

对于任何变量来说，typeof 运算符总是从字符串的形式返回上述 6 种类型之一。如果不考虑面向对象的编程，这 6 种基本类型已经够用了。但是，你也会发现：JavaScript 解释器认为 null 是属于 object 数据类型的一种特殊形式，而 function(){}是 function 类型，也就是说函数也是一种基本数据类型，而不是对象的一种特殊形式。

实际上，在 JavaScript 中，函数是一个极容易引起误解或引发歧义的数据类型，它可以是独立的函数类型，又可以作为对象的方法，也可以被称为类或构造器，还可以作为函数对象而存在等。

所以，在《JavaScript 权威指南》中把 function 被看做是 object 基本数据类型的一种特殊对象，另外《悟透 JavaScript》和《JavaScript 高级程序设计》也把函数视为对象，而不是一种基本数据类型。但是在《JavaScript 语言精髓与编程实践》中却把 function 视为一种基本数据类型，而把 null 视为 object 类型的一种特殊形式。至于谁对谁错，看来只有根据具体情况而定了。

4.1.4 数之源——值类型和引用类型

上述类型是从数据的形论进行归类的。我们还可以从用法的角度对其进行概括。任何语言、任何数据类型，它的数据都可以分为两大类：值类型和引用类型。

- 值类型：也称为原始数据或原始值（primitive value）。这类值存储在栈（stack）中，栈是内存中一种特殊的数据结构，也称为线性表，栈按照后进先出的原则存储数据，先进入的数据被压入栈底，最后插入（push）的数据放在栈顶，需要读取数据时从栈顶开始弹出（pop）数据，即最后一个数据被第一个读出来。因此说，值类型都是简单的数据段。变量的位置和变量值的位置是重叠的，也就是说值类型的数据被存储在变量被访问的位置。
- 引用类型：这类值存储在堆（heap）中，堆是内存中的动态区域，相当于自留空间，在程序运行期间会动态分配给代码和堆栈。堆中存储的一般都是对象，然后通过一个编号传递给栈内变量，这个编号就是所谓的引用指针（point），这样变量和变量值之间是分离的，它们通过指针相联系。当读写数据时，计算机通过变量的指针找到堆中的数据块，并进行操作。栈和堆数据的区别如表 4-2 所示。

表 4-2 栈和堆数据结构比较

	栈数据	堆数据
管理方式	由编译器管理	由程序员管理
空间大小	固定，一般比较小	不固定，一般比较大。在 Win32 中，堆可达 4GB，在 VC 中栈默认为 1MB，可以修改
垃圾问题	没有垃圾	容易产生数据碎片
生长方向	向下	向上，即向着内存增加的方向
分配方式	静态和动态分配	动态分配，没有静态分配
执行效率	由系统提供底层支持，有专门的寄存器存放栈地址，效率高	由库函数提供支持，效率底

在 JavaScript 中，number、string、boolean 和 undefined 型数据都是值类型。由于值类

型数据占据的空间都是固定的，所以可以把它们存储在狭窄的内存栈区。这种存储方式更方便计算机进行查找和操作，所以执行速度会非常快。

而对于 object 型数据（包括 function 和 array）来说，由于它们的大小是不固定的，所以不能存储在栈区，只能被分配到堆区，如果存储在栈区，则会降低计算机寻址的速度。而堆的空间是不固定的，所以很适合存储大小不固定的对象数据，然后在栈区存储对象在堆区的地址即可，而地址的大小是固定的，所以这种分离存储的方法不会影响计算机的寻址速度，对于变量的性能也没有任何负面影响（如图 4-1 所示）。

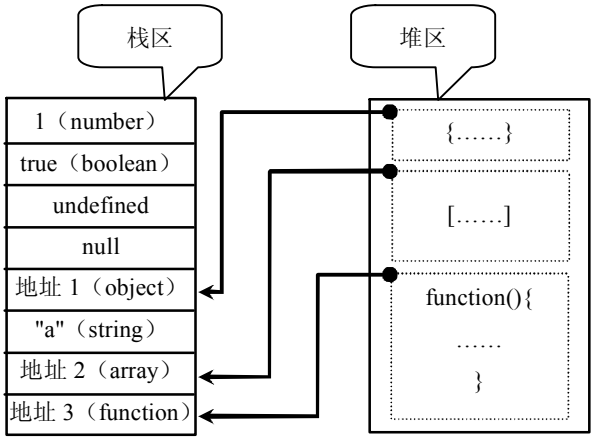

图 4-1　内存中的栈和堆模型示意图

在 JavaScript 语言中，object、function 和 array 等对象都是引用型数据。很多语言都把字符串视为引用型数据，而不是值类型，因为字符串的长度是可变的。但是 JavaScript 比较特殊，它把字符串作为值类型进行处理。不过，字符串在复制和传递运算中，是以引用型数据的方法来处理的。

4.1.5　原始值和引用值的操作本质

首先明确，值和类型是两个不同的概念。例如，null 是 null 类型的唯一值、undefined 是 undefined 类型的唯一值、而 true 和 false 是 boolean 类型仅有的两个值等。在任何语言中，值的操作都可以归纳为以下 3 个方面。

- 复制值：即把值赋值给新变量，或者通过变量把值赋值给另一个变量、属性或数组元素。
- 传递值：即把值作为参数传递给函数或方法。
- 比较值：即把值与另一个值进行比较，看是否相等。

由于值类型数据和引用型数据的值存在形式不同，自然操作它们的方法和所产生的结果也是不同的。注意，当值为值类型数据时，我们常称之为原始值或基本值；当值为引用型数据时，我们常称之为引用值或复合值。

1．使用原始值

对于原始值来说，其操作的 3 个层面说明如下。

1）复制值

在赋值语句中，操作的过程将会产生一个实际值的副本，副本的值和实际值之间没有任何联系，它们独自位于不同的栈区或者堆区。这个副本可以存储变量、对象的属性和数组的元素。例如：

```
var n = 123, a, b = [], c = {};
a = n;                  // 复制数字 123
b[0] = n;               // 复制数字 123
c.x = n;                // 复制数字 123
(a == b[0]) && (a == c.x) && (b[0] == c.x) && alert("复制的值都是相等的");
                        // 检测它们的值都是相等的
```

在上面示例中，分别把值 123 复制 3 份给变量 a、数组 b 和对象 c，虽然它们的值是相等的，但是它们之间是相互独立的。

2）传递值

当把值传递给函数或方法时，传递的值仅是副本，而不是值本身。例如，如果在函数中修改传递进来的值时，结果只能够影响这个参数值的副本，并不会影响到原来的值。

```
var a = 123;            // 原来的值
function f(x){
    x = x + x;
}
f(a);                   // 调用函数修改传递的值
alert(a);               // 查看变量 a 的值是否受影响，返回值为 123，说明没有变化
```

3）比较值

在上面的示例中我们也可以看到，当对原始值进行比较时，进行逐字节的比较来判断它们是否相等。比较的是值本身，而不是值所处的位置，固然比较的结果可能会相等，但只是说明它们所包含的字节信息是相同的。

2．使用引用值

对于引用值来说，其操作的 3 个层面说明如下。

1）复制值

在赋值语句中，所赋的值是对原值的引用，而不是原值副本，更不是原值本身。也就是说，进行赋值之后，变量保存的都是对原值的引用（即原值的存储地址）。当在多个变量、数组元素或对象属性中间复制时，它们都会与原始变量保存的引用相同。

所有引用具有相同的效力和功能，都可以执行操作，如果通过其中的一个引用编辑数据，这种修改将会在原值及其他相关引用中体现出来。例如：

```
var a = [1,2,3];        // 赋值数组引用
b = a;                  // 复制值
b[0] = 4;               // 修改变量 b 中第一个元素的值
alert(a[0]);            // 返回 4，显示变量 a 中第一个元素的值也被修改为 4
```

但是，如果给变量 b 重新赋予新值，则新值不会影响原值内容。例如：

```
var a = [1,2,3];        // 赋值数组引用
b = a;                  // 复制值
```

```
b = 4;                          // 为变量 b 重写赋值
alert(a[0]);                    // 变量 a 的内容保持不变
```

重复赋值实际上是覆盖变量对原值的引用，变为另一个值的副本或对其引用。所以不会对原值产生影响，演示示意图如图 4-2 所示。

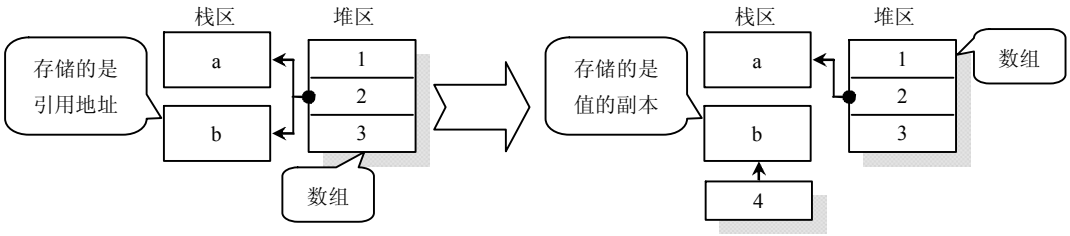

图 4-2　引用值的变化

2）传递值

当使用引用将数据传递给函数时，传递给函数的也是对原值的一个引用，函数可以使用这个引用来修改原值本身，任何修改在函数外部都是可见的。例如：

```
var a = [1,2,3];
function f(x){
    x[0] = 4;                   // 在函数中修改参数值
}
f(a);                           // 传递引用值
alert(a[0]);                    // 返回 4，原值发生变化
```

请注意，在函数内修改的是对外部对象或数组的引用，而不是对象或数组本身的值。在函数内可以使用引用来修改对象的属性或数组的元素，但是如果在函数内部使用一个新的引用覆盖原来的引用，那么在函数内部的修改就不会影响原引用的值，函数外部也是看不到的。读者可以根据如图 4-2 所示的演示清楚其中的道理。

3）比较值

当比较两个引用值时，比较的是两个引用地址，看它们引用的原值是否为同一个副本，而不是比较它们的原值字节是否相等。当对两个不同值进行引用时，尽管它们具有相同的字节构成，但是这两个引用的值却是不相等的。

```
var a = new Number(1);          // 引用值 a
var b = new Number(1);          // 引用值 b
var c = a;                      // 把 a 的引用赋值给 c
alert(a==b);                    // 返回 false
alert(a==c);                    // 返回 true
```

总之，对于任何语言来说，使用值和使用引用都是数据操作的两种基本方法。当我们操作数据时，要采用什么方法来进行处理，主要看数据的类型。值类型和引用型数据参与运算的方式不同，值类型数据通过使用值来操作数据，而引用型数据使用引用来操作数据。运算方式的不同，自然所产生的结果也不同。我们不妨再看一个示例：

```
var s = "abc";                  // 字符串，值类型数据
var o = new String(s);          // 字符串对象，被装箱后的字符串
function f(v){                  // 运算函数
    v.toString = function(){    // 修改参数的方法 toString()
```

```
            return 123;
        };
    }
    f(s);                           // 传入值
    alert(s);                   // 返回字符串"abc"，说明运算没有对原数据造成影响
    f(o);                       // 传入引用
    alert(o);                   // 返回数值 123，说明运算已经影响到原数据的内部结构
```

值类型是以实际值参与运算的，因此与原数据没有直接联系。而引用型以引用地址参与运算，计算的结果会影响到引用地址所关联的堆区数据块。但是，有一点例外，对于JavaScript 的字符串来说，它的操作方法就比较复杂，详细讲解请参阅第 10 章内容。

4.2 值类型数据

JavaScript 定义了 5 种基本数据类型（也称为原始数据类型或值类型）：null、undefined、boolean、string 和 number。每种类型都定义了它所包含值的范围，以及直接量的表示形式。同时，我们还可以通过 typeof 运算符来准确判断值的基本类型（读者可以参考上面示例代码）。

4.2.1 线性思维的符号——数值

数值（Number）也称为数字或数，它是所有编程语言中最原始、最基本的数据类型。有人把数值比做线性事物，所谓线性就是事物具有规律性，拥有大小、先后、增减等秩序，而其他类型的数据就没有这样的规律。程序员正是利用数值的这种规律来开发各种线性控制的代码段，如循环、迭代等。

不过与其他高级程序语言相比，JavaScript 似乎简化了数值类型的结构，不再去细分整型、浮点型等各种复杂的数值结构，所有数值都属于浮点型，这为 JavaScript 程序开发减少了很多麻烦。

1．数值直接量

当数值直接出现在程序中时，我们称之为数值直接量，通俗地说就是直接输入的任何数字都被看做数值直接量，当然通过变量存储或访问的数值就不是直接量了，它是变量的临时值，而不是直接量，很多时侯我们可以把看做是数值常量。

数值直接量可以细分为整型直接量和浮点型直接量。也许这样刻意细分的类型会让你很迷惑，实际上不管是整数还是浮点数，它们都是数值，在程序执行中是没有差异的，无非是精度问题不同罢了，但这在普通开发中好像没有多大实际价值。

也许你仍然好奇什么是整数或浮点数？通俗地说，浮点数就是带有小数点的数值，而整数是没有这样的精度要求。例如：

```
var int = 1;                        // 整型数值
var float = 1.0;                    // 浮点型数值
```

浮点数与整数最大的区别在于它们的位数不同，整数一般都是 32 位数值，而浮点数一般都是 64 位数值。更为有趣的是：浮点数是以字符串的形式进行存储的。

浮点数还可以使用科学计数法来表示。例如：

```
var float = 1.2e3;
```

等价于：

```
var float = 1.2*10*10*10;
```

或：

```
var float = 1200;
```

其中 e（或 E）表示底数，其值为 10，而 e 后面跟随的是 10 的指数。指数是一个整型数值，可以取正负值。例如：

```
var float = 1.2e-3;
```

等价于：

```
var float = 0.0012;
```

但不等于：

```
var float = 1.2*1/10*1/10*1/10;    // 返回 0.0012000000000000001
```

或：

```
var float = 1.2/10/10/10;          // 返回 0.0012000000000000001
```

最后，如果你感兴趣的话，可以了解整数和浮点数的精度范围，好像这在实际开发中很少去考虑。

- 整数精度：-2^{53}~2^{53}（-9007199254740992~9007199254740992），如果超出了这个范围，整数将会失去尾数的精确度。
- 浮点数精度：$\pm1.7976931348623157\times10^{308}$~$\pm5\times10^{-324}$，遵循 IEEE754 标准定义的 64 位浮点格式。

2．八进制和十六进制数值

上一小节就十进制的数值直接量进行了介绍，当然我们也可以把十进制数值转换为八进制和十六进制数值直接量，具体表示法如下。

- 十六进制数值直接量：以“0X”或“0x”作为前缀，后面跟随十六进制的数值直接量。例如：

```
var num = 0x1F4;                   // 十六进制数值
alert(num);                        // 返回 500
```

十六进制的数值是从 0~9，再从 a~f 的数字或字母任意组合，用来表示 0~15 之间的某个字，超过这个范围则以进制进行表示。

不管使用什么进制显示或处理数值，它们都是以浮点型格式进行存储的，不会改变数值的性质。在 JavaScript 中，我们可以使用 Number 对象的 toString(16)方法把十进制整数转换为十六进制字符串的形式显示。

- 八进制数值直接量：以数字 0 为前缀，其后跟随一个八进制的数值直接量。例如：

```
var num = 0764;                    // 八进制数值
alert(num);                        // 返回 500
```

尽管整数可以使用八进制或十六进制形式表示，但是所有数值在参与数学运算之后，返回的都是十进制数值，这一点通过上面的代码都能够看出来。

由于 ECMAScript 标准不支持八进制数值直接量，考虑到安全性，一般不建议大家使用八进制数值直接量。因为很多时候，JavaScript 会误解析为十进制数值。

3．数值运算

借助算术运算符，数值可以参与各种复杂的计算，这大概是程序运行的灵魂了，虽然你所看到的各种复杂逻辑或界面变化，但是计算机底层都是简单的加、减、乘、除等基本运算操作了。有关这些运算符的使用请参阅后面章节讲解。JavaScript 还提供了大量的数值运算函数，了解这些函数可以参阅本书附赠的 JavaScript 参考手册（Math 对象下）。这些函数都是静态方法，可以直接调用，例如：

```
var a = Math.floor(20.5);                    // 调用数学函数，下舍入
var b = Math.round(20.5);                    // 调用数学函数，四舍五入
alert(a);                                    // 返回 20
alert(b);                                    // 返回 21
```

当然，数值运算中最有用的方法应该是 toString()了，它可以根据所传递的参数把数值转换为对应进制的数值字符串。参数可以接受 2~36 之间的任意整数，也就是说，该方法可以把数值转换为 2~36 之间任意一种进制数值字符串。例如：

```
var a = 32;
document.writeln(a.toString(2));       // 返回字符串 100000
document.writeln(a.toString(4));       // 返回字符串 200
document.writeln(a.toString(16));      // 返回字符串 20
document.writeln(a.toString(30));      // 返回字符串 12
document.writeln(a.toString(32));      // 返回字符串 10
```

但是对于数值直接量来说，不能够直接调用 toString()方法，必须使用小括号强制运算数值直接量后，再调用该方法：

```
document.writeln(32.toString(16));     // 执行错误
document.writeln((32).toString(16));   // 返回 20
```

4．特殊数值

在 JavaScript 中，有几个怪异的数值是无法通过常规方法来表示的。可能你不可思议，但是它们确实存在（如表 4-3 所示），正如现实世界中是否真的存在外星人、大脚怪一样，信则有，不信则无。但是下面这几个值是必须相信且记住的。

表 4-3　特殊数值列表

特殊数值常量	说明
Infinity	无穷大。当数值超过浮点型所能够表示的范围。反之，负无穷大为-Infinity
NaN	非数值。不等于任何数值，包括自己。如当 0 除以 0 时会返回这个怪异的值
Number.MAX_VALUE	表示最大数值
Number.MIN_VALUE	表示最小数值，一个接近 0 的数值
Number.NaN	非数值，与 NaN 常量相同
Number.POSITIVE_INFINITY	表示正无穷大的数值
Number.NEGATIVE_INFINITY	表示负无穷大的数值

可能有读者会疑惑：为什么会定义两个无穷大的常量呢？原来，ECMAScript 1.0 版本定义了 Infinity 和 NaN 数值常量，但是 JavaScript 1.3 版本以前不支持它们，JavaScript 1.1 版本

开始支持 Number 对象包含的各种特殊数值常量，所以就有了上面部分功能相同的常量。

检测特殊数值常量的方法如下。

- 使用 isFinite()函数检测 NaN、正负无穷大常量。如果是有限数值，或者可以转换为有限数值，那么将返回 true。否则，如果只是 NaN、正负无穷大的数，则返回 false。
- 使用 isNaN()函数检测 NaN 常量。

```
document.writeln(isFinite(0));          // 返回 true
document.writeln(isNaN(0));             // 返回 false
document.writeln(isFinite(0/0));        // 返回 false
document.writeln(isNaN(0/0));           // 返回 true
document.writeln(isFinite("a"));        // 返回 false
document.writeln(isNaN("a"));           // 返回 true
```

NaN 是 Not a Number 短语的缩写。一般来说，当数据类型转换失败时会返回这个值，如把字母 a 转换为数值时会返回 NaN 常量。由于没有确切的值，NaN 是不会参与运算的。所以在检测一个值是否为 NaN 时，建议使用 isNaN()专有函数来实现。同时，NaN 不等于任何数值，包括自身，所以(NaN==NaN)就会返回 false。

4.2.2　形象思维的颜料——字符串

字符串（String）是计算机语言中另一种最基本、最原始的数据类型。但是与数值类型不同，它没有固定的大小。我们知道：整数大小一般都是 32 位，浮点数大小一般都是 64 位，而逻辑值只有 1 位，但是字符串是无法确定大小的，故有些语言把字符串视为一种对象（即拥有不固定的存储空间和结构），而不是简单的值类型。

如果说，数值反映了计算机的思维方式，即以线性逻辑来处理所有问题。那么字符串则反映了人类的思维模式，即以抽象无序的符号概括人类的情感和交流方式。把数值看做是计算机的交流语言，那么字符串应该是人类的交流媒介。因此说，数值和字符串构成了人机交互的基础，计算机依靠数值进行逻辑处理，然后通过字符串来理解人类的思维，同时人又通过字符串来间接了解计算机的数值处理的状态和结果。

1．字符串直接量

在 JavaScript 语言中，字符串，我们常常亲切地称之为文本（具有情感意义），是由 Unicode 字符、数字和各种符号组合而成的（在 JavaScript 1.3 版本以前仅支持 ASCII 字符集和 Latin-1 字符集），并包含在单引号或双引号之中。更专业地讲，它应该是字符串直接量（即字符串常量）。认识字符串时，大家还应该注意下面几个问题。

- 如果字符串包含在双引号中，则字符串内可以包含单引号。反之，就不行了，但是在 VBScript 中就可以在单引号中包含双引号。同样在 Java 语言中，字符串只能够由双引号包含来声明，而单引号包含的是字符，所谓字符就是单个字母，JavaScript 语言没有这样细分类型。
- 字符串应该在一行内显示，换行显示是不允许的。例如，下面字符串直接量的写法是错误的。

```
alert("字符串
直接量");                                    // 返回错误
```

当然如果编辑器无法在一行内显示而换行时则依然为一行内文本。如果换行显示字符串，可以在字符串中添加一个换行符（\n）。例如：

```
alert("字符串\n 直接量");                    // 在字符串中添加换行符
```

- 如果要在字符串中添加特殊字符，则需要使用转义字符进行转义，如单引号、双引号等。关于这个话题将在下面讲解。
- 字符串中每个字符都有特定的位置。首字符的位置为 0，第 2 个字符的位置为 1，以此类推。这与数组元素的位置是一样的，最后一个字符与数组中最后一个元素的位置一样都是字符串或数组长度减 1。

2．转义序列

转义序列，这个名词听起来很技术，通俗地说就是字符的一种间接表示方式。为什么要使用间接方式来表示字符呢？这是因为某些特殊字符包含其他含义或功能。例如，想在字符串中包含某个人说的话：

```
"子曰:"学而不思则罔，思而不学则殆。""
```

但是，由于 JavaScript 已经赋予了双引号为字符串直接量的声明符号，如果再在字符串中包含双引号，就会破坏字符串直接量。解决此类问题的方法就是使用一些特殊的字符来间接表示说话内容的双引号。

```
"子曰:\"学而不思则罔，思而不学则殆。\""
```

JavaScript 语言规定反斜杠加上字符就表示字符的本来意思或者某种特殊的含义（如表 4-4 所示），这样间接表示某些特殊字符的方法就被称为转义序列。

表 4-4　JavaScript 转义序列

序列	序列所代表的字符
\0	Null 字符（\u0000）
\b	退格符（\u0008）
\t	水平制表符（\u0009）
\n	换行符（\u000A）
\v	垂直制表符（\u000B）
\f	换页符（\u000C）
\r	回车符（\u000D）
\"	双引号（\u0022）
\'	单引号（\u0027）
\\	反斜线（\u005C）
\xXX	由两位十六进制数值 XX 指定的 Latin-1 字符
\uXXXX	由 4 位十六进制数值 XXXX 指定的 Unicode 字符
\XXX	由 1~3 位八进制数值指定的 Latin-1 字符。ECMAScript 3.0 版本不支持，一般不建议使用

由于反斜杠具有转义功能，但它仅对特殊字符有转义功能（如表 4-2 所示），因此当在一个正常字符前添加反斜杠时，则 JavaScript 会忽略该反斜杠。例如：

```
alert("子曰:\"学\而\不\思\则\罔\，\思\而\不\学\则\殆\。\"")
```

等价于：

```
alert("子曰:\"学而不思则罔，思而不学则殆。\"")
```

3．字符串操作

字符串操作在应用开发中是一个非常重要的环节，虽然字符串没有数组那样具有线性排列的结构和特性，但是借助 String 提供的众多属性和方法，我们也可以灵活操作字符串，从而实现复杂开发的目的。如果再配合正则表达式，那么字符串操作就会变得如鱼得水了，有关字符串操作的函数可以参阅 JavaScript 参考手册的 String 所定义的成员，而关于正则表达式相关知识请参阅后面章节讲解。下面介绍字符串的基本操作。

加号（+）运算符用于数值相加，同时在 JavaScript 中也可以用来连接两个字符串。例如，下面代码将返回“学而不思则罔思而不学则殆”合并后的字符串。

```
alert("学而不思则罔" + "思而不学则殆");
```

要确定字符串的长度（即字符串的字符数），可以使用字符串的 length 属性。例如，下面代码将返回 13。有关字符串的更多复杂操作请参阅后面相关字符串章节。

```
alert("学而不思则罔，思而不学则殆".length);     // 返回 13
```

4.2.3　逻辑思维的卵细胞——布尔型

布尔型（Boolean）是计算机语言中最简单的一种数据类型，仅包含两个固定的值（true 和 false），所占据的空间也是最小的（仅有 1 个字节）。其中常量 true 代表“真”，而 false 代表“假”。不过计算机正是通过逻辑的真与假来完成所有计算。因此，毫不夸张地说布尔型数据是计算机数据的鼻祖。

为了简化开发，JavaScript 还特别规定 undefined、null、""和 0 这 4 个特殊值转换为逻辑值时就是 false，除了这 4 个特殊值之外，其他任何类型的数据（包括值类型和引用类型）转换为逻辑值时都是 true。例如，下面使用 Boolean 构造函数强制转换这些特殊值的布尔型常量。

```
alert(Boolean(0));              // 返回 false
alert(Boolean(1));              // 返回 13
alert(Boolean(null));           // 返回 false
alert(Boolean(""));             // 返回 false
alert(Boolean(undefined));      // 返回 false
```

JavaScript 的这种规定对于我们在开发中检测一个变量或对象是否存在提供了方便。例如，下面代码判断变量 a 如果为空，则提示错误信息。

```
var a;
if(!a){
   alert("该变量为空，还没有赋值！");
}
```

或者，我们通过下面方式检测变量 b，并根据情况补加赋值：

```
var b;
b = b?b:"OK";                  // 如果变量 b 为空则重新为其赋值，否则采用原来的值
alert(b);
```

布尔型值在条件判断、循环计算等环节设计中非常实用，我们还将在后面的章节中不断渗透讲解布尔值的应用技巧。当然这需要结合逻辑运算符和各种方法实现综合开发。

4.2.4 空无道人——null

null 也许是最玄的一种数据类型了。盘古开天，万物皆空，世界本来是不存在的，这是唯心主义思想。但是在编程世界里，所有概念都是从空、无中诞生的。null 类型可以说是数据的鼻祖了，它不存在，当然要比存在的时间早；它没有东西、身无分文，但是它存在于意识中。是有还无、无中生有。鬼是不存在的，但是它有影子，活灵活现；数据不存在，但是它有概念，我们可以使用代码来表示，这个代码就是 null。

也许你会疑问：使用 typeof 运算符计算 null 值，返回的是 object，这是为什么呢？

实际上，这是 JavaScript 最初实现时犯的一个低级错误，至今还沿袭这个错误，但是从技术的角度分析，它任然属于值类型。我们可以把 null 视为对象的占位符，或者看做是对象的一个特殊值，它代表无对象的意思。既然说 null 是对象值，为什么又说它不是对象呢，这是一个想起来就让人头疼的问题，也许利用唯心论会可以很好地解释这个矛盾。

null 型数据只有一个值，即 null，或者说 null 是 null 型的直接量。当一个变量值为 null 时，说明它不是一个有效的对象，或者说是无对象。在对象中，当为属性设置值为 null 时，会提醒 JavaScript 回收它们，无用也即无存在的价值，所以赋值 null 就等于判变量死刑（即删除变量）。打一个不恰当的比方，你可以把 null 想象为数据生之前或死之前的存在符号。

4.2.5 人之初——undefined

undefined 是从 null 派生而来的，有史为证：

```
alert(null == undefined);          // 返回 true
```

两个值相等，这说明它们确实存在关联，不是空穴来风。null 和 undefined 都表示缺少值，所以这种相等也是可以理解的。但是，它们的含义不同，undefined 表示变量被声明但还没有初始化时被赋予的默认值，或者使用一个不存在的对象属性时，也会返回该值。undefined 犹如刚出生的孩子，通体稚嫩、全身乳气、世事不明。

当然，null 和 undefined 毕竟是两种不同类型的基本数据，我们可以使用全等运算符（===）或 typeof 运算符比较它们的异同：

```
alert(null === undefined);         // 返回 false
alert(typeof null);                // 返回"object"
alert(typeof undefined);           // 返回"undefined "
```

检测一个变量是否被初始化，也可以借助 undefined 值进行快速检测：

```
var a;                             // 声明变量
alert(a);                          // 返回变量默认值为 undefined
(a == undefined) && (a = 0);       // 检测变量是否初始化，否则为其赋值
alert(a);                          // 返回初始值 0
```

当然你还可以使用 typeof 运算符来检测变量的类型是否为 undefined：

```
(typeof a == "undefined") && (a = 0); // 检测变量是否初始化，否则为其赋值
```

注意

值 undefined 并不同于未定义的值。但是 typeof 运算符并能够区分它们的不同。例如，下面代码中声明了变量 a，而没有声明变量 b，但是使用 typeof 运算符对它们进行运算时，返回的值都是字符串"undefined"。

```
var a;
alert(typeof a);                    // 返回"undefined"
alert(typeof b);                    // 返回"undefined"
```

不过，对于未声明的变量 b 来说，如果使用其他运算符来对其进行计算的话，都会引发错误，因为其他运算符只能够用于已经声明的变量。例如，下面代码就会引发错误：

```
alert(b == undefined);              // 提示未定义的错误信息
```

对于函数、属性等对象来说，如果没有明确的返回值，则默认返回值都为 undefined。

```
function f(){
}
alert(f());                         // 返回"undefined"
```

与 null 还不同，undefined 不是 JavaScript 的保留字，在 ECMAScript v3 标准中才定义 undefined 为全局变量，初始值为 undefined。因此，在使用 undefined 值时，就存在一个兼容问题（早期浏览器可能会不支持 undefined）。例如，在 IE 5 及其以下版本中，除了直接赋值和 typeof 运算符外，其他任何运算符对 undefined 的操作都会引发异常。不过，我们可以声明 undefined 变量，然后查看它的值，如果为 undefined，则说明浏览器支持 undefined 值。

```
var undefined;
alert(undefined);                   // 返回"undefined"，说明浏览器支持
```

如果浏览器不支持 undefined 关键字，我们也可以自定义 undefined 变量，并赋值为 undefined。例如：

```
var undefined = void null;
```

声明变量 undefined，初始化为表达式 void null 的值，由于运算符 void 执行其后的表达式时，会忽略表达式的结果值，而总是返回值 undefined。利用这种方法可以定义一个变量 undefined，并赋值为 undefined。

既然是为变量 undefined 赋值为 undefined，则还可以使用如下方式：

```
var undefined = void 1;
```

或者使用没有返回值的函数：

```
var undefined = function(){}();
alert(undefined);                   // 返回"undefined"
```

在 IE 5 及其以下版本中，可以使用 typeof 运算符来检测某个变量的值是否为 undefined：

```
var a;
if(typeof a == "undefined"){
    // 执行代码
}
```

如果在 IE 5.5 及其以上版本浏览器中，undefined 已经被定义为全局变量，作为一个关键字被 JavaScript 支持。因此检测一个值是否为 undefined，可以直接进行比较：

```
var a;
if(a == undefined){
    // 执行代码
}
```

4.3 引用类型数据

在 JavaScript 中，引用型数据只有一种，即 object。但是它却可以细化出很多类型的数据结构，如 function、array 等。这些被细化的不同引用型数据被抽象化后就被称为类（class），类是不可以直接操作的。要操作某类数据，就必须调用 new 运算符，把它转换为实例对象。我们所处理的各种类型的对象，都是经过 new 运算符处理后的具体对象，而不是抽象的数据类型。

当然，JavaScript 语言没有真正意思上的类，但是定义对象实际上就是定义类，关于这个问题我们会在后面章节中进行深入探析。function、object 和 array 等都是最核心的引用型数据。另外，JavaScript 还预定义了很多比较常用且实用的引用型数据，如 Date、Math、ExgRep 等，即使是值类型的数据也可以被包装为引用型对象，如 String、Number、Boolean，这个问题可以参考下一节讲解。下面，我们将概述对象、函数和数组这 3 个基本引用型数据，更多更详细的内容都留待后面分章探索。

4.3.1 有序数据结构——数组

中国传统建筑中讲究对称，刻意方正，结构的严谨和守规蹈矩，目的就是为了牢固，坚不可摧，住在里面心理感觉踏实和安全，数组正体现了这种设计思想。

1．数组意象

数组（Array）是有序数据的集合，其结构的最大特征就是通过有序编号固定集合内每个数据的位置（如图 4-3 所示）。这个有序编号被称为下标，被固定的每个数据也被称为元素。数组内所包含元素的数据类型没有限制，可以是任意类型的数据，如数值、字符串、布尔型、对象、数组、函数等。下标值是一个从 0 开始的连续正整数。

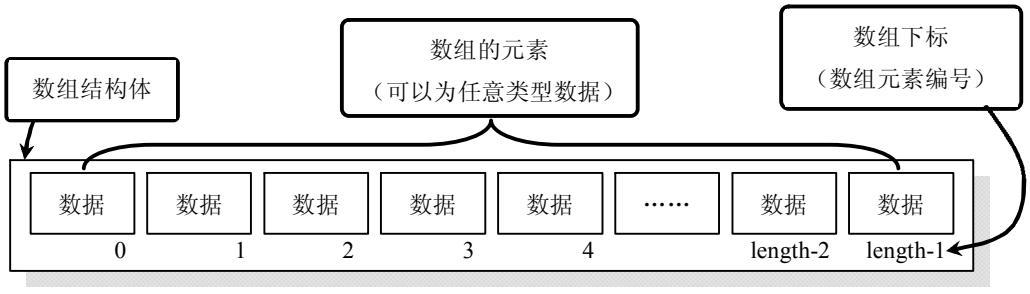

图 4-3　数据结构模型

获取数组元素值的方法是通过下标来定位的。例如：

```
var a = [[1,2],{x:1,y:2},function(){alert("我是数组元素")}]
```

```
alert(a[0]);          // 返回第一个元素的值，显示为数组结构
a[2]();               // 返回第三个元素的值，返回一个函数体，然后通过小括号执行运
                         算，弹出一个提示对话框
```

上述都是常规数组的结构特征，实际上在 JavaScript 中还有一种特殊的数组结构，即所谓的关联数组。关联数组是以字符串为下标来定位元素，关于这个技术话题请参阅第 9 章详细内容。

JavaScript 数组的结构比较固定，仅支持定义一维数组，虽然这简化了数组操作，但是扩展性比较差，你不能够使用 JavaScript 定义多维数组。这种固化的结构也显示了 JavaScript 语言的柔弱性。但是，JavaScript 对于数组元素所包含的数据类型没有限制，这在其他强类型语言也是很罕见的。不过这给开发人员留下了想象的空间，你可以通过为数组元素传递数组，从而模拟多维数组的结构。例如：

```
var a = [[11,12,13],[21,22,23]];        // 模拟二维数组结构
```

当然这仅是一种使用技巧，它没有改善数据的存储结构，所以，在性能上也是大打折扣。

2．第一次接触数组

定义数组类型的数据有多种方法，常用方法如下所示。

（1）通过构造函数 Array()创建数组。然后为数组中每个元素装入任意类型的数据。这犹如工厂中先设计好产品的模型，然后再向模型中浇灌塑料生成产品一样。

```
var a = new Array();                     // 使用构造函数构造数组结构体
a[0] = 0;                                // 为数组元素赋值
a[1] = "1";
a[2] = true;
```

当然，在构造数组时，可以直接在构造函数中传递值。例如，在下面这个新创建的数组中，第一个元素是数组类型数据，第二个元素是对象类型数据，第三个元素是函数类型的数据。

```
var a = new Array([1,2],{x:1,y:2},function(){alert("我是数组元素")})
```

我们也可以直接使用构造函数创建一个指定数组长度的空数组。例如，下面代码将创建一个包含 3 个未定义元素的新数组。

```
var a = new Array(3);
```

（2）通过数组直接量定义数组。所谓数组直接量就是就是通过中括号语法直接包含一组数据，或者也可以称之为数组常量。中括号内每个元素序列通过逗号语法分隔。例如，使用数组直接量定义上面的数组。当然，数组结构也是可以嵌套的，通过下面代码大家可以看到这一特点。

```
var a = [[1,2],{x:1,y:2},function(){alert("我是数组元素")}];
```

数组中不仅可以包含任意类型的数组，还可以包含任意形式的表达式。例如，在下面这行代码中，数组的第一个元素值为一个简单的算术表达式，第二个元素值为一个比较运算表达式，第三个元素值为一个条件表达式。

```
var a = [(3-2),(3<2),(true)?1:0];
```

当然，我们也可以使用数组直接量创建空数组，定义的方法是在逗号之间省去元素的值即可，此时元素的默认值为 undefined。例如，下面代码定义了包含 5 个元素的空数组。

```
var a = [,,,,];
```

与对象直接量一样，数组直接量也可以嵌套。例如，下面变量 a 是一个嵌套了 4 层的复杂结构数组。

```
var a = [1,[2,[3,[4,5,6]]]];
```

关于数组更高级操作与应用将在第 9 章中进行详细讲解。

4.3.2 离散数据结构——对象

离散是相对于连续的一个数学概念，离散结构体现了结构内部的数据不连续性。如果说 1 是属于 a 的，但不能够推论说 2 也是属于 a 的，因为 1 和 2 没有必然的联系，这就是离散的核心。对象的结构体现了离散性，它与数组结构的连续性截然不同。

1．对象意象

对象（Object）是无序数据的集合，看来它与数组一样都是复杂的数据容器，唯一的区别就是容器的结构不同罢了，自然装载数据的方式也就不同。如果说，数组是线性数据结构，即结构一般有一定的规律，适合进行统一的操作，如迭代、循环等批量操作。那么对象应该是离散数据结构，数据之间分散在结构体内，相互之间没有必然的联系。对象结构包含的数据没有位置概念，放在前面或后面都没有必然的联系（如图 4-4 所示），也不会影响对数据的存取操作。

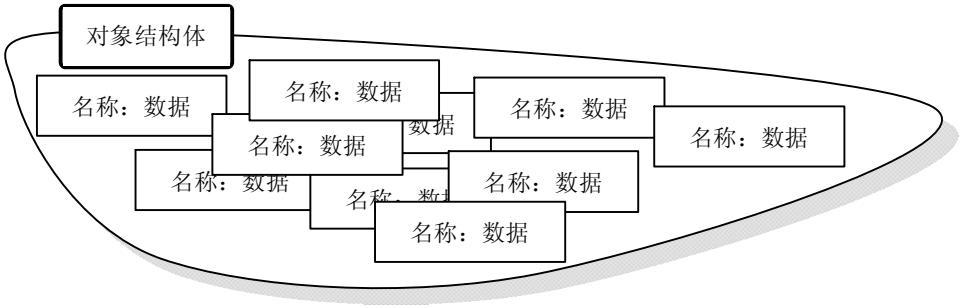

图 4-4　对象结构模型

对象内的数据具有数组的部分特点，多个数据成员之间通过逗号进行分隔，但是成员之间淡化了位置关系。对象很像一本字典。每个数据成员都被标识了一个名称，犹如字典的词条名，然后在其后包含了对词条的解释，这个解释就是对象包含的数据，故称为数据字典。

也有人把对象比做是已命名的数据集合，这些已命名的数据通常被称为对象的属性，属性实际上是一个私有变量，这个私有变量中包含的数据就是对象结构所包含的数据。

如果从另一个角度来看对象，对象的数据结构实际上就是一个名/值对的集合，故有人把它称为哈希表，不过这种列表结构却通过复杂的形式使 JavaScript 对象拥有强大的灵魂。

对象的结构就这么简单，但是在不同语言中却被演绎为不同的形象，如字典、数组、队列、列表、哈希表等，真是很有意思。不过，任你使用什么名词来描述它，对象结构绝对是数据世界里最灵活的精灵，找人喜爱，当然也让人烦恼。

2．第一次接触对象

定义对象结构有多种方法，常用方法如下所示。

（1）通过构造函数创建对象。例如：

```
var o = new Object();                    // 创建普通对象
var d = new Date();                      // 创建时间对象
var r = new RegExp();                    // 创建正则表达式对象
```

创建对象之后，就可以使用点号运算符为其定义属性，即为对象结构装入数据。例如：

```
var o = new Object();                    // 创建普通对象
o.a = "string";                          // 定义属性 a，值为字符串"string"
o.b = true;                              // 定义属性 b，值为布尔值 true
```

（2）通过对象直接量定义对象。即通过大括号语法来实现，大括号包含的是一个名/值对列表，名与值之间通过冒号隔开，而成员之间通过逗号分隔。例如：

```
var o = {                                // 对象直接量
    a : 1,                               // 定义属性
    b : true                             // 定义属性
}
```

对象的属性与 JavaScript 变量基本相似，但是变量名是标识符，而在对象直接量中属性名仅是一个字符串标签。所以，对于上面示例中定义的对象直接量，也可以这样来表示：

```
var o = {                                // 对象直接量
    "a" : 1,                             // 定义属性
    "b" : true                           // 定义属性
}
```

此时对象的属性名仅是一个字符串标签。但是在使用构造函数创建对象时，就不能够使用字符串标签来命名对象的属性名，因为此时属性已经变成了对象的成员，成员是合法的系统标识符。对象的属性可以是任意类型数据，如值类型数据、数组、对象、函数等。

- 如果属性值是函数，则该属性就成为对象的方法，读取这个特殊的属性值时，就必须附加小括号运算符。例如：

```
var o = {                                // 对象直接量
    a : function(){                      // 属性值为函数
        return 1;
    }
}
alert(o.a());                            // 附加小括号读取属性值，即调用方法
```

- 如果属性值是对象，则可以设计连续使用点号运算符引用内层对象的属性值。例如：

```
var o = {                                // 对象直接量
    a : {                                // 属性值为对象
        b:1
    }
}
alert(o.a.b);                            // 连续使用点号运算符读取内层对象的属性值
```

- 如果属性值是数组，则必须使用数组下标来读取某个元素的值。例如：

```
var o = {                                    // 对象直接量
    a : [1,2,3]                              // 属性值为数组
}
alert(o.a[0]);                               // 使用下标来读取属性包含的元素值
```

读取对象的属性值，可以调用点语法来实现，如上面示例所示。还可以使用关联数组来实现，即通过字符串下标来读取指定属性的值，例如：

```
var o = {                                    // 对象直接量
    a : 1
}
alert(o["a"]);                               // 使用关联数组来读取对象的属性值
```

更详细的讲解和示例请参阅第 8 章内容。

4.3.3 魔兽数据——函数

记得《北京人在纽约》一书中有句名言：“如果你爱他，就把他送到纽约，那里是天堂；如果你恨他，就把他送到纽约，那里是地狱”。函数大概也有点类似的味道，我很喜欢这句话，但不知道你喜欢吗？如果喜欢，就请你阅读下面的内容；如果不喜欢，那就更应该琢磨下面的文字。

1．函数意象

函数不是类型，更不是数据。这是大部分语言公认的法则。习惯于传统过程式编程，或者熟悉于现代面向对象编程的读者都应该知道：**函数仅是可执行的代码段，或者是对象的方法**。而所有这些都说明函数是语言的基本语法特性，但它不是数据，也不是类型。你可以定义函数，调用函数，却不能够把函数作为数值进行传递或赋值，甚至不能够把函数作为一个值参与到表达式运算中。

然而，JavaScript 却颠覆了开发人员对于函数的一贯认识。JavaScript 的函数具有其他语言中函数的各种特性，同时也拥有独特的个性。虽然 JavaScript 是基于对象的语言，但是 Object（对象）在 JavaScript 中不是第一型的，Function（函数）却是第一型的。所谓第一型，就是第一种类型，其他类型的对象都继承于 Function，Object 也是由函数来实现的。

2．大话函数演绎之路

对于初学者来说，要一下理清函数概念之间错综复杂的逻辑关系，确实是一件很费脑子的事情。下面我们不妨以一种轻松愉悦的心情大话函数的演绎历程（你可以联想和想象，但不要较真）。

函数的基本功能就是封装可执行的代码段，这对于任何语言来说都是一样的，正如人生下来自动会吃喝拉撒一样，JavaScript 语言也不例外。

```
function exec(){                    // 封装可执行代码的结构
    var sum = 0;
    for(var i = 0; i < 100; i ++ ){
        sum += i;
    }
    document.write(sum);
}
```

```
exec();                                    // 调用函数，实际上是执行一次封装的代码块
```

一次定义，多次调用。代码封装的目的就是这么简单。最初封装代码的方法不仅仅是函数的专利，如一些语言中的过程（sub），也具有代码封装的功能。

简单的封装不能够满足复杂的设计需要，于是就为封装代码挖个窗口，往里面塞东西，让它执行不同的动作，发展到后来就是所谓的参数，为了能够看到执行的结果，再强制封装代码执行后能够返回一个值。给封装的代码起一个好听的名字——函数，因为函数是数学领域的一种关系，描述两个集合之间的关系。传入数据，就应该传出结果，礼尚往来，函数懂得人之常情。

```
function exec(x){                          // 开了窗口的函数，实现人机交互
    return x*x;                            // 吃进去的是草，挤出来的是奶
}
exec(5);                                   // 不信你试一试，塞进去 5 元钱，返回的是 25
```

语言发展到现代阶段，有了面向对象程序设计。对象的方法是也是一个可执行的代码块，调用方法就会执行这个代码块。方法与函数本质上是相同的，实际应用中它们也是相等的。

```
document.write("谁把我写出来的？");
```

document 对象的方法 write()实际上就是一段可执行代码的封装，它能够把给定的字符串显示在浏览器中。

在函数式程序设计中，函数的功能得到了突破。函数不再是可执行的代码块，它可以作为运算元参与其他表达式的运算，此时把函数视为一个表达式，表达式的计算结果就是函数的返回值。如果没有返回值，则约定函数返回值为 undefined。这个约定为函数参与表达式运算奠定了基础，因为表达式都是有计算结果的。

```
(function(){
    // 晕死你，我就不给你返回值
}() == undefined ) &&
alert("哎呀，你咋啦，没有返回值")
```

上面代码虽然有多行，实际上它只是一个表达式。函数作为逻辑真运算的一个运算元，虽然没有返回值，但是它的返回值是 undefined，所以最终这个表达式执行计算之后，会弹出一个提示对话框，显示有趣的对话。

函数作为表达式的一部分参与运算是函数功能的一次质的飞跃。于是开发人员又尝试把函数作为一个值进行传递，从此函数就慢慢演化为一种结构奇特的数据，这种奇特的数据结构就是 function 数据类型。

```
var a = function(){                        // 把函数作为值赋值给变量 a
    return 1;
}
alert(a());                                // 计算变量 a，实际上就是调用匿名函数
```

上面这个没有名称的函数，被称为匿名函数，也被称为函数直接量。函数直接量的出现，正式宣布函数就是一种数据，一种令传统开发人员大跌眼镜的奇特结构的数据。

把函数作为数据，于是就可以尽情发挥自己的想象力和创造力，你可以把函数作为值赋值给对象的属性，于是对象的属性就变成了方法。例如，下面示例演示了把函数作为值传递给对象的属性，这个属性就变成了一个方法。

```
var o = {
    alert : function(x){                    // 把函数传递给对象属性
        alert("温馨提示：\n\n    " + x);
    }
}
o.alert("你吃饭了吗？");                    // 定义你的提示对话框
```

进一步的去想象，既然函数是一种复杂结构，是不是也可以在其中包含数据呢？当然可以：

```
function f(){                               // 简单的包含数据的函数
    var a =1;
    return a;
}
```

但是，函数结构体封闭严密，因为 JavaScript 把函数视为一个独立的作用域，即相当于一个与世隔绝的世外桃源，外界无法进入读取数据，此时我们只能够通过函数返回值实现读取函数内部的数据。当然，这也限制了函数与外界融合的机会。于是就有了构造函数：

```
function f(){                               // 构造函数
    this.a =1;
    this.b = function(){
        return this.a + this.a;
    };
}
```

构造函数通过关键字 this 来建立一个秘密通道，实现函数内包含的数据可以转运出去。然后再通过运算符 new 来具体找到这个数据通道，从而实现函数也可以是包含数据的结构：

```
var f1 = new f();
var a = f1.a;                               // 返回 1，即函数包含的数据
```

而这个密码通道又与对象访问自己的属性值是异曲同工的，说明函数所挖的秘密通道正是对象的数据结构。于是构造函数就成为定义对象的模板，即所谓的类。

```
alert(typeof f1);                           // 返回 object
```

3．探究奇特的函数解析机制

在众多语言中，JavaScript 的函数绝对是一道奇特的风景线。一会函数是可执行代码块，一会又是数据类型，背过脸却又成为替天行道的万物之类。为什么呢？

JavaScript 对函数的解析机制是不同的，也是多样的。对于使用 function 语句声明的函数，JavaScript 解释器会在预编译期就把函数处理了，而对于匿名函数却视而不见，直到执行期才按表达式逐行进行解释。解析时间的不同，必然导致函数拥有不同的特性。

我们不妨用一个示例验证一下。下面是使用 function 语句声明的两个同名函数 f，声明之后马上进行调用，代码如下：

```
function f(){                               // 声明函数 f
    return 1;
}
alert(f());                                 // 返回 2
function f(){                               // 声明函数 f
    return 2;
```

```
    }
    alert(f());                          // 返回 2
```

如果按代码从上到下的一般执行顺序，则第一次调用函数应该返回值为 1，第二次调用函数应该返回值为 2。但是，上面示例并不是这样。原来，JavaScript 解释器在预编译时就会把所有使用 function 语句声明的函数进行处理，如果发现同名函数，则后面的函数体会覆盖前面的函数体。所以，当在执行期时，就会看到两次调用函数 f 时，返回的值都是 2。

但是，如果我们把第一个函数改为匿名函数，继续试验，你会发现两次调用函数返回值都为 1。

```
    var f = function(){                  // 定义匿名函数 f
        return 1;
    }
    alert(f());                          // 返回 1
    function f(){                        // 声明函数 f
        return 2;
    }
    alert(f());                          // 返回 1
```

真是很有意思。首先，看 JavaScript 解释器在预编译期的处理，我们知道，在预编译期，使用 var 语句声明的变量和使用 function 语句声明的函数都会被处理。但是，JavaScript 解释器只是对 var 语句声明的变量名进行索引，变量的初始化值却被忽略，直到执行期时才去为变量读取初始值。而对于 function 语句创建的函数，JavaScript 解释器不仅对函数名按变量标识符进行索引，而且对于函数体也提前进行处理。于是，在预编译期，同名的变量被后来的同名函数所覆盖。但是，在执行期，第一行初始化变量 f 值为一个匿名函数，于是又覆盖了变量 f 在预编译建立的索引，即指向一个函数体。所以，两次调用函数最后都返回匿名函数的返回值 1。

如果我们把第二个函数改为匿名函数，则两次调用函数的返回结果又不相同。

```
    function f(){                        // 声明函数 f
        return 1;
    }
    alert(f());                          // 返回 1
    var f = function(){                  // 定义匿名函数 f
        return 2;
    }
    alert(f());                          // 返回 2
```

这次返回值的不同，与上面分析的原因都是相同的。因为在第一次调用函数 f 时，它指向的还是在预编译期索引的声明函数体，当第二次调用函数 f 时，该变量 f 已经被匿名函数所覆盖。

如果我们把两个函数都修改为匿名函数，则 JavaScript 在预编译期没有处理函数，仅是建立变量 f 的索引。当在执行期，才按顺序处理每一个匿名函数。

```
    var f = function(){                  // 定义匿名函数 f
        return 1;
    }
    alert(f());                          // 返回 1
    var f = function(){                  // 定义匿名函数 f
```

```
    return 2;
}
alert(f());                          // 返回 2
```

请注意，JavaScript 解释器在预编译期处理函数时，是按代码段分别执行的，也就是说每段 JavaScript 脚本是分隔开的，这样就可以避免在逻辑上出现混乱。所谓代码段，就是被<script>标签分隔的 JavaScript 脚本。例如，如果把两个被声明的同名函数放在不同的代码段中，则在预编译时，不会出现相互覆盖：

```
<script language="javascript" type="text/javascript">
// JavaScript 脚本段 1
function f(){                        // 声明函数 f
    return 1;
}
alert(f());                          // 返回 1
</script>
<script language="javascript" type="text/javascript">
// JavaScript 脚本段 2
function f(){                        // 声明函数 f
    return 2;
}
alert(f());                          // 返回 2
</script>
```

但是，同处于一个文档中的 JavaScript 脚本，即使它们分别位于不同的代码段中，但是它们都属于同一个作用域，即 window，因此相互之间是可以通信和调用的。例如：

```
<script language="javascript" type="text/javascript">
// JavaScript 脚本段 1
function f(){                        // 声明函数 f
    return 1;
}
alert(f());                          // 返回 1
</script>
<script language="javascript" type="text/javascript">
// JavaScript 脚本段 2
alert(f());                          // 返回 1
</script>
```

总之，在 JavaScript 脚本中，代码是以 function 为组织单元，可以说只有一种 function 形式，如在其他语言中，还有 sub、method 等代码组织结构。但是，对于 JavaScript 语言来说，function 是一个值，更是一种数据类型，一种最基本的数据类型，即函数类型。当我们使用 typeof 运算符来检测函数的类型时，返回的都是 function。

但是，当我们检测使用函数定义对象时，返回的数据类型却为 object。例如：

```
var f = function(){                  // 定义函数 f
}
alert(typeof f);                     // 返回类型为 function
var f = new f()                      // 定义函数 f 的对象
alert(typeof f);                     // 返回类型为 object
```

函数确实是一个很调皮的孩子，聪明可爱，但可爱得有时也让你郁闷，甚至让你暴跳如雷。

4.4　数据类型检测和转换

数据类型检测和转换是任何语言的基本能力，但是 JavaScript 语言具有更加灵活和强大的类型转换功能，更多的时候，你还没有发现错误，但是 JavaScript 解释器已经帮助你纠正了类型操作方面的错误。这确实是 JavaScript 更具亲和力的一种表现。

常用数据类型转换包括：把值转换为字符串、把字符串转换为数值，或者把值转换为布尔值。更复杂的类型转换就是装箱和拆箱，即值类型与引用型数据之间的相互转换。

4.4.1　数据类型检测

在程序设计中数据类型检测应该算为一项常规工作了，特别对于如 JavaScript 这样弱类型语言尤其显得重要。JavaScript 语言虽然对于数据类型的要求不是很苛刻，但是基本类型的检测还是必须的，读者应该知道 JavaScript 定义了 5 种原始数据类型：number、string、boolean、null、undefined，其他的都是复合数据类型 object。对于如何检测这些类型，JavaScript 提供了多种方法，下面我们将重点介绍两种。

1．使用 typeof 运算符

typeof 运算符专门用来测试值的类型，特别对于原始值比较有效，但是对于对象或数组类型数据，返回的值都是字符串"object"，显然这不是我们需要的。

```
alert( typeof 1);              // 返回字符串"number"
alert( typeof "a");            // 返回字符串"string"
alert( typeof true);           // 返回字符串"boolean"
alert( typeof {});             // 返回字符串"object"
alert( typeof []);             // 返回字符串"object"
alert( typeof function(){});   // 返回字符串"function"
alert( typeof undefined);      // 返回字符串"undefined"
alert( typeof null);           // 返回字符串"object"
alert( typeof NaN);            // 返回字符串"number"
```

由于 null 值返回类型为 object，所以我们可以定义一个检测值类型数据的一般方法：

```
function type(o){              // 返回值类型数据的类型字符串
   return (o === null) ? "null" : (typeof o);
      // 如果是 null 值，则返回字符串"null"，否则返回(typeof o)表达式的值
}
```

这样就可以避开因为 null 值影响基本数据的类型检测。至于如何检测对象和数组数据的类型，则请读者阅读后面的内容。

2．使用 constructor 属性

对于对象或数组，我们可以使用 JavaScript 对象自带的 constructor 属性，这个属性也称为构造函数属性，该属性值引用的是原来构造该对象的函数。例如：

```
var o = {};
var a = [];
alert(o.constructor == Object);    // 返回 true
alert(a.constructor == Array);     // 返回 true
```

通过上面方法，我们可以确定 typeof 运算符返回字符串为"object"的值是对象，还是数组。如果结合 typeof 运算符和 constructor 属性，我们基本能够完成数据类型的检测，如表 4-5 所示列举了不同类型数据的检测结果。测试代码：

```
var value = 1;                  // 输入不同类型的值（第一列）
alert(typeof value);            // 返回 typeof 运算符返回的字符串（第二列）
alert(value.constructor);       // 返回 constructor 属性返回的对象（第三列）
```

表 4-5　数据类型检测

值（value）	typeof value（表达式返回值）	value.constructor（构造函数的属性值）
var value = **1**	"number"	Number
var value = **"a"**	"string"	String
var value = **true**	"boolean"	Boolean
var value = **{}**	"object"	Object
var value = **new Object()**	"object"	Object
var value = **[]**	"object"	Array
var value = **new Array()**	"object"	Array
var value = **function(){}**	"function"	Function
function className(){}; var value = **new className();**	"object"	className

那么我们就可以利用这个参考表检测不同值的类型了。其实使用 constructor 属性可以判断绝大部分数据的类型。但是，对于 undefined 和 null 特殊值，就不能够使用 constructor 属性，JavaScript 解释器会抛出异常。这时，我们可以先把值转换为布尔值，如果为 true，则说明是存在值的，然后再调用 constructor 属性。例如：

```
var value = undefined;
alert(typeof value);                    // 返回字符串"undefined"
alert(value && value.constructor);      // 返回 undefined
var value = null;
alert(typeof value);                    // 返回字符串"object"
alert(value && value.constructor);      // 返回 null
```

请注意，对于数值直接量也不能够使用 constructor 属性，例如，下面代码将会提示语法错误：

```
alert(10.constructor);
```

但是如果加上一个小括号，则可以检测：

```
alert((10).constructor);
```

这是因为小括号运算符能够把数值转换为对象。

3．框架窗口的数组类型问题

constructor 属性是检测数据类型的最佳方法，但是有一点例外：那就是当在框架窗口中检测数组时很容易出现问题。先看一个示例（注意，下面示例在 IE 浏览器中无法正常运行）：

```
var iframe = document.createElement("iframe");// 创建一个浮动框架
document.body.appendChild(iframe);              // 在文档中插入该框架
var A = window.frames[0].Array;                 // 获取该浮动框架窗口包含
                                                   的 Array 构造函数
var a = new A();       // 利用该浮动框架的 Array 构造函数初建一个数组对象
alert(a.constructor == Array);              // 返回 false
alert(a.constructor);                       // 返回 Array 对象的字符串提示
```

通过上面示例可以看到，浮动窗口的 Array 构造函数与当前窗口的 Array 构造函数并不相等，虽然它们的 Array 类型结构相同，但是由于它们在内存堆区所存放的位置不同，所以结果也不相同。换句话说，使用 constructor 属性不能够很好地检测框架窗口中的数组类型。为此，我们必须另辟蹊径，下面是几种尝试。

（1）检测该数组中是否包含数组特有的方法或属性。

```
function isArray(o){
    return o!=null && typeof o==="object"&&"splice" in o &&"join"in o;
}
alert(isArray(a));                  // 返回 true
```

该方法是，判断值是否为空，如果不为空，则判断是否为 object 类型，然后探测该对象中是否包含数组特有的方法 splice()和 join()。如果找到这些方法，则说明该对象是数组类型。

（2）匹配 toString()方法返回的字符串。

使用第一种方法也容易造成误解，如果用户自定义了一个包含名称为 splice 和 join 的对象，则也会把它检测为数组类型。例如：

```
var o = {
    splice:1,
    join:2
}
alert(isArray(o));                  // 返回 true
```

但是如果把该对象转换为字符串，然后通过检测字符串中是否包含数组所特有的标志字符，也可以确定对象的类型。例如，对于数组对象来说，当直接使用 toString()方法时，将转换的字符串为数组元素的值。如果没有元素，则返回空字符串。

```
alert([].toString());               // 返回""
```

然而使用 call()或者 apply()方法调用 toString()方法时，返回的字符串就是"[object Array]"。所以我们可以这样设计：

```
function isArray(o) {
    return Object.prototype.toString.call(o) ==="[object Array]";
}
alert(isArray(a));                  // 返回 true
```


注意

在调用 toString()方法时，必须指定该方法具体的作用域路径，否则系统因为无法找到 toString()方法而报错。这样返回的字符串就可以包含“Array”标志字符了，然后通过字符串比较，就可以解决跨窗口判定对象是否为数组类型。

4．设计更安全的数据类型检测方法

受上面方法的启发，我们可以设计一种更安全的检测 JavaScript 基本数据类型和内置对象的方法，当然你还可以进一步补充该方法的检测数据类型范围。方法很简单：就是在数据类型的字符串标志数组中补充不同类型数据的字符串标志即可，如客户端 JavaScript 对象、自定义对象等不同类型数据的字符串标志。

首先，仔细分析不同类型对象的 toString()方法返回值，你会发现由 Object 对象定义的 toString()方法（注意，是默认方法）都比较有趣，即所有内置对象的 toString()方法返回的字符串形式总是：

```
[object class]
```

其中 object 表示对象的通用类型，class 表示对象的内部类型，内部类型的名称与该对象的构造函数名对应。例如，Array 对象的 class 为“Array”，Function 对象的 class 为“Function”，Date 对象的 class 为“Date”，内部 Math 对象的 class 为“Math”，所有 Error 对象（包括各种 Error 子类的实例）的 class 为“Error”。

客户端 JavaScript 的对象和由 JavaScript 实现定义的其他所有对象都具有预定义的特定 class 值，如“Window”、“Document'”和“Form”等。用户自定义对象的 class 为“Object”。

class 值提供的信息与对象的 constructor 属性值相似，但是 class 值是以字符串的形式提供这些信息的，而不是以构造函数的形式提供这些信息的，所以在特定的环境中是非常有用的。如果使用 typeof 运算符来检测，则所有对象的 class 值都为“Object”或“Function”。所以不能够提供有效信息。

但是，要获取对象的 class 值的唯一方法是必须调用 Object 定义的默认 toString()方法，因为不同对象都会预定义自己的 toString()方法，所以不能直接调用对象的 toString()方法。例如，下面对象的 toString()方法返回的就是当前 UTC 时间字符串，而不是字符串“[object Date]”。

```
var d = new Date();
alert(d.toString());                    // 返回当前 UTC 时间字符串
```

调用 Object 对象定义的默认 toString()方法，可以通过调用 Object.prototype.toString 对象的默认 toString()函数，再调用该函数的 apply()方法在想要检测的对象上执行即可。例如，结合上面的对象 d，具体实现代码如下：

```
var d = new Date();
var m = Object.prototype.toString;
alert(m.apply(d));                      // 返回字符串" [object Date] "
```

明白了上面的技术细节，下面就是一个比较完整的数据类型安全检测方法源代码：

```
// 安全检测 JavaScript 基本数据类型和内置对象
// 参数：o 表示检测的值
// 返回值：返回字符串"undefined"、"number"、"boolean"、"string"、"function"、
```

```
        "regexp"、"array"、"date"、"error"、"object"或"null"
    function typeOf(o){
        // 获取对象的 toString()方法引用
        var _toString = Object.prototype.toString;
        // 列举基本数据类型和内置对象类型，你还可以进一步补充该数组的检测数据类型范围
        var _type ={
            "undefined" : "undefined",
            "number" : "number",
            "boolean" : "boolean",
            "string" : "string",
            "[object Function]" : "function",
            "[object RegExp]" : "regexp",
            "[object Array]" : "array",
            "[object Date]" : "date",
            "[object Error]" : "error"
        }
        return _type[typeof o] || _type[_toString.call(o)] || (o ? "object" :
            "null");
        // 通过把值转换为字符串，然后匹配返回字符串中是否包含特定字符进行检测
    }
```

看个应用示例：

```
    var a = Math.abs;
    alert(typeOf(a));                    // 返回字符串"function"
```

上述方法适用于 JavaScript 基本数据类型和内置对象，但是对于自定义对象是无效的。因为自定义对象被转换为字符串后，返回的值是没有规律的，且不同浏览器的返回值也是不同的。因此，如果要检测非内置对象，只能够使用 constructor 属性和 instaceof 运算符来实现。

4.4.2　值类型数据的自动转换

JavaScript 是一种弱类型语言，在前面章节中也曾详细说明了它的柔弱性，例如，声明变量时无须指定数据类型，同时 JavaScript 能够自动转换变量的数据类型。

这种转换都是隐性的和自动的，不需要你去指挥，由 JavaScript 根据具体的语境自动完成。例如，当使用 alert()方法时，JavaScript 会自动把所有类型的值转换为字符串。如果在逻辑表达式中，则会自动把值转换为布尔值，转换的规则则事先已经定好，如空字符串和数字 0 为 false、实字符串和其他数字都被转换为 true 等。

JavaScript 在自动转换数据类型时，一般都会遵循这样一条基本原则：**如果某个类型的值被用于需要其他类型的值的环境中，JavaScript 就自动将这个值转换成所需要的类型。**

例如，如果在执行字符串连接操作时，则会把数字转换为字符串；如果在执行基本数学运算时，则会尝试把字符串转换为数值；如果在逻辑运算环境中，则会尝试把值转换为布尔值等。如表 4-6 所示，是常见值在不同环境中被自动转换的值列表。

表 4-6　数据类型自动转换列表

值（value）	字符串操作环境	数字运算环境	逻辑运算环境	对象操作环境
undefined	"undefined"	NaN	false	Error
null	"null"	0	false	Error
非空字符串	不转换	字符串对应的数字值 NaN	true	String
空字符串	不转换	0	false	String
0	"0"	不转换	false	Number
NaN	"NaN"	不转换	false	Number
Infinity	"Infinity"	不转换	true	Number
Number.POSITIVE_INFINITY	"Infinity"	不转换	true	Number
Number.NEGATIVE_INFINITY	"-Infinity"	不转换	true	Number
-Infinity	"-Infinity"	不转换	true	Number

（续表）

值（value）	字符串操作环境	数字运算环境	逻辑运算环境	对象操作环境
Number.MAX_VALUE	"1.7976931348623157e+ 308"	不转换	true	Number
Number.MIN_VALUE	"5e-324"	不转换	true	Number
其他所有数字	"数字的字符串值"	不转换	true	Number
true	"true"	1	不转换	Boolean
false	"false"	0	不转换	Boolean
对象	toString()	valueOf()或 toString()或 NaN	true	不转换

4.4.3　引用型数据的自动转换

表 4-6 中有关把对象转换为原始值时，只是简单地提供了转换的方法，但是如何转换，以及转换的结果都没有说明，下面结合几个很容易忽略的问题进行详细解释，其他问题比较简单就不再深入论述了。

1．对象在逻辑运算环境中的特殊情况

如果把非空对象用在逻辑运算环境中，则对象被转换为 true。这包括所有类型的对象，即使是值为 false 的包装对象也为 true。例如：

```
var a = new Boolean(false);
var b = new Number(0);
var c = new String("");
a && alert(a); // a 转换为布尔值为 true，但是提示它的字符串转换值为"false"
b && alert(b); // b 转换为布尔值为 true，但是提示它的字符串转换值为"0"
c && alert(c); // c 转换为布尔值为 true，但是提示它的字符串转换值为""
```

2．对象在数值运算环境中的特殊情况

如果把对象用在数值运算环境中，则对象会被自动转换为数字，如果转换失败，则返回值 NaN。具体转换过程如下。

首先，调用对象的 valueOf()方法，返回对象自身的值。大多数对象都继承了 Object 对象的默认方法 valueOf()。请注意，valueOf()方法仅能够返回自身值，但是不会转换值的类型，所以不要误认为 valueOf()方法取出的值都是数值，下面示例会演示这个问题。

如果对象自身值不为数值，因为方法 valueOf()不返回原始值。所以就会调用对象的 toString()方法，把对象自身值转换为字符串。然后调用 parseInt()或 parseFloat()函数把字符串转换成数字。

如果上述方法不能够成功，或者返回值为 NaN，则 JavaScript 会尝试通过强制方法把对象转换为数值。如果成功则已，不成功就返回 NaN。

从而实现把对象转换成数字，当然这个转换过程是在瞬间完成的，你自然没有感觉到。例如，看下面这个示例。

```
var a = new Boolean(true);          // 把 true 封装为对象
alert(a.valueOf());                 // 测试该对象的值为 true
alert(typeof (a.valueOf()));        // 测试值的类型为 boolean
a = a - 0;                          // 投放到数值运算环境中
alert(a);                           // 返回值为 1
alert(typeof a);                    // 再次测试它的类型，则为 number
```

如果我们用慢镜头分解 JavaScript 自动转换对象 a 到数字的过程，则转换过程如下。

首先，直接使用 valueOf()方法取值，没有成功。

```
a = a.valueOf();                    // 取出对象自身值
alert(a);                           // 返回 true
alert(typeof a);                    // 返回类型为 boolean
```

然后，把对象自身值转换为字符串，再次尝试转换，没有成功。

```
a = a.valueOf();                    // 取出对象自身值
a = a.toString();                   // 转换为字符串，返回"true"
a = parseFloat(a);                  // 转换为数值
alert(a);                           // 返回为 NaN
alert(typeof a);                    // 返回类型为 number
```

最后，尝试强制转换，则成功。

```
a = a.valueOf();                    // 取出对象自身值
a = Number(a);                      // 强制转换
alert(a);                           // 返回 1
alert(typeof a);                    // 返回类型为 number
```

3．数组在数值运算环境中的特殊情况

当数组被用在数值运算环境中时，数组将根据包含的元素来决定转换的值，具体说明如下。

- 如果为空数组，则被转换为数值 0。当数组为空时，JavaScript 将调用 toString()方法把数组转换为空字符串，然后再将空字符串强制转换为数值 0。

- 如果数组仅包含一个数字元素，则被转换为该数字的数值。例如：

```
var a = [5];
a = a * 1;                          // 投放到数值运算环境中
alert(a);                           // 返回数值 5
alert(typeof a);                    // 返回类型为 number
```

- 如果数组包含多个元素，或者仅包含一个非数字元素，则返回 NaN。例如：

```
var a = [true];
a = a * 1;
alert(a);                           // 返回数值 NaN
alert(typeof a);                    // 返回类型为 number
```

4．对象在模糊运算环境中的情况处理

当对象用于字符串环境中时，JavaScript 能够调用 toString()方法把对象转换为字符串再进行相关计算。而在数值运算环境中时，则会根据上面两小节介绍的方法进行转换操作。但是，在 JavaScript 中有两处运算环境比较模糊：加号运算符和比较运算符。当值进行加号运算或者比较运算时，即可以作用于数值，也可以作用于字符串。那么当对象用于这样的模糊环境中，该如何处理呢？

- 当对象与数值进行加号运算时，则会尝试把对象转换为数值，然后参与求和运算。如果不能够转换为有效数值，则执行字符串连接操作。例如：

```
var a = new String("a");            // 字符串封装为对象
var b = new Boolean(true);          // 布尔值封装为对象
a = a + 0;                          // 加号运算
b = b + 0;                          // 加号运算
alert(a);                           // 返回字符串"a0"
alert(b);                           // 返回数值 1
```

- 当对象与字符串进行加号运算时，则直接转换为字符串，进行连接操作。例如：

```
var a = new String(1);
var b = new Boolean(true);
a = a + "";
b = b + "";
alert(a);                           // 返回字符串"1"
alert(b);                           // 返回字符串"true"
```

- 当对象与数值进行比较运算时，则会尝试把对象转换为数值，然后参与求和运算。如果不能够转换为有效数值，则执行字符串比较运算。例如：

```
var a = new String("true");         // 无法转换为数值
var b = new Boolean(true);          // 可以转换为数值 1
a = a > 0 ;
b = b > 0 ;
alert(a);                           // 返回 false，以字符串形式进行比较
alert(b);                           // 返回 true，以数值形式进行比较
```

- 当对象与字符串进行比较运算时，则直接转换为字符串，进行比较操作。

不过，对于 Date 对象来说，加号运算符会先调用 toString()方法进行转换。因为当加号运算符作用于 Date 对象时，一般都是字符串连接操作。而当比较运算符作用于 Date 对象

时，则会转换为数字以便比较时间的先后。

请注意，在大多数情况下，JavaScript 会先尝试调用对象的 valueOf()方法对它进行转换。如果该方法返回了原始值（即数值或字符串），就使用那个值。如果 valueOf()方法返回值是未被转换的对象时，则 JavaScript 再调用对象的 toString()方法对它进行转换。

4.4.4　把值转换为字符串

把值转换为字符串是编程中常见行为。由于太常用，也太频繁，JavaScript 就自动帮办了这件事情，所以在你还未察觉错误使用数据类型之前，已经是雨过天晴。当然，自己动手进行转换也是很轻松的，具体方法如下。

1．使用加号运算符

当值与空字符串相加运算时，JavaScript 会自动把值转换为字符串。例如：

- 把数字转换为字符串：

```
var a = 123456;
a = a + "";
alert(typeof a);                    // 返回类型为 string
```

- 把布尔值转换为字符串，返回字符串"true"或"false"：

```
var a = true;
a = a + "";
alert(a);                 // 返回字符串"true"
```

- 把数组转换为字符串，返回数组元素列表，以逗号分隔：

```
var a = [1,2,3];
a = a + "";
alert(a);                 // 返回字符串"1,2,3"
```

- 把函数转换为字符串，返回函数结构的代码字符串：

```
var a = function(){
   return 1;
};
a = a + "";
alert(a);                 // 返回字符串"var a = function(){ return 1;}"
```

如果把 JavaScript 预定义对象（即默认的构造函数类）转换为字符串，则只返回构造函数的基本框架，而自定义的构造函数，则与普通函数一样，返回函数结构的代码字符串。

```
a = Date + "";
alert(a);                 // 返回字符串" function Date () { [ native code ]}"
```

其他 JavaScript 预定义对象转换为字符串类似，区别是函数名称字符串不同。

但是如果预定义对象为静态对象（即内置对象），则返回字符串又不同。例如：

```
a = Math + "";
alert(a);                 // 返回字符串"[object Math ]"
```


注意

Global 是一个空对象，不能够返回字符串。因此能够转换字符串的静态对象只有 Math。

如果把对象实例转换为字符串，则返回的字符串会根据不同子类型或定义对象的方法和参数而不同，具体说明如下。

- 对象直接量，则返回字符串为"[object object]"。

```
var a = {
   x :1
}
a = a + "";
alert(a);                    // 返回字符串"[object object]"
```

- 如果是自定义类的对象实例，则返回字符串为"[object object]"。

```
var a =new function(){}();
a = a + "";
alert(a);                    // 返回字符串"[object object]"
```

- 如果是预定义对象实例，具体返回字符串必须根据传递的参数而定，这个比较复杂。

例如，正则表达式对象会返回匹配模式字符串，时间对象会返回当前 GMT 格式的时间字符串，数值对象会返回传递的参数值字符串或者 0 等。

```
a = new RegExp(/^\w$/) + "";
alert(a);                                  // 返回字符串"/^\w$/"
```

2．圆滑的加号运算符

加号运算符有两个计算功能：数值求和、字符串连接。但是字符串连接操作的优先级要大于求和运算。因此，在可能的情况下，即运算元的数据类型不一致时，加号运算符会尝试把数值运算元转换为字符串，再执行连接操作。在下面小节中我们还会介绍这样的应用。

但是当多个加号运算符位于同一行时，这个问题就比较复杂，例如：

```
var a = 1 + 1 + "a";
var b = "a" + 1 + 1 ;
alert(a);                                  // 返回字符串"2a"
alert(b);                                  // 返回字符串"a11"
```

通过上面示例可以看到，加号运算符不仅仅优先于连接操作，同时还会考虑运算的顺序。对于变量 a 来说，按着从左到右的运算顺序，加号运算符会执行求和运算，然后再执行连接操作。但是对于变量 b 来说，由于"a" + 1 表达式运算将根据连接操作来执行，所以返回字符串"a1"，然后再用这个字符串与数值 1 进行运算，再次执行连接操作，最后返回字符串字符串"a11"，而不是字符串"a2"。如果要避免此类现象的发生，可以考虑使用小括号运算符来改变一行内表达式的运算顺序。例如：

```
var b = "a" + (1 + 1) ;                    // 返回字符串"a2"
```

3．使用 toString()方法

JavaScript 能够把值类型数据进行装箱，从而把值类型转换为对象，并且可以为它们定义属性或方法，而且它们还会从 Object 对象继承对象的基本方法和属性，而 toString()方法就是其中一个。关于装箱和拆箱操作请参阅 4.4.7 小节的内容。

当我们为具体的值调用 toString()方法时，JavaScript 会自动把它们装箱为对象。然后调用 toString()方法就可以把它们转换为字符串。例如：

```
var a = 123456;
a.toString();
alert(a);                                    // 返回字符串"123456"
```

其他类型数据转换为字符串的结果与使用加号运算符执行结果相同，这里就不再一一列举。请注意，使用加号运算符进行转换，实际上也是调用 toString()方法来完成。只不过是 JavaScript 自动调用 toString()方法实现的。

4．数字转换为字符串的模式问题

Number 对象的 toString()方法与其他对象的该方法都比较特殊，因为它扩展了对 toString()方法的支持，增加了一个整数参数，该参数可以设置数字的显示模式，默认为十进制显示方法，即所谓的默认模式，而当指定参数之后就称为基模式。

（1）如果采用默认模式，则 toString()方法会直接把数值转换为数字字符串。例如：

```
var a = 1.000;
var b = 0.0001;
var c = 1e-4;
alert(a.toString());                         // 返回字符串"1"
alert(b.toString());                         // 返回字符串"0.0001"
alert(c.toString());                         // 返回字符串"0.0001"
```

toString()方法能够根据相应的字符串输出数值，对于整数和浮点数来说直接输出，保留小数位。小数位末尾的零会被清除。但是对于科学计数法，则在条件许可的情况把它转换为浮点数，否则就使用科学计数法方式输出字符串。例如：

```
var a = 1e-14;
alert(a.toString());                         // 返回字符串"1e-14;"
```

在默认模式下，无论数值采用什么进制数值，toString()方法返回的都是数字的十进制表示。因此，对于八进制、二进制或十六进制数值，toString()方法都会先把它们转换为十进制数值之后再输出。例如：

```
var a = 010;                                 // 八进制数值 10
var b = 0x10;                                // 十六进制数值 10
alert(a.toString());                         // 返回字符串"8"
alert(b.toString());                         // 返回字符串"16"
```

（2）如果采用基模式，则 toString()方法会先把传递的数值转换为对应进制的值之后再输出。例如：

```
var a = 10;                                  // 十进制数值 10
alert(a.toString(2));                        // 返回二进制数字字符串"1010"
alert(a.toString(8));                        // 返回八进制数字字符串"12"
alert(a.toString(16));                       // 返回二进制数字字符串"a"
```

5．数字转换为字符串的位数问题

使用 toString()方法把数值转换为字符串时，无法保留小数位，或者指定是否采用科学计数法。这个缺陷对于表示货币格式、科学计数等特殊行业或领域是非常不方便的。为此，从 1.5 版本开始，JavaScript 新定义了 3 个新方法：toFixed()、toExponential()和 toPrecision()。

1）toFixed()

toFixed()能够把数值转换为字符串，并显示小数点后的指定位数。例如：

```
var a = 10;
alert(a.toFixed(2));              // 返回字符串"10.00"
alert(a.toFixed(4));              // 返回字符串"10.0000"
```

2）toExponential()

toFixed()方法不采用科学计数法，但是 toExponential()方法专门用来把数字转换为科学计数法形式的字符串。例如：

```
var a = 123456789;
alert(a.toExponential(2));        // 返回字符串"1.23e +8 "
alert(a.toExponential(4));        // 返回字符串"1.2346 e+8 "
```

toExponential()方法的参数指定了保留的小数位数。省略的部分采用四舍五入的方法进行处理。

3）toPrecision()

toPrecision()方法与 toExponential()方法不同，它是指定有效数字的位数，而不仅仅是指小数位数。例如：

```
var a = 123456789;
alert(a.toPrecision(2));          // 返回字符串"1.2e+ 8"
alert(a.toPrecision(4));          // 返回字符串"1.235 e+8"
```

4.4.5 把值转换为数字

JavaScript 提供了两种静态方法把非数字的原始值转换为数字：parseInt()和 parseFloat()。其中函数 parseInt()可以把值转换为整数，而函数 parseFloat()可以把值转换为浮点数。parseInt()和 parseFloat()函数对字符串类型的值有效，其他类型的值调用这两个函数都会返回 NaN。在转换字符串为数字之前，它们都会对字符串进行分析，以验证转换是否继续，具体分析如下。

1．使用 parseInt()函数

在开始转换时，parseInt()函数会先查看位置 0 处的字符，如果该位置不是有效数字，则将返回 NaN，不再深入分析。如果位置 0 处的字符是数字，则将查看位置 1 处的字符，并进行同样的测试，以此类推，在整个验证过程中，直到发现非数字字符为止，此时 parseInt()函数将把前面分析合法的数字字符转换为数值，并返回。例如：

```
alert(parseInt("123abc"));        // 返回数字 123
alert(parseInt("1.73"));          // 返回数字 1
alert(parseInt(".123"));          // 返回值 NaN
```

在浮点数中的点号对于 parseInt()函数来说是属于非法字符的，因此不会转换它，并返回。

如果以 0 为开头的数字字符串，则 parseInt()函数会把它作为八进制数字处理，先把它转换为数值，然后再转换为十进制的数字返回。如果以 0x 为开头的数字字符串，则 parseInt()函数会把它作为十六进制数字处理，先把它转换为数值，然后再转换为十进制的数字返回。

```
var d = "010";                          // 八进制数字字符串
var e = "0x10";                         // 十六进制数字字符串
alert(parseInt(d));                     // 返回十进制数字 8
alert(parseInt(e));                     // 返回十进制数字 16
```

parseInt()也支持基模式，可以把二进制、八进制、十六进制等不同进制的数字字符串转换为整数。基由 parseInt()函数的第二个参数指定。例如，下面是把十六进制数字字符串"123abc"转换为十进制整数：

```
var a = "123abc";
alert(parseInt(a,16));                  // 返回值十进制整数 1194684
```

再如，把二进制、八进制和十进制数字字符串转换为整数：

```
alert(parseInt("10",2));                // 把二进制数字 10 转换为十进制整数为 2
alert(parseInt("10",8));                // 把八进制数字 10 转换为十进制整数为 8
alert(parseInt("10" ,10));              // 把十进制数字 10 转换为十进制整数为 10
```

如果第一个参数是十进制的值，包含 0 前缀，为了避免被误解为八进制的数字，则应该指定第二个参数值为 10，即显式定义基，而不是采用默认基。例如：

```
alert(parseInt("010"));                 // 把八进制数字 10 转换为十进制整数为 8
alert(parseInt("010",8));               // 把八进制数字 010 转换为十进制整数为 8
alert(parseInt("010",10));              // 把十进制数字 010 转换为十进制整数为 10
```

2．使用 parseFloat()函数

parseFloat()函数与 parseInt()函数用法基本相同。但是它能够识别第一个出现的小数点号，而第二个小数点号被视为非法的。例如：

```
alert(parseFloat("1.234.5"));           // 返回数值 1.234
```

此外，数字必须是十进制形式的字符串，而不能够使用八进制或十六进制的数字字符串。同时对于数字前面的 0（八进制数字标识）会忽略，而对于十六进制形式的数字，则返回 0 值。例如：

```
alert(parseFloat("123"));               // 返回数值 124
alert(parseFloat("123abc"));            // 返回数值 123
alert(parseFloat("010"));               // 返回数值 10
alert(parseFloat("0x10"));              // 返回数值 0
alert(parseFloat("x10"));               // 返回数值 NaN
```

3．使用乘号运算符

加号运算符不仅能够执行数值求和运算，还可以把字符串连接起来。由于 JavaScript 处理字符串连接操作的优先级要高于数字求和运算。因此，当数字字符串与数值使用加号连接时，将优先执行连接操作，而不是求和运算。例如：

```
var a = 1;                              // 数值
var b = "1";                            // 数字字符串
alert(a+b);                             // 返回字符串"11"
```

在执行表达式 a+b 的运算时，变量 a 先被转换为字符串，然后以求和进行计算，所以计算结果为字符串"11"，而不是数值 2。因此，我们常常使用加号运算符把一个值转换为字符串。

不过，如果我们让变量 b 乘以 1，则加号运算符就以求和进行计算了。例如：

```
var a = 1;                          // 数值
var b = "1";                        // 数字字符串
alert(a + (b * 1));                 // 返回数值 2
```

也就是说，如果让一个数字字符串变量乘以 1，则 JavaScript 解释器能够自动把数字字符串转换为数值，然后再继续求和运算，而不是进行字符串连接操作。

4.4.6 把值转换为布尔值

在 JavaScript 中，任何数据都可以被自动转换为布尔值，这种转换往往都是自动完成的。例如，把值放入条件或循环结构的条件表达式中，或者参与到逻辑运算时，JavaScript 解释器都会自动把它们转换为布尔值。我们也可以手动进行转换，具体方法如下。

1．使用双重逻辑非运算符

任何一个值如果在前面加上一个逻辑非运算符，JavaScript 都会把这个表达式看做是逻辑运算。执行运算时，先把值转换为布尔值，然后再执行逻辑非运算。例如：

```
var a =!0;                          // 返回 true
var b =!1;                          // 返回 false
var c =!NaN;                        // 返回 true
var d =!null;                       // 返回 true
var e =!undefined;                  // 返回 true
var f =![];                         // 返回 false
var g =!{};                         // 返回 false
```

如果我们再给这个表达式添加一个逻辑非运算符，所得的布尔值就是该值被转换为布尔型数据的真实值了。例如：

```
var a =!!0;                         // 返回 false
var b =!!1;                         // 返回 true
var c =!!NaN;                       // 返回 false
var d =!!null;                      // 返回 false
var e =!!undefined;                 // 返回 false
var f =!![];                        // 返回 true
var g =!!{};                        // 返回 true
```

2．使用 Boolean()构造函数转换

使用 Boolean()构造函数转换的方法如下：

```
var a =0;
var b =1;
a = new Boolean(a);                 // 返回 false
b = new Boolean(b);                 // 返回 true
```

不过这种方法会改变布尔值的性质，此时它们都是引用型对象，而不再是原始值了。使用 typeof 运算符检测如下：

```
var a =0;
var b = !!a;
var c = new Boolean(a);
alert(typeof b);                    // 返回 boolean
```

```
alert(typeof c);                    // 返回 object
```

这就是所谓的数据类型的装箱和拆箱，关于这个复杂的技术话题请参阅下一节内容。注意，有关不同类型的值，以及各种特殊值转换为布尔值的规定，请参阅第 4.2 节内容。

4.4.7　装箱和拆箱

在面向对象编程中，一切数据都被视为对象，对象可以分为值类型和引用类型两种。其中值类型数据与引用类型数据可以相互转换，转换时需要进行装箱和拆箱操作。

1．从值到引用——装箱

所谓装箱（boxing）是指将值类型对象包装为对应的引用类型对象，这个包装对象就被称为值类型的包装对象。例如，数值对应的包装对象为 Number 对象、布尔型对应的包装对象为 Boolean 对象、字符串对应的包装对象为 String 对象、正则表达式直接量对应的包装对象为 RegExp 对象等。

在 JavaScript 对象系统中，包含有 String、Number、Function、Boolean 4 种基本对象构造器，它们是构造 JavaScript 对象系统的基础。例如，下面这个示例中，变量 a 和 b 的值都是 1，但是它们属于不同数据类型，其中 a 为数值，而 b 为对象。

```
var a = 1;                          // 直接赋值
var b = new Number(1);              // 通过 Number 构造函数装箱后赋值
alert(typeof a);                    // 返回 number，说明其值为值类型的数值
alert(typeof b);                    // 返回 object，说明其值为引用类型的对象
```

虽然变量 a 和 b 的类型不同，但是它们仍然属于 Number 构造器的实例。如果使用对象的 constructor 属性来检测，可以看到返回值相同，通过与 Number 对象比较，返回值都为 true，说明它们都是 Number 构造器的实例：

```
alert(a.constructor);       // 返回 function Number () { [ native code ]}
alert(b.constructor);       // 返回 function Number () { [ native code ]}
alert(a.constructor === Number);        // 返回 true
alert(b.constructor === Number);        // 返回 true
```

在表达式中，JavaScript 能够通过调用 valueOf()方法试图将引用类型的操作数拆箱为值类型，然后再进行运算。例如，在下面示例中，当加号运算符准备计算变量 a 和 b 的和时，先把引用类型的变量 b 借助 valueOf()方法转换为值类型，然后再进行计算，最后返回值为 2。

```
var a = 1;                          // 值类型
var b = new Number( 1 );            // 引用类型
alert( a + b );                     // 返回值为 2
alert( typeof (a + b) );            // 返回类型为 number
```

JavaScript 的 typeof 运算符能够返回 6 种数据类型：number、string、boolean、object、function 和 undefined。其中 object 和 function 为引用类型数据。

点号运算符一般要求左侧运算元为引用类型对象，因此当值类型变量进行此类操作时，JavaScript 将会自动把值类型的变量进行装箱操作。例如，为 Object 对象定义一个扩展方法 test()，该方法能够检测当前对象的数据类型是否为 Object 对象的实例，当前对象的构造器是否为 Number 或 String。

```
Object.prototype.test = function(){    // 扩展 Object 构造器的方法 f()
  alert(typeof this);                  // 显示当前对象的数据类型
  alert(this instanceof Object);       // 显示当前对象是否为对象的实例
  alert(this.constructor == Number);   // 显示当前对象的构造器是否为 Number
  alert(this.constructor == String);   // 显示当前对象的构造器是否为 String
}
```

如果定义一个数值变量，使用点运算符调用 test()方法：

```
var a = 1;                             // 数值变量
alert(typeof a);                       // 返回 number 数值类型
a.test();                              // 调用检测方法
```

可以看到，变量 a 的类型为引用型对象，而不再是值类型的数值。该对象为 Object 对象的实例，是 Number 对象的实例，但不是 String 对象的实例。

如果定义一个字符串变量，使用点运算符调用 test()方法：

```
var b = "string"                       // 字符串变量
alert(typeof b);                       // 返回 string 字符串类型
b.test();                              // 调用检测方法
```

可以看到，变量 b 的类型为引用型对象，而不再是值类型的字符串。该对象为 Object 对象的实例，是 String 对象的实例，但不是 Number 对象的实例。

2．从引用到值——拆箱

所谓拆箱（unboxing）是指将引用类型对象转换为对应的值类型对象，它是通过引用类型的 valueOf()方法来实现的。JavaScript 抽象了对象类型的拆箱方法，调用任何对象的 valueOf()方法都会返回这个对象的值。对于自定义的对象，程序员可以自定义它的 valueOf()方法，并把它理解为对这个类型的对象的拆箱。在 JavaScript 中实现把引用类型的对象拆箱为数值对象，还可以使用 toString()等方法来实现。例如：

```
var a = new Number(1);
var b =new String("1");
alert(typeof a);               // 返回 object 类型
alert(typeof b);               // 返回 object 类型
alert(typeof a.valueOf());     // 返回 number 数值类型
alert(typeof b.valueOf());     // 返回 string 字符串类型
alert(typeof a.toString());    // 返回 string 字符串类型
alert(typeof b.toString());    // 返回 string 字符串类型
```

利用装箱和拆箱功能，可以通过允许值类型的任何值与引用类型的值相互转换，将值类型与引用类型有机联系起来。

4.4.8 数据类型的强制转换

使用强制类型转换可以访问特定的值，即使值的类型已经变成了另一种类型。JavaScript 定义了 3 种强制类型转换。

- Boolean(value)：把参数值转换为 boolean 型。
- Number(value)：把参数值转换为 number 型。

- String(value)：把参数值转换为 string 型。

当调用这 3 个函数时，将会创建一个新值，并用这个新值存放原始值直接转换成的值。例如：

```
var a = String(true);                    // 返回字符串"true"
var a = String(0);                       // 返回字符串"0"
var b = Number("1");                     // 返回数值 1
var c = Number(true);                    // 返回数值 1
var d = Number("a");                     // 返回 NaN
var e = Boolean(1);                      // 返回 true
var f = Boolean("");                     // 返回 false
```

使用强制方式转换数据类型，有时候会造成意想不到的后果。例如，在上面示例中，true 被强制转换为数值 1。同理 Number(false)会转换为 0。而当使用 parseInt()方法转换时，它们都返回 NaN。

```
var a = parseInt(true);                  // 返回 NaN
var b = parseInt(false);                 // 返回 NaN
```

当要转换的值是至少有一个字符的字符串、非 0 数字或对象时，Boolean()函数将返回 true。如果该值是空字符串、数字 0、undefined 或 null，则它将返回 false。

Number()函数的强制类型转换与 parseInt()和 parseFloat()方法的处理方式相似，只是它转换的是整个值，而不是部分值。例如：

```
var a = Number("123abc");                // 返回 NaN
var b = parseInt("123abc");              // 返回数值 123
```

String()函数与 toString()方法功能基本相同。但是 String()函数能够把 null 或 undefined 值强制转换为字符串，而不引发错误。例如：

```
var a = String(null);                    // 返回字符串"null"
var b = String(undefined);               // 返回字符串"undefined"
```

但是下面用法都将导致异常：

```
var a = null.toString();
var b = undefined.toString();
```

在 JavaScript 中，使用强制类型转换有时候会非常有用，但是应该确保转换值的正确。

4.5　使用变量

前面曾讲过，**变量与值是两个不同的概念**。实际上，变量就是内存中的一块存储空间，这个空间中存放的数据就是变量的值。为这块区域贴个标识符，就是变量名。程序能够根据标识符索引找到变量所在的位置，然后再存取该空间内的数据。你可以把变量想象为一个盒子，盒子存储或包含的数据就是变量的值。

有了变量，你就可以在程序中存储和运算数据了。表面上看，程序操作的是变量，而实际操作的内容应为变量包含的值。当然，程序又不能缺少变量，没有变量，数据计算就变为静态行为了。代码操作的数据永远不变，这与单片机上的嵌入程序没有什么两样。发出指令，就执行规定的动作，而不是根据参数执行灵活的操作。

4.5.1 声明变量

所谓变量声明，就是定义变量（也有人认为，定义变量就是为变量赋值，而不是声明变量），再通俗一点说，声明变量就是在 JavaScript 中注册一个名字（即标识符），JavaScript 解释器在执行代码之前，会先对这些已声明的变量名进行索引，以建立一个索引表，这样当在程序执行中检索变量时就非常快，以此提高程序的执行效率。人们也把这个过程称为 JavaScript 的预编译处理。

在变量使用过程中，有一个非常容易模糊的概念：未定义变量。什么是未定义变量？不同的人可能理解不同：一种理解是没有声明的变量，读取这种未定义的变量，会引发语法错误。另一种理解就是变量被声明，但是没有被赋值，读取这种未定义的变量，返回的值总是 undefined。在本书中，我们将把未定义变量视为未声明的变量。

1．变量声明的方法

在 JavaScript 中，使用变量之前必须先声明。好像不声明也可以使用，那只不过是 JavaScript 解释器自动帮助你补办了这个手续，所以就给人一种幻觉：不声明照样可以使用变量。不声明就使用变量，这个本没有错，但却给了偷懒者犯懒的机会，不过这种行为犹如饭前不洗手一样，弊肯定大于利。

声明变量很简单，使用 var 语句声明即可：

```
var a;                          // 声明一个变量
var a, b, c;                    // 声明多个变量
```

当声明多个变量时，应使用逗号运算符进行分隔变量名。我们不仅可以一次声明多个变量，而且还可以在声明中为变量赋值。其实，即使你不给变量指定一个初始值，JavaScript 也会自动为声明的变量初始化为 undefined（未定义）值。例如：

```
var a;                          // 声明但没有赋值
var b = 1;                      // 声明并赋值
alert(a);                       // 返回 undefined
alert(b);                       // 返回 1
```

一旦使用 var 语句声明了变量，该变量会永久存储在 JavaScript 变量索引表中，它不会被回收，也不允许被删除（使用 delete 运算符）。

2．3 种变量命名法

变量名应该遵循 JavaScript 标识符命名规则，关于这个话题请参与第 3 章讲解。当然，只是因为变量名合法，并不意味着就应该随意使用。在程序设计中，开发人员经过多年的实践和经验积累形成了很多套比较成熟的命名方法，其中比较经典的变量命名法有 3 种。

1）匈牙利命名法

这种命名方法是由微软公司的一位程序员查尔斯•西蒙尼提出来的，匈牙利命名法被广泛应用于 Microsoft Windows 编程环境中。它通过在变量名前面加上相应的小写字母的符号标识作为前缀，标识出变量的作用域、类型等，前缀后面是一个或多个单词组合，单词描述了变量的用途。例如，i 表示整数，s 表示字符串，命名示例如下：

```
var sUserName = "css8", iCount = 0;
```

下表列举了匈牙利命名法定义 JavaScript 变量的前缀字符与数据类型对照表（如表 4-7

所示）。

表 4-7　匈牙利命名法前缀与变量类型

类型	前缀	示例
整数	i	iValue
浮点数	f	fValue
布尔型	b	bFound
字符串	s	sValue
数组	a	aValues
对象	o	oType
函数	fn	fnMethod
正则表达式	re	rePattern
泛型	v	vValue

2）骆驼式（Camel）命名法

正如它的名称一样，骆驼式命令法是混合使用大小写字母来构成变量的名称。例如，下面分别用骆驼式命名法和下画线命名法定义同一个函数。

```
function printLoadTemplates(){}
function print_load_templates(){}
```

第一个函数名使用了骆驼式命名法，这种命名方法规定每一个单词首字母应使用大写字母来标记，而名称的首字母要小写，这与匈牙利命名法的名称首字母类似，第二个函数名使用了下画线法，函数名中的每一个单词都用一个下画线来标记。

骆驼式命名法近年来很流行，在很多新的语言和编程环境中，它应用得比较多。另一方面，下画线命名法是 C 语言出现后开始流行起来的，在许多旧的程序和 UNIX 这样的环境中，它的使用非常普遍。

3）帕斯卡（Pascal）命名法

帕斯卡命名法与骆驼式命名法类似，只不过骆驼式命名法是第 1 个单词首字母小写，而帕斯卡命名法是第 1 个单词首字母要大写，例如，MyFunction 就是一个帕斯卡命名的示例，而 myFunction 是一个骆驼命名法。在 C#中，帕斯卡命名法和骆驼命名法使用比较多。

由于 JavaScript 程序规模比较小，对于命名没有强类型语言要求那么强烈。同时，如果读者分析 Google 网站的 JavaScript 程序，变量命名非常简洁，一般都为 1 或 2 个字母组成，这样做能够提高开发速度，便于书写，更为重要的是简洁的命名方法可以节省带宽。

4.5.2　丑陋的 JavaScript 变量

很多熟悉强类型语言的用户当初次使用 JavaScript 变量时，会非常不习惯，这种不习惯不是因为它很难，相反是因为它太简单、太随意了，反倒让人不习惯。正如城里人下乡，可以随意在地上吐痰、擤鼻涕很是不习惯（比喻有点不恰当，如今乡下也很文明了）。说到 JavaScript 变量的种种陋习，下面我们列举一二，当然这个陋习不是因为你造成的，而是 JavaScript 语言的不严谨性和对用户过于溺爱造成的。

1．陋习一：变量没有类型

JavaScript 是一种弱类型语言，它与其他强类型语言（如 C#或 Java）相比，最大的不同就是对于变量的要求很宽容，即不用指定变量类型。这样变量就可以包含任意类型的数据。

你可能疑惑了：变量不是有类型吗？例如：

```
var a = 1;
alert(typeof a);                    // 类型为 number
```

其实这个类型是指变量包含数据的类型，变量（内存地址）自身没有类型。此时，如果给变量再赋予其他类型的值，则变量的类型又会发生变化。

而在其他强类型语言中，这种做法简直是不可思议的，因为它们认为，任何一个变量在声明之前，都必须先指定类型，声明类型之后就不能够再进行修改，且变量只能够存储与自己类型相一致的数据。

JavaScript 变量不仅能够根据所包含的值显示不同的类型，同时还会自动转换变量包含数据的类型，以适应不同操作环境。

2．陋习二：变量可以重复声明

在 JavaScript 中，如果不怕累，你可以反复声明同一个变量。JavaScript 能够容忍你的这种非礼，且没有怨言的一次次地帮助你进行预处理。当然处理的结果是：只有最后一次声明的变量名是有效的，其他声明的变量都因为重名被覆盖了。

即使是重复声明并初始化变量的值，JavaScript 也仅把它看做是赋值命令的连续操作，会一丝不苟地一遍遍执行。例如：

```
var a = 1;
var a = 2;
var a = 3;
alert(a);                           // 返回 3
```

如果在其他强类型语言中出现这种行为，早就抛出异常了，也许会让你暴跳如雷。JavaScript 真的是太好说话了。

3．陋习三：变量可以隐式声明

上面曾提过，当户不声明变量而直接为变量赋值时，JavaScript 不会提示语法错误，这是因为 JavaScript 解释器能帮助你隐式声明变量。但是隐式声明的变量总是作为全局变量而存在的，JavaScript 解释器不会帮助你隐式声明局部变量。

所以，这就容易引发问题：当在函数中不声明就直接为变量赋值时，JavaScript 会把它视为全局变量进行处理。由于是全局变量，函数外代码可以访问该变量的值，这等于间接泄露了函数的隐私。例如：

```
function f(){
    a = 1;                          // 未声明直接赋值
    var b = 2;                      // 声明并赋值
}
f();                                // 调用函数，实现变量初始化
alert(a);                           // 返回 1
alert(b);                           // 提示语法错误，找不到该变量
```

不过，如果尝试读取一个未声明的变量的值，JavaScript 会提示语法错误。注意，为变

量赋值的过程，实际上 JavaScript 也会隐式进行声明。总之，使用变量应该记住以下 3 条。

- 养成良好习惯：**先声明，后读写；先赋值，后运算**。
- 避免不良习惯：**不声明，先赋值，隐声明；只声明，不赋值，undefined**。
- 戒掉错误习惯：**不声明，未赋值，尝试读，异常现**。

4．陋习四：变量的灰色潜规则

JavaScript 语言没有旗帜鲜明的规定：变量必须先声明后使用，而是慈善地认为，不声明就直接为变量赋值仅是一种疏忽，原谅吧，于是 JavaScript 就任劳任怨地为用户隐式声明了变量。看似很善良，但是这也容易让人滋生一种错误的意识。先看一个示例：

```
if(!a) {                          // 如果变量 a 不存在
   a = 0;                         // 则为变量 a 赋值为 0
}
alert(a);
```

上面示例设计的本意是：如果变量 a 不存在，就为其赋值为 0。也许这样设计愿望很美好，但是变量 a 在没有声明的前提下就被直接读取使用，根据上节讲解可知，JavaScript 是不允许的，将会抛出语法错误。

但是，试验发现，如果变量 a 未定义，在 Firefox 中会提示错误。而在 IE 里不会报错，且也不会执行后面所有代码，就这样僵持着，这种装傻行为其实很要命，不执行也不告诉你为什么，对于初学者来说，如果不明白内幕，也不知道从何纠错，就非常郁闷了。

不过可以考虑使用属性法来读取。因为当读取一个未声明的属性时，JavaScript 不会报错（很奇怪的潜规则）。而变量 a 是全局变量，作为全局变量，它应该是 window 对象的一个属性，所以可以这样来设计：

```
if(!window.a) {                   // 如果 window 对象的属性 a 不存在
   a = 0;                         // 则为变量 a 赋值为 0
}
alert(a);
```

但是上述方法只适用全局变量，如果是局部变量，就只能够通过类型检测法来判断了：

```
(function f(){
   if(typeof a == "undefined"){   // 如果变量 a 的类型为不可知
      a = 0;                      // 则为变量 a 赋值为 0
   }
   alert(a);
})()
```

总之，读者应该养成良好的习惯，对于变量使用，应该先声明，后读写，不要在变量未声明之前使用它。

4.5.3　不可跨越的门槛——变量的作用域

所谓变量的作用域（scope），就是指变量在程序中可供访问的有效范围，也称为变量的声明周期，即变量存活于哪些代码行中。在 JavaScript 中，变量作用域可以分为全局作用域和局部作用域。

- 全局变量的作用域是全局性的（global），即在整个 JavaScript 程序中到处都可以访问全局变量。
- 局部变量的作用域是局部的（local），仅能够在声明的函数内部可以访问。局部变量也只能在函数内部起作用。注意，函数的参数也是局部性的，只在函数内部起作用。

1．变量优先级问题

在函数体内，局部变量的优先级要比同名的全局变量高。此时，局部变量会覆盖同名全局变量。例如：

```
var a = 1;                          // 全局变量
(function f(){
   var a = 2;                       // 局部变量
   alert(a);                        // 返回 2
})();                               // 直接在函数体上调用函数
```

如果在函数体内存在同名参数变量和全局变量，则参数变量的优先级要比同名全局变量高。例如：

```
var a = 1;                          // 全局变量
(function f(a){                     // 参数变量
   alert(a);                        // 返回 3
})(3);                              // 直接调用函数，并传递参数值为 3
```

如果在函数体内存在同名参数变量和局部变量，则局部变量的优先级要比同名参数变量高。例如：

```
var a = 1;                          // 全局变量
(function f(a){                     // 参数变量
   var a = 2;                       // 局部变量
   alert(a);                        // 返回 2
})(3);                              // 直接调用函数，并传递参数值为 3
```

2．局部作用域的嵌套

在 JavaScript 中，函数可以嵌套，但是由于每个函数都有自己的局部作用域，所以也就形成了多个局部作用域嵌套的现象。

```
var a = 1;                          // 全局变量
(function(){
   var a = 2;                       // 第 1 层局部变量
   (function(){
      var a = 3;                    // 第 2 层局部变量
      (function(){
         var a = 4;                 // 第 3 层局部变量
         alert(a);                  // 返回 4
      })()                          // 直接调用函数
   })()                             // 直接调用函数
})()                                // 直接调用函数
```

这时你可以看到，内层的局部作用域要比外层局部作用域的变量优先级高。所以，在上面示例中，alert(a);语句最终显示的是 4，而不是其他局部变量值。

同时，内层函数可以访问外层函数的局部变量，而外层函数却不能够访问内层函数的变量。这就是所谓的变量作用域链（请参阅 4.5.5 节讲解）。

3．利用函数作用域实现技术封装

在 JavaScript 中，只有函数体结构才拥有独立封闭的域，外界是不允许访问函数内部的变量和参数等数据。而在其他复合类型的数据结构中，都不能提供这种封闭严密的作用域空间。因此，一般开发人员都利用函数结构来封装 JavaScript 技术框架，避免框架内的变量与其他框架或全局变量相互影响。如果读者仔细分析当前流行的框架，就会发现都是这样设计的。例如，jQuery 框架的基本结构如下：

```
(function(){                                    // 定义封装的独立作用域
   var jQuery = window.jQuery = window.$ = function( selector, context ){
      return new jQuery.fn.init( selector, context );
   };
   jQuery.fn = jQuery.prototype ={
      // 详细代码
   }
   jQuery.fn.init.prototype = jQuery.fn;
   jQuery.extend = jQuery.fn.extend = function(){
      // 详细代码
   }
})()                                            // 直接调用函数
```

通过使用函数局部作用域的封装，这样就可以有效避免了在同一个文档中多个技术框架或其他 JavaScript 代码之间的相互影响。这样，如果用户在文档全局域中又定义了同名变量 jQuery 或$，不至于把 jQuery 技术框架给覆盖了。

```
(function(){                                    // 定义封装的独立作用域
   var jQuery=window.jQuery=window.$=function(selector, context ){ };
})()                                            // 直接调用函数
var jQuery = 1;
var $ = 2;
```

虽然在全局作用域中使用变量，可以不用 var 语句，但是如果在声明局部变量时，一定要确保使用 var 语句。前面我们也曾经讲解了这个问题。例如，下面示例演示了如果不显式声明局部变量所带来的后果：

```
var jQuery = 1;
(function(){
   jQuery = window.jQuery = window.$ = function(){};
})()
alert(jQuery);                                  // 结果读取了函数内部封装的代码
```

对于函数来说，它并不知道全局作用域中已经定义了哪些变量，也无法预知这些全局变量的功能。因此，在函数体内使用全局变量是一种很危险的行为，很可能函数就会改变程序中其他部分的使用值。为了避免此类问题的发生，读者应该养成在函数体内使用局部变量，声明变量时应该使用 var 语句。

4．变量的解析过程

关于变量的解析问题，我们曾经在前面的章节中略有介绍。就是说，JavaScript 在预编

译期会先处理声明的变量和函数。但是，还有一个细节需要读者注意，那就是变量的赋值操作发生在 JavaScript 执行期，而不是预编译期。我们先看下面这个示例：

```
function f(){
    a = 1;                              // 全局变量 a 赋值为 1
    var b = 2;                          // 局部变量 b 赋值为 2
}
try{
    alert(a);                           // 尝试读取全局变量 a
}
catch(e){
    alert(e.message);                   // 显示错误信息：变量 a 未定义
}
f();                                    // 调用函数
alert(a);                               // 读取全局变量 a，返回值为 1
```

通过上面示例，可以看出，在函数未调用之前，函数内部定义的全局变量是无效的，这是为什么呢？原来，在 JavaScript 预编译期，仅仅是对函数的名称、函数内部的各种标识符进行检索，并建立索引。实际上，对于全局变量的预处理也是如此。

而只有当在 JavaScript 执行期时，才按顺序为变量进行赋值，并初始化。而在执行期，如果函数未被调用，则函数内代码是不被解析的，所以才有了上面你看到的示例演示效果。

根据 JavaScript 解析过程，我们再看下面这个奇怪的示例：

```
var a = 1;                              // 声明并初始化全局变量
(function f(){
    alert(a);                           // 返回 undefined
    var a = 2;                          // 声明并初始化局部变量
    alert(a);                           // 返回 2
})()
```

上面代码显示，由于在函数内部声明了一个同名局部变量 a，所以在预编译期，JavaScript 就使用该变量覆盖掉全局变量对于函数内部的影响。而在执行初期，局部变量 a 未赋值，所以在函数第 1 行代码中读取局部变量 a 的值也就是 undefined 了。当执行到函数第 2 行代码时，则为局部变量赋值 2，所以在第 3 行中就显示为 2。

5．被 JavaScript 忽略的块级作用域

在强类型语言中，作用域的种类还是很多的，除了全局作用域、函数作用域，还有块级作用域，但是 JavaScript 没有块级作用域。

所谓块级作用域，就是在独立的逻辑结构中，其内部的变量也具有独立性和封闭性，如条件结构、循环结构等。例如，在下面示例中，你会看到只要在函数体内，不管变量位于哪个语句块，都可以相互进行访问，这说明 JavaScript 是不支持块级作用域的。

```
(function f(){                          // 函数结构体
    for(var i =0 ; i <20; i++){         // for 语句块
        document.write(i+"<br />");
        if(i == 19) {                   // if 语句块
            var j = "for";
        }
    }
    alert(j);
```

```
})()
```

虽然说，JavaScript 没有块级作用域，但是下面示例警告了我们，在声明变量时，还是要小心，避免在条件或循环结构中声明变量。

```
<script language="javascript" type="text/javascript">
window.a = 1;
</script>
<script language="javascript" type="text/javascript">
if(false){
    var a = 2;
}
alert(a);
</script>
```

上面示例如果在 IE 中预览，则 alert(a);语句会返回 undefined 值，而在 Firefox 等符合 DOM 标准的浏览器中显示为 1。出现这种不兼容性问题，主要还是因为块级作用域的影响，虽然 JavaScript 不支持块级作用域。也许这仅是 IE 浏览器的一个 Bug，如果我们稍微改动一下代码的组成形式，就不会出现这种异常了。

6．函数闭包的作用域

函数闭包是一种特殊的函数结构，简单说就是在函数内部包含一个函数，被包含的函数引用函数外部的变量，从而形成一个闭包体，有关闭包技术的详细讲解请参阅第 20 章相关内容。

首先，请看下面这个示例。读者可以思考一下，当我们调用闭包函数时，然后访问变量 a 和 b，那么它将读取的是哪个域上的值呢？

```
var a = 1;                          // 全局变量
var b = 2;                          // 全局变量
function f(){
    var a = 3;                      // 局部变量
    function f(){                   // 闭包函数
        alert(a);                   // 读取变量 a 的值，返回 3
        var b;                      // 声明局部变量 b
        alert(b);                   // 读取变量 b 的值，返回 undefined
    }
    return f                        // 返回闭包函数
}
var c = f();                        // 调用函数，返回闭包函数
c();                                // 调用闭包函数
```

可能你会误认为，闭包函数是在全局作用域中被调用的，则它应该读取全局变量 a，而不是局部变量 a。同时，读取全局变量 b 的值，因为局部作用域中没有为变量 b 赋值。

通过上面演示示例可以看到，在闭包函数中，变量的作用域并没有因为使用闭包而发生改变，同时函数与闭包之间的作用域链也没有因为返回的闭包而破坏。在闭包函数中，它会读取上一级函数中的局部变量 a，而不是全局变量 a 的值。换句话说，闭包函数是根据定义作用域来确定自己包含变量的作用域以及作用域链，而不是根据闭包函数的执行作用域来确定。可能提及这些概念对于初学者来说，有点生疏，但是你可以在学习完后面章节的知识之后，再回头阅读上面示例代码就容易理解了。同时，读取的变量 b 的值为 undefined，

而不是全局变量 b 的值，这说明一旦当在闭包函数中声明了变量，它就不再受外部变量的影响。

4.5.4 圈里圈外——属性与变量

简单一看，在 JavaScript 中变量与属性是两个截然不同的概念，实际上它们在本质上是相同的，用法也基本相同。下面就从全局变量和局部变量来探索两者之间的联系。

1．全局变量是全局对象的属性

我们曾经介绍过 JavaScript 解释器在执行 JavaScript 代码之前，有一个预编译处理期，在这个处理过程中，JavaScript 解释器会首先创建一个对象，它就是全局对象（global）。然后为该对象定义全局属性，这些属性就是 JavaScript 全局变量。同时，JavaScript 解释器还会使用预定义的值和函数来初始化全局对象的属性。例如，undefined 属性值为 undefined、Infinity 属性值为 Infinity、null 属性值为 null、所有用户自定义变量的初始值为 undefined 等。在全局作用域中，我们可以使用 this 关键字来引用全局对象，于是你就可以这样来读取变量：

```
var a = 1;                          // 定义全局变量
alert(this.a);                      // 使用全局对象的属性方式来读取变量值
```

在客户端 JavaScript 中，window 对象代表浏览窗口，它表示全局对象，并允许使用 window 属性引用自己，因此我们也可以这样来读取变量：

```
var a = 1;
alert(window.a);                    // 使用全局对象 window 来读取变量值
```

总之，所有全局变量都是 window 对象的属性，全局变量的作用域实际上也是 window 对象的包含范围。除了用户自定义的全局变量外，客户端 JavaScript 也定义了大量全局属性。它们都可以在 Window 窗口中自由使用。

2．局部变量是调用对象的属性

window 是全局对象，所有全局变量都是它的属性。也许你很容易接受这样的观点，但是对于函数内的局部变量，它们的对象是谁？难道它就是函数自己？

不是。当我们声明函数时，JavaScript 会自动建立一个对象，这个对象被称为调用对象。在预编译期函数处理时，函数的参数和局部变量都将作为调用对象的属性进行存储。使用独立的调用对象来存储局部变量，这样就避免了全局变量与局部变量之间的相互覆盖。不过，这个调用对象是不能够访问的，我们只能够想象着它就存在那儿，不如全局对象可以使用 window 属性和 this 关键字进行引用。

4.5.5 变量的作用域链

在前面两节中我们介绍了两个重要知识点：第一，局部变量都比全局变量的优先权高，内层局部变量又比外层局部变量的优先权高；第二，变量都是对象的属性，全局变量是 window 对象的属性，局部变量是调用对象的变量。下面我们就来研究一下 JavaScript 解释器是如何访问变量的？

变量的作用域是基于 JavaScript 代码的词法结构来定义的，而不是根据它的执行结构来确定的，关于这个观点我们曾经在第 4.5.3 节中的闭包函数的作用域中分析过，我们可以再温习一下：

- 全局变量具有全局作用域。
- 在函数体内声明的变量具有局部作用域。
- 如果一个定义函数（注意，是定义函数，而不是调用或执行函数）被嵌套在另一个函数中，那么在嵌套的函数体内声明的变量就具有嵌套的局部作用域。

有了上面的知识基础，下面我们就来讨论变量的作用域链。作用域链是 JavaScript 提供的一套如何解决变量访问的机制。JavaScript 规定每一个执行环境（作用域）都有一个与之相关联的作用域链。

所谓的作用域链，实际上就是一个对象列表，多个对象串联在一起，犹如一个链条，故称为作用域链。当 JavaScript 代码需要存取变量的值时，JavaScript 解释器会就近查询当前作用域对应的对象是否存在同名属性。如果该对象有同名属性，那么就采用这个属性的值。如果该对象没有同名属性，则 JavaScript 就会向上继续查询作用域链中的上一级对象。如果上一级对象仍然没有同名属性，那么就继续向上查询对象，以此类推。最后，查询到作用域链的顶部(即全局对象),如果在全局对象中仍然没有找到同名属性,则返回 undefined 的属性值。下面以一个示例进行说明：

```
var a = 1;                              // 全局变量
(function(){
   var b = 2;                           // 第 1 层局部变量
   (function(){
      var c = 3;                        // 第 2 层局部变量
      (function(){
         var d = 4;                     // 第 3 层局部变量
         alert(a+b+c+d);                // 返回 10
      })()                              // 直接调用函数
   })()                                 // 直接调用函数
})()                                    // 直接调用函数
```

在这个示例中，JavaScript 解释器首先在最内层调用对象中查询属性 a、b、c 和 d，其中只找到了属性 d，并获得它的值（4），然后沿着作用域链，在上一层调用对象中继续查找属性 a、b 和 c，其中找到了属性 c，获得它的值（3），依次类推，直到找到所有需要的变量值为止（如图 4-5 所示）。

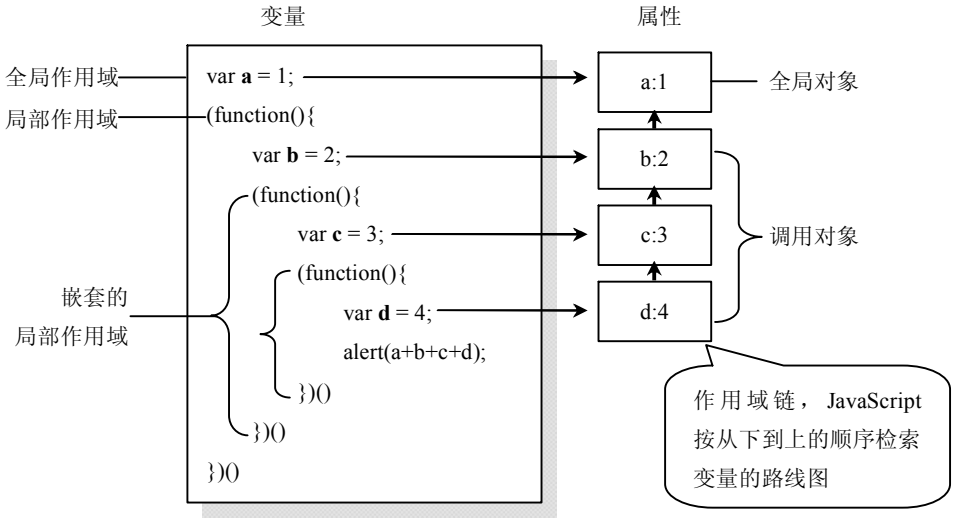

图 4-5　变量的作用域链

4.5.6　变量的垃圾回收

我们都知道，在超市购买食品，东西吃完之后，或者东西不用了，就需要对大大小小的各种包装袋、包装盒、不用或变质的食品进行处理，或卖给收废品的，或者扔到垃圾桶中。总之，如果不及时清理垃圾，屋子就会被这些无用的垃圾所占据，浪费空间。

进行一个对比，如果说内存就是居家，那么食品就是数据，而变量就是食品包装盒了。如何处理不用的变量或数据呢？应该说，不同语言都有自己的一套方法，在 C 语言中，垃圾需要人工手动清理，这个比较麻烦。你必须知道哪些东西没用，哪些还有用，或者还有潜在作用。工作起来还是比较麻烦的，弄不好会产生 Bug，给程序带来严重的后果。好在，JavaScript 已经能够自动化处理这件事情，用户不用再操心了。

当我们每次创建字符串、数组或对象时，JavaScript 解释器都必须分配内存来存储这些数据。由于字符串、对象和数组大小是不固定的，所以也只有当它们的大小确定时，JavaScript 才为它们动态分配存储空间。JavaScript 能够动态分配存储空间，当然也能够自动清理无用的数据，这里面就有一个关键问题：如何确定安全回收内存的时机。决不能回收那些仍在使用的值，而应该收集再也不会使用的值，也就是那些不会再由程序中的任何变量、对象的属性或数组的元素引用的值。

JavaScript 包含一个垃圾回收的小程序，这个程序能够周期性的遍历 JavaScript 环境中的所有变量的列表，并且给这些变量所引用的值做个标记。如果被引用的值是对象或数组，那么对象的属性或者数组的元素就会被递归地做个标记。通过递归遍历所有值的树或者图，垃圾回收器就能够找到（并标记）仍旧使用的每个值。那些没有标记的值就是无用的存储单元。

当给所有正在使用的变量做完标记之后，垃圾回收器就会开始进行清除。在这个阶段中，它将遍历环境中所有值的列表，同时释放那些没有标记的值。

例如，在下面这个示例中，变量 a 指向一块内存空间（堆区），这个空间中存储的是字

符串"javascript"，然后我们再给变量 a 赋其他值，这时候在内存中字符串"javascript"所占据的空间就没有被任何变量引用，此时 JavaScript 垃圾回收器就会把这个字符串视为垃圾，并执行回收，释放它占据的内存空间。

```
var a = "javascript";
a = 123456;
```

同时，如果我们为变量 a 赋值为 null，则 JavaScript 垃圾回收器就知道这个变量也没有用，于是把这个变量视为垃圾一并进行回收（栈区），也就是说，如果一个变量、属性、元素或对象被赋值为 null，也就意味着它们是无用的垃圾了，JavaScript 垃圾回收器将择机对其进行回收。

```
a = null;
```