第8章

函数

函数是这样的一段JavaScript代码，它只定义一次，但可能被执行或调用任意次。你可能已经从诸如子例程（subroutine）或者过程（procedure）这些名字里对函数的概念有所了解。JavaScript函数是参数化的：函数的定义会包括一个称为形参（parameter）的标识符列表，这些参数在函数体中像局部变量一样工作。函数调用会为形参提供实参的值[译注1]。函数使用它们实参的值来计算返回值，成为该函数调用表达式的值。除了实参之外，每次调用还会拥有另一个值——本次调用的上下文——这就是this关键字的值。

如果函数挂载在一个对象上，作为对象的一个属性，就称它为对象的方法。当通过这个对象来调用函数时，该对象就是此次调用的上下文（context），也就是该函数的this的值。用于初始化一个新创建的对象的函数称为构造函数（constructor）。6.1节会对构造函数有进一步的讲解，第9章还会再谈到它。

在JavaScript里，函数即对象，程序可以随意操控它们。比如，JavaScript可以把函数赋值给变量，或者作为参数传递给其他函数。因为函数就是对象，所以可以给它们设置属性，甚至调用它们的方法。

JavaScript的函数可以嵌套在其他函数中定义，这样它们就可以访问它们被定义时所处的作用域中的任何变量。这意味着JavaScript函数构成了一个闭包（closure），它给JavaScript带来了非常强劲的编程能力。

译注1：参数有形参（parameter）和实参（argument）的区别，形参相当于函数中定义的变量，实参是在运行时的函数调用时传入的参数。

8.1 函数定义

函数使用 function 关键字来定义，它可以用在函数定义表达式（见4.3节）或者函数声明语句（见5.3.2节）里。在两种形式中，函数定义都从function关键字开始，其后跟随这些组成部分：

- 函数名称标识符。函数名称是函数声明语句必需的部分。它的用途就像变量的名字，新定义的函数对象会赋值给这个变量。对函数定义表达式来说，这个名字是可选的：如果存在，该名字只存在于函数体中，并指代该函数对象本身。
- 一对圆括号，其中包含由0个或者多个用逗号隔开的标识符组成的列表。这些标识符是函数的参数名称，它们就像函数体中的局部变量一样。
- 一对花括号，其中包含0条或多条JavaScript语句。这些语句构成了函数体：一旦调用函数，就会执行这些语句。

例8-1 分别展示了函数语句和表达式两种方式的函数定义。注意，以表达式来定义函数只适用于它作为一个大的表达式的一部分，比如在赋值和调用过程中定义函数：

例 8-1：定义JavaScript函数

```
//输出o的每个属性的名称和值，返回undefined
function printprops(o) {
    for(var p in o)
        console.log(p + ": " + o[p] + "\n");
}

// 计算两个笛卡尔坐标 (x1,y1) 和 (x2,y2)之间的距离
function distance(x1, y1, x2, y2) {
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

// 计算阶乘的递归函数（调用自身的函数）
// x!的值是从x到x递减（步长为1）的值的累乘
function factorial(x) {
    if (x <= 1) return 1;
    return x * factorial(x-1);
}

// 这个函数表达式定义了一个函数用来求传入参数的平方
// 注意我们把它赋值给一个变量
var square = function(x) { return x*x; }

// 函数表达式可以包含名称，这在递归时很有用
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };

// 函数表达式也可以作为参数传给其他函数
```

```
data.sort(function(a,b) { return a-b; });

//函数表达式有时定义后立即调用
var tensquared = (function(x) {return x*x;}(10));
```

注意：以表达式方式定义的函数，函数的名称是可选的。一条函数声明语句实际上声明了一个变量，并把一个函数对象赋值给它。相对而言，定义函数表达式时并没有声明一个变量。函数可以命名，就像上面的阶乘函数，它需要一个名称来指代自己。如果一个函数定义表达式包含名称，函数的局部作用域将会包含一个绑定到函数对象的名称。实际上，函数的名称将成为函数内部的一个局部变量。通常而言，以表达式方式定义函数时都不需要名称，这会让定义它们的代码更为紧凑。函数定义表达式特别适合用来定义那些只会用到一次的函数，比如上面展示的最后两个例子。

函数命名

任何合法的JavaScript标识符都可以用做一个函数的名称。命名时要尽量选择描述性强而又简洁的函数名。在这两者之间做到恰到好处是一门艺术，需要丰富的经验。精心挑选的函数名可以极大地改善代码的可读性（从而也提高了可维护性）。

函数名称通常是动词或以动词为前缀的词组。通常函数名的第一个字符为小写，这是一种编程约定。当函数名包含多个单词时，一种约定是将单词以下划线分隔，就像`like_this()`。还有另外一种约定，就是除了第一个单词之外的单词首字母使用大写字母，就像`likeThis()`。有一些函数是用做内部函数或私有函数（不是作为公用API的一部分），这种函数名通常以一条下划线为前缀。

在一些编程风格中，或者编程框架里，通常为那些经常调用的函数指定短名称，比如客户端JavaScript框架jQuery（第19章会详细讲述）就将最常用的方法重命名为`$()`（一个美元符号）（2.4节提到，美元符号和下划线是除了字母和数字之外的两个合法的JavaScript标识符）。

如5.3.2节所述，函数声明语句“被提前”到外部脚本或外部函数作用域的顶部，所以以这种方式声明的函数，可以被在它定义之前出现的代码所调用。不过，以表达式定义的函数就另当别论了，为了调用一个函数，必须要能引用它，而要使用一个以表达式方式定义的函数之前，必须把它赋值给一个变量。变量的声明提前了（参见3.10.1节），但给变量赋值是不会提前的，所以，以表达式方式定义的函数在定义之前无法调用）。

请注意，例8-1中的大多数函数（但不是全部）包含一条`return`语句（见5.6.4节）。`return`语句导致函数停止执行，并返回它的表达式（如果有的话）的值给调用者。如果

return语句没有一个与之相关的表达式，则它返回undefined值。如果一个函数不包含return语句，那它就只执行函数体中的每条语句，并返回undefined值给调用者。

例8-1中的大多数函数都是用来计算出一个值的，它们使用return把值返回给调用者。而printprops()函数的不用之处在于，它的任务是输出对象各属性的名称和值。没有必要返回值，该函数不包含return语句。printprops()函数的返回值始终是undefined。（没有返回值的函数有时候称为过程）

嵌套函数

在 JavaScript 里，函数可以嵌套在其他函数里。例如：

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

嵌套函数的有趣之处在于它的变量作用域规则：它们可以访问嵌套它们（或多重嵌套）的函数的参数和变量。例如，在上面的代码里，内部函数square()可以读写外部函数hypotenuse()定义的参数a和b。这些作用域规则对内嵌函数非常重要，我们会在8.6节再深入了解它们。

5.3.2节曾提到，函数声明语句并非真正的语句，ECMAScript规范只是允许它们作为顶级语句。它们可以出现在全局代码里，或者内嵌在其他函数中，但它们不能出现在循环、条件判断，或者try/cache/finally以及with语句中[注1]。注意，此限制仅适用于以语句声明形式定义的函数。函数定义表达式可以出现在JavaScript代码的任何地方。

8.2 函数调用

构成函数主体的JavaScript代码在定义之时并不会执行，只有调用该函数时，它们才会执行。有4种方式来调用JavaScript函数：

- 作为函数
- 作为方法
- 作为构造函数
- 通过它们的call()和apply()方法间接调用

注1：有些JavaScript的实现并未严格遵守这条规则，比如，Firefox就允许在if语句中出现条件函数声明。

8.2.1 函数调用

使用调用表达式可以进行普通的函数调用也可进行方法调用（见4.5节）。一个调用表达式由多个函数表达式组成，每个函数表达式都是由一个函数对象和左圆括号、参数列表和右圆括号组成，参数列表是由逗号分隔的零个或多个参数表达式组成。如果函数表达式是一个属性访问表达式，即该函数是一个对象的属性或数组中的一个元素，那么它就是一个方法调用表达式。下面将会解释这种情形。下面的代码展示了一些普通的函数调用表达式：

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

在一个调用中，每个参数表达式（圆括号之间的部分）都会计算出一个值，计算的结果作为参数传递给另外一个函数。这些值作为实参传递给声明函数时定义的形参。在函数体中存在一个形参的引用，指向当前传入的实参列表，通过它可以获得参数的值。

对于普通的函数调用，函数的返回值成为调用表达式的值。如果该函数返回是因为解释器到达结尾，返回值就是undefined。如果函数返回是因为解释器执行到一条return语句，返回值就是return之后的表达式的值，如果return语句没有值，则返回undefined。

根据 ECMAScript 3 和非严格的 ECMAScript 5对函数调用的规定，调用上下文（this的值）是全局对象。然而，在严格模式下，调用上下文则是undefined。

以函数形式调用的函数通常不使用this关键字。不过，“this”可以用来判断当前是否是严格模式。

```
//定义并调用一个函数来确定当前脚本运行时是否为严格模式
var strict = (function() { return !this; }());
```

8.2.2 方法调用

一个方法无非是个保存在一个对象的属性里的JavaScript函数。如果有一个函数f和一个对象o，则可以用下面的代码给o定义一个名为m()的方法：

```
o.m = f;
```

给对象o定义了方法m()，调用它时就像这样：

```
o.m();
```

或者，如果m()需要两个实参，调用起来则像这样：

```
o.m(x, y);
```

上面的代码是一个调用表达式：它包括一个函数表达式o.m，以及两个实参表达式x和y，函数表达式本身就是一个属性访问表达式（见4.4节），这意味着该函数被当做一个方法，而不是作为一个普通函数来调用。

对方法调用的参数和返回值的处理，和上面所描述的普通函数调用完全一致。但是，方法调用和函数调用有一个重要的区别，即：调用上下文。属性访问表达式由两部分组成：一个对象（本例中的o）和属性名称（m）。在像这样的方法调用表达式里，对象o成为调用上下文，函数体可以使用关键字this引用该对象。下面是一个具体的例子：

```
var calculator = { //对象直接量
    operand1: 1,
    operand2: 1,
    add: function() {
    //注意this关键字的用法，this指代当前对象
        this.result = this.operand1 + this.operand2;
    }
};
calculator.add(); //这个方法调用计算1+1的结果
calculator.result // => 2
```

大多数方法调用使用点符号来访问属性，使用方括号（的属性访问表达式）也可以进行属性访问操作。下面两个例子都是函数调用：

```
o["m"](x,y); // o.m(x,y)的另外一种写法
a[0](z)      //同样是一个方法调用（这里假设a[0]是一个函数）
```

方法调用可能包括更复杂的属性访问表达式：

```
customer.surname.toUpperCase();        //调用customer.surname的方法
f().m();                               //在f()调用结束后继续调用返回值中的方法m()
```

方法和this关键字是面向对象编程范例的核心。任何函数只要作为方法调用实际上都会传入一个隐式的实参——这个实参是一个对象，方法调用的母体就是这个对象。通常来讲，基于那个对象的方法可以执行多种操作，方法调用的语法已经很清晰地表明了函数将基于一个对象进行操作，比较下面两行代码：

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

我们假设这两行代码的功能完全一样，它们都作用于一个假定的对象rect。可以看出，第一行的方法调用语法非常清晰地表明这个函数执行的载体是rect对象，函数中的所有操作都将基于这个对象。

方法链

当方法的返回值是一个对象，这个对象还可以再调用它的方法。这种方法调用序列中（通常称为“链”或者“级联”）每次的调用结果都是另外一个表达式的组成部分。比如，基于jQuery库（参见第19章），我们常常会这样写代码：

```
// 找到所有的header，取得它们id的映射，转换为数组并对它们进行排序
$(":header").map(function() { return this.id }).get().sort();
```

当方法并不需要返回值时，最好直接返回this。如果在设计的API中一直采用这种方式（每个方法都返回this），使用API就可以进行“链式调用”[注2]风格的编程，在这种编程风格中，只要指定一次要调用的对象即可，余下的方法都可以基于此进行调用：

```
shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();
```

不要将方法的链式调用和构造函数的链式调用混为一谈，9.7.2节将会讨论构造函数的链式调用。

需要注意的是，this是一个关键字，不是变量，也不是属性名。JavaScript的语法不允许给this赋值。

和变量不同，关键字this没有作用域的限制，嵌套的函数不会从调用它的函数中继承this。如果嵌套函数作为方法调用，其this的值指向调用它的对象。如果嵌套函数作为函数调用，其this值不是全局对象（非严格模式下）就是undefined（严格模式下）。很多人误以为调用嵌套函数时this会指向调用外层函数的上下文。如果你想访问这个外部函数的this值，需要将this的值保存在一个变量里，这个变量和内部函数都同在一个作用域内。通常使用变量self来保存this，比如：

```
var o = {                               // 对象o
    m: function() {                     // 对象中的方法m()
        var self = this;                // 将this的值保存至一个变量中
        console.log(this === o);        // 输出true，this就是这个对象o
        f();                            // 调用辅助函数f()

        function f() {                  // 定义一个嵌套函数f()
            console.log(this === o);    // "false": this的值是全局对象或undefined
            console.log(self === o);    // "true": self指外部函数的this值
        }
    }
};
```

注2：这个术语最初是由Martin Fowler提出的，参见*http://martinfowler.com/dslwip/MethodChaining.html*。

```
o.m();                                  // 调用对象o的方法m()
```

在8.7.4节的例8-5中有var self=this更切合实际的用法。

8.2.3 构造函数调用

如果函数或者方法调用之前带有关键字new，它就构成构造函数调用（构造函数调用在4.6节和6.1.2节有简单介绍，第9章会对构造函数做更详细的讨论）。构造函数调用和普通的函数调用以及方法调用在实参处理、调用上下文和返回值方面都有不同。

如果构造函数调用在圆括号内包含一组实参列表，先计算这些实参表达式，然后传入函数内，这和函数调用和方法调用是一致的。但如果构造函数没有形参，JavaScript构造函数调用的语法是允许省略实参列表和圆括号的。凡是没有形参的构造函数调用都可以省略圆括号，比如，下面这两行代码就是等价的：

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

构造函数调用创建一个新的空对象，这个对象继承自构造函数的prototype属性。构造函数试图初始化这个新创建的对象，并将这个对象用做其调用上下文，因此构造函数可以使用this关键字来引用这个新创建的对象。注意，尽管构造函数看起来像一个方法调用，它依然会使用这个新对象作为调用上下文。也就是说，在表达式new o.m()中，调用上下文并不是o。

构造函数通常不使用return关键字，它们通常初始化新对象，当构造函数的函数体执行完毕时，它会显式返回。在这种情况下，构造函数调用表达式的计算结果就是这个新对象的值。然而如果构造函数显式地使用return语句返回一个对象，那么调用表达式的值就是这个对象。如果构造函数使用return语句但没有指定返回值，或者返回一个原始值，那么这时将忽略返回值，同时使用这个新对象作为调用结果。

8.2.4 间接调用

JavaScript中的函数也是对象，和其他JavaScript对象没什么两样，函数对象也可以包含方法。其中的两个方法call()和apply()可以用来间接地调用函数。两个方法都允许显式指定调用所需的this值，也就是说，任何函数可以作为任何对象的方法来调用，哪怕这个函数不是那个对象的方法。两个方法都可以指定调用的实参。call()方法使用它自有的实参列表作为函数的实参，apply()方法则要求以数组的形式传入参数。8.7.3节会有关于call()和apply()方法的详细讨论。

8.3 函数的实参和形参

JavaScript中的函数定义并未指定函数形参的类型，函数调用也未对传入的实参值做任何类型检查。实际上，JavaScript函数调用甚至不检查传入形参的个数。下面几节将会讨论当调用函数时的实参个数和声明的形参个数不匹配时出现的状况，同样说明了如何显式测试函数实参的类型，以避免非法的实参传入函数。

8.3.1 可选形参

当调用函数的时候传入的实参比函数声明时指定的形参个数要少，剩下的形参都将设置为undefined值。因此在调用函数时形参是否可选以及是否可以省略应当保持较好的适应性。为了做到这一点，应当给省略的参数赋一个合理的默认值，来看这个例子：

```
// 将对象o中可枚举的属性名追加至数组a中，并返回这个数组a
// 如果省略a，则创建一个新数组并返回这个新数组
function getPropertyNames(o, /* optional */ a) {
    if (a === undefined) a = []; // 如果未定义，则使用新数组
    for (var property in o) a.push(property);
    return a;
}

// 这个函数调用可以传入1个或2个实参
var a = getPropertyNames(o);  // 将o的属性存储到一个新数组中
getPropertyNames(p, a);       // 将p的属性追加至数组a中
```

如果在第一行代码中不使用if语句，可以使用“||”运算符，这是一种习惯用法[译注2]：

```
a = a || [];
```

回忆一下，4.10.2节介绍了“||”运算符，如果第一个实参是真值的话就返回第一个实参；否则返回第二个实参。在这个场景下，如果作为第二个实参传入任意对象，那么函数就会使用这个对象。如果省略掉第二个实参（或者传递null以及其他任何假值），那么就新创建一个空数组，并赋值给a。

需要注意的是，当用这种可选实参来实现函数时，需要将可选实参放在实参列表的最后。那些调用你的函数的程序员是没办法省略第一个实参并传入第二个实参的，它必须将undefined作为第一个实参显式传入[译注3]。同样注意在函数定义中使用注释/*optional*/来强调形参是可选的。

译注2：需要注意的是，使用“||”运算符代替if语句的前提是a必须预先声明，否则a=a||[]会报引用错误，在这个例子中a是作为形参传入的，相当于var a，即已经声明了a，所以这样用是没有问题的。

译注3：当函数的实参可选时往往传入一个无意义的占位符，惯用做法是传入null作为占位符，当然也可以使用undefined作为占位符。

8.3.2 可变长的实参列表：实参对象

当调用函数的时候传入的实参个数超过函数定义时的形参个数时，没有办法直接获得未命名值的引用。参数对象解决了这个问题。在函数体内，标识符arguments是指向实参对象的引用，实参对象是一个类数组对象（参照7.11节），这样可以通过数字下标就能访问传入函数的实参值，而不用非要通过名字来得到实参。

假设定义了函数f，它的实参只有一个x。如果调用这个函数时传入两个实参，第一个实参可以通过参数名x来获得，也可以通过arguments[0]来得到。第二个实参只能通过arguments[1]来得到。此外，和真正的数组一样，arguments也包含一个length属性，用以标识其所包含元素的个数。因此，如果调用函数f()时传入两个参数，arguments.length的值就是2。

实参对象在很多地方都非常有用，下面的例子展示了使用它来验证实参的个数，从而调用正确的逻辑，因为JavaScript本身不会这么做：

```
function f(x, y, z)
{
    //首先，验证传入实参的个数是否正确
    if (arguments.length != 3) {
        throw new Error("function f called with " + arguments.length +
                        "arguments, but it expects 3 arguments.");
    }
    // 再执行函数的其他逻辑...
}
```

需要注意的是，通常不必像这样检查实参个数。大多数情况下JavaScript的默认行为是可以满足需要的：省略的实参都将是undefined，多出的参数会自动省略。

实参对象有一个重要的用处，就是让函数可以操作任意数量的实参。下面的函数就可以接收任意数量的实参，并返回传入实参的最大值（内置函数Max.max()的功能与之类似）：

```
function max(/* ... */) {
    var max = Number.NEGATIVE_INFINITY;
    // 遍历实参，查找并记住最大值
    for (var i = 0; i < arguments.length; i++)
        if (arguments[i] > max) max = arguments[i];
    // 返回最大值
    return max;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6); // => 10000
```

类似这种函数可以接收任意个数的实参，这种函数也称为“不定实参函数”（varargs function）[译注4]，这个术语源自古老的C语言。

注意，不定实参函数的实参个数不能为零，arguments[]对象最适合的应用场景是在这样一类函数中，这类函数包含固定个数的命名和必需参数，以及随后个数不定的可选实参。

记住，arguments并不是真正的数组，它是一个实参对象。每个实参对象都包含以数字为索引的一组元素以及length属性，但它毕竟不是真正的数组。可以这样理解，它是一个对象，只是碰巧具有以数字为索引的属性。参照7.11节以获得更多关于类数组对象的信息。

数组对象包含一个非同寻常的特性。在非严格模式下，当一个函数包含若干形参，实参对象的数组元素是函数形参所对应实参的别名，实参对象中以数字索引，并且形参名称可以认为是相同变量的不同命名。通过实参名字来修改实参值的话，通过arguments[]数组也可以获取到更改后的值，下面这个例子清楚地说明了这一点：

```
function f(x) {
    console.log(x);           // 输出实参的初始值
    arguments[0] = null;      // 修改实参数组的元素同样会修改x的值
    console.log(x);           // 输出 "null"
}
```

如果实参对象是一个普遍数组的话，第二条console.log(x)语句的结果绝对不会是null，在这个例子中，arguments[0]和x指代同一个值，修改其中一个的值会影响到另一个。

在ECMAScript 5中移除了实参对象的这个特殊特性。在严格模式下还有一点（和非严格模式下相比的）不同，在非严格模式中，函数里的arguments仅仅是一个标识符，在严格模式中，它变成了一个保留字。严格模式中的函数无法使用arguments作为形参名或局部变量名，也不能给arguments赋值。

callee和caller属性

除了数组元素，实参对象还定义了callee和caller属性。在ECMAScript 5严格模式中，对这两个属性的读写操作都会产生一个类型错误。而在非严格模式下，ECMAScript标准规范规定callee属性指代当前正在执行的函数。caller是非标准的，但大多数浏览器都实现了这个属性，它指代调用当前正在执行的函数的函数。通过caller属性可以访问调

译注4：原文用了三个单词来描述“不定实参函数”，variadic function、variable arity function和varargs function，variadic的含义是实参（模板）的顺序不定，variable arity的含义是实参的个数不定，varargs的含义是实参的值不定，这里统一译成“不定实参函数”。本书采用最通俗的术语“不定实参”（vararg）。作者在这里选用最常见的一种情形，即“实参的值不定”，但在后续章节中，这个单词的含义应当是包含前两种情形的，即包含实参顺序不定和实参个数不定。

用栈。callee属性在某些时候会非常有用，比如在匿名函数中通过callee来递归地调用自身。

```
var factorial = function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x-1);
};
```

8.3.3 将对象属性用做实参

当一个函数包含超过三个形参时，对于程序员来说，要记住调用函数中实参的正确顺序实在让人头疼。每次调用这个函数时都要不厌其烦地查阅文档，为了不让程序员每次都翻阅手册这么麻烦，最好通过名/值对的形式来传入参数，这样参数的顺序就无关紧要了。为了实现这种风格的方法调用，定义函数的时候，传入的实参都写入一个单独的对象之中，在调用的时候传入一个对象，对象中的名/值对是真正需要的实参数据。下面的代码就展示了这种风格的函数调用，这种写法允许在函数中设置省略参数的默认值：

```
// 将原始数组的length元素复制至目标数组
// 开始复制原始数组的from_start元素
// 并且将其复制至目标数组的to_start中
// 要记住实参的顺序并不容易
function arraycopy(/* array */ from, /* index */ from_start,
                   /* array */ to, /* index */ to_start,
                   /* integer */ length)
{
    // 逻辑代码
}

// 这个版本的实现效率稍微有些低，但你不必再去记住实参的顺序
// 并且from_start和to_start都默认为0
function easycopy(args) {
    arraycopy(args.from,
              args.from_start || 0, // 注意这里设置了默认值
              args.to,
              args.to_start || 0, args.length);
}
// 来看如何调用easycopy()
var a = [1, 2, 3, 4], b = [];
easycopy({ from: a, to: b, length: 4 });
```

8.3.4 实参类型

JavaScript方法的形参并未声明类型，在形参传入函数体之前也未做任何类型检查。可以采用语义化的单词来给函数实参命名，或者像刚才的示例代码中的arraycopy()方法一样给实参补充注释，以此使代码自文档化，对于可选的实参来说，可以在注释中补充一下“这个实参是可选的”。当一个方法可以接收任意数量的实参时，可以使用省略号：

```
function max(/* number... */) { /* 代码区 */ }
```

3.8节已经提到，JavaScript在必要的时候会进行类型转换。因此如果函数期望接收一个字符串实参，而调用函数时传入其他类型的值，所传入的值会在函数体内将其用做字符串的地方转换为字符串类型。所有的原始类型都可以转换为字符串，所有的对象都包含toString()方法（尽管不一定有用），所以这种场景下是不会有任何错误的。

然而事情不总是这样，回头看一下刚才提到的arraycopy()方法。这个方法期望它的第一个实参是一个数组。当传入一个非数组的值作为第一个实参时（通常会传入类数组对象），尽管看起来是没问题的，实际上会出错。除非所写的函数是只用到一两次的“用完即丢”函数，你应当添加类似的实参类型检查逻辑，因为宁愿程序在传入非法值时报错，也不愿非法值导致程序在执行时报错，相比而言，逻辑执行时的报错消息不甚清晰且更难处理。下面这个例子中的函数就做了这种类型检查。注意这里使用了7.11节的isArrayLike()函数：

```
// 返回数组（或类数组对象）a的元素的累加和
// 数组a中必须为数字、null和undefined的元素都将忽略
function sum(a) {
    if (isArrayLike(a)) {
        var total = 0;
        for (var i = 0; i < a.length; i++) { // 遍历所有元素
            var element = a[i];
            if (element == null) continue; // 跳过null和undefined
            if (isFinite(element)) total += element;
            else throw new Error("sum(): elements must be finite numbers");
        }
        return total;
    }
    else throw new Error("sum(): argument must be array-like");
}
```

这里的sum()方法进行了非常严格的实参检查，当传入非法的值时会给出容易看懂的错误提示信息。但当涉及类数组对象和真正的数组（不考虑数组元素是否是null还是undefined），这种做法带来的灵活性其实并不大。

JavaScript是一种非常灵活的弱类型语言，有时适合编写实参类型和实参个数的不确定性的函数。接下来的flexisum()方法就是这样（可能走向了一个极端）。比如，它可以接收任意数量的实参，并可以递归地处理实参是数组的情况，这样的话，它就可以用做不定实参函数或者实参是数组的函数。此外，这个方法尽可能的在抛出异常之前将非数字转换为数字：

```
function flexisum(a) {
    var total = 0;
    for (var i = 0; i < arguments.length; i++) {
```

```
        var element = arguments[i], n;
        if (element == null) continue;          // 忽略null和undefined实参
        if (isArray(element))                   // 如果实参是数组
            n = flexisum.apply(this, element);  // 递归地计算累加和
        else if (typeof element === "function") // 否则，如果是函数...
            n = Number(element());              // 调用它并做类型转换
        else
            n = Number(element);                // 否则直接做类型转换
        if (isNaN(n)) // 如果无法转换为数字，则抛出异常
            throw Error("flexisum(): can't convert " + element + " to number");
        total += n;   // 否则，将n累加至total
    }
    return total;
}
```

8.4 作为值的函数

函数可以定义，也可以调用，这是函数最重要的特性。函数定义和调用是JavaScript的词法特性，对于其他大多数编程语言来说亦是如此。然而在JavaScript中，函数不仅是一种语法，也是值，也就是说，可以将函数赋值给变量，存储在对象的属性或数组的元素中，作为参数传入另外一个函数等[注3]。

为了便于理解JavaScript中的函数是如何用做数据的以及JavaScript语法，来看一下这样一个函数定义：

```
function square(x) { return x*x; }
```

这个定义创建一个新的函数对象，并将其赋值给变量`square`。函数的名字实际上是看不见的，它（square）仅仅是变量的名字，这个变量指代函数对象。函数还可以赋值给其他的变量，并且仍可以正常工作：

```
var s = square;       // 现在s和square指代同一个函数
square(4);            // => 16
s(4);                 // => 16
```

除了可以将函数赋值给变量，同样可以将函数赋值给对象的属性。当函数作为对象的属性调用时，函数就称为方法：

```
var o = {square: function(x) { return x*x; }}; // 对象直接量
var y = o.square(16);                          // y 等于 256
```

函数甚至不需要带名字，当把它们赋值给数组元素时：

注3：这看起来不足为奇，但如果你对Java很熟悉，你会发现Java中的函数是程序的一部分，但无法被程序操作。

```
var a = [function(x) { return x*x; }, 20]; // 数组直接量
a[0](a[1]); // => 400
```

最后一句代码看起来很奇怪，但的确是合法的函数调用表达式！

例8-2展示了将函数用做值时的一些例子，这段代码可能会难读一些，但注释解释了代码的具体含义：

例8-2：将函数用做值

```
// 在这里定义一些简单的函数
function add(x, y) { return x + y; }
function subtract(x, y) { return x - y; }
function multiply(x, y) { return x * y; }
function divide(x, y) { return x / y; }

// 这里的函数以上面的某个函数作为参数
// 并给它传入两个操作数然后调用它
function operate(operator, operand1, operand2) {
    return operator(operand1, operand2);
}

// 这行代码所示的函数调用实际上计算了 (2+3) + (4*5) 的值
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// 我们为这个例子重复实现一个简单的函数
// 这次实现使用函数直接量，这些函数直接量定义在一个对象直接量中
var operators = {
    add: function(x, y) { return x + y; },
    subtract: function(x, y) { return x - y; },
    multiply: function(x, y) { return x * y; },
    divide: function(x, y) { return x / y; },
    pow: Math.pow //使用预定义的函数
};

// 这个函数接收一个名字作为运算符，在对象中查找这个运算符
// 然后将它作用于所提供的操作数
// 注意这里调用运算符函数的语法
function operate2(operation, operand1, operand2) {
    if (typeof operators[operation] === "function")
        return operators[operation](operand1, operand2);
    else throw "unknown operator";
}

// 这样来计算 ("hello" + " " + "world")  的值
var j = operate2("add", "hello", operate2("add", " ", "world"));
// 使用预定义的函数Math.pow()
var k = operate2("pow", 10, 2);
```

这里是将函数用做值的另外一个例子，考虑一下Array.sort()方法。这个方法用来对数组元素进行排序。因为排序的规则有很多[译注5]（基于数值大小、字母表顺序、日期大小、从小到大、从大到小等），sort()方法可以接收一个函数作为参数，用来处理具体

的排序操作。这个函数的作用非常简单，对于任意两个值都返回一个值，以指定它们在排序后的数组中的先后顺序。这个函数参数使得Array.sort()具有更完美的通用性和无限可扩展性，它可以对任何类型的数据进行任意排序。7.8.3节有示例代码。

自定义函数属性

JavaScript中的函数并不是原始值，而是一种特殊的对象，也就是说，函数可以拥有属性。当函数需要一个“静态”变量来在调用时保持某个值不变，最方便的方式就是给函数定义属性，而不是定义全局变量，显然定义全局变量会让命名空间变得更加杂乱无章。比如，假设你想写一个返回一个唯一整数的函数，不管在哪里调用函数都会返回这个整数。而函数不能两次返回同一个值，为了做到这一点，函数必须能够跟踪它每次返回的值，而且这些值的信息需要在不同的函数调过程中持久化。可以将这些信息存放到全局变量中，但这并不是必需的，因为这个信息仅仅是函数本身用到的。最好将这个信息保存到函数对象的一个属性中，下面这个例子就实现了这样一个函数，每次调用函数都会返回一个唯一的整数：

```
// 初始化函数对象的计数器属性
// 由于函数声明被提前了，因此这里是可以在函数声明
// 之前给它的成员赋值的
uniqueInteger.counter = 0;

// 每次调用这个函数都会返回一个不同的整数
// 它使用一个属性来记住下一次将要返回的值
function uniqueInteger() {
    return uniqueInteger.counter++; // 先返回计数器的值，然后计数器自增1
}
```

来看另外一个例子，下面这个函数factorial()使用了自身的属性（将自身当做数组来对待）来缓存上一次的计算结果：

```
// 计算阶乘，并将结果缓存至函数的属性中
function factorial(n) {
    if (isFinite(n) && n>0 && n==Math.round(n)) {       // 有限的正整数
        if (!(n in factorial))                          // 如果没有缓存结果
            factorial[n] = n * factorial(n-1);          // 计算结果并缓存之
        return factorial[n];                            // 返回缓存结果
    }
    else return NaN; // 如果输入有误
}
factorial[1] = 1; // 初始化缓存以保存这种基本情况
```

译注5：通常我们认为的排序都是按照值从小到大，实际上排序参照的维度不同排序结果也不尽相同。

8.5 作为命名空间的函数

3.10.1节介绍了JavaScript中的函数作用域的概念：在函数中声明的变量在整个函数体内都是可见的（包括在嵌套的函数中），在函数的外部是不可见的。不在任何函数内声明的变量是全局变量，在整个JavaScript程序中都是可见的。在JavaScript中是无法声明只在一个代码块内可见的变量的[译注6]，基于这个原因，我们常常简单地定义一个函数用做临时的命名空间，在这个命名空间内定义的变量都不会污染到全局命名空间。

比如，假设你写了一段JavaScript模块代码，这段代码将要用在不同的JavaScript程序中（对于客户端JavaScript来讲通常是用在各种各样的网页中）。和大多数代码一样，假定这段代码定义了一个用以存储中间计算结果的变量。这样问题就来了，当模块代码放到不同的程序中运行时，你无法得知这个变量是否已经创建了，如果已经存在这个变量，那么将会和代码发生冲突。解决办法当然是将代码放入一个函数内，然后调用这个函数。这样全局变量就变成了函数内的局部变量：

```
function mymodule() {
    // 模块代码
    // 这个模块所使用的所有变量都是局部变量
    // 而不是污染全局命名空间
}
mymodule(); //不要忘了还要调用这个函数
```

这段代码仅仅定义了一个单独的全局变量：名叫“mymodule”的函数。这样还是太麻烦，可以直接定义一个匿名函数，并在单个表达式中调用它：

```
(function() {          // mymodule()函数重写为匿名的函数表达式
    // 模块代码
}());                  // 结束函数定义并立即调用它
```

这种定义匿名函数并立即在单个表达式中调用它的写法非常常见，已经成为一种惯用法了。注意上面代码的圆括号的用法，function之前的左圆括号是必需的，因为如果不写这个左圆括号，JavaScript解释器会试图将关键字function解析为函数声明语句。使用圆括号JavaScript解释器才会正确地将其解析为函数定义表达式。使用圆括号是习惯用法，尽管有些时候没有必要也不应当省略。这里定义的函数会立即调用。

例8-3展示了这种命名空间技术。它定义一个返回extend()函数的匿名函数，正如在例6-2中所展示的那样，匿名函数中的代码检测了是否出现了一个众所周知的IE bug，如果出现了这个bug，就返回一个带补丁的函数版本。此外，这个匿名函数命名空间用来隐藏一组属性名。

译注6：在客户端JavaScript中这种说法不完全正确，比如，在有些JavaScript的扩展中就可以使用let来声明语句块内的变量，详细内容请参照第11章。

例8-3：特定场景下返回带补丁的extend()版本

```
// 定义一个扩展函数，用来将第二个以及后续参数复制至第一个参数
// 这里我们处理了IE bug：在多数IE版本中
// 如果o的属性拥有一个不可枚举的同名属性，则for/in循环
// 不会枚举对象o的可枚举属性，也就是说，将不会正确地处理诸如toString的属性
// 除非我们显式检测它
var extend = (function() { //将这个函数的返回值赋值给extend
    // 在修复它之前，首先检查是否存在bug
    for (var p in { toString: null}) {
        // 如果代码执行到这里，那么for/in循环会正确工作并返回
        // 一个简单版本的extend()函数
        return function extend(o) {
            for (var i = 1; i < arguments.length; i++) {
                var source = arguments[i];
                for (var prop in source) o[prop] = source[prop];
            }
            return o;
        };
    }
    // 如果代码执行到这里，说明for/in循环不会枚举测试对象的toString属性
    // 因此返回另一个版本的extend()函数，这个函数显式测试
    // Object.prototype中的不可枚举属性
    return function patched_extend(o) {
        for (var i = 1; i < arguments.length; i++) {
            var source = arguments[i];
            //复制所有的可枚举属性
            for (var prop in source) o[prop] = source[prop];

            // 现在检查特殊属性
            for (var j = 0; j < protoprops.length; j++) {
                prop = protoprops[j];
                if (source.hasOwnProperty(prop)) o[prop] = source[prop];
            }
        }
        return o;
    };

    // 这个列表列出了需要检查的特殊属性
    var protoprops = ["toString", "valueOf", "constructor", "hasOwnProperty",
                      "isPrototypeOf", "propertyIsEnumerable", "toLocaleString"];
    } ());
```

8.6 闭包

和其他大多数现代编程语言一样，JavaScript也采用词法作用域（lexical scoping），也就是说，函数的执行依赖于变量作用域，这个作用域是在函数定义时决定的，而不是函数调用时决定的。为了实现这种词法作用域，JavaScript函数对象的内部状态不仅包含函数的代码逻辑，还必须引用当前的作用域链（在继续阅读后续的章节之前，应当复习一下3.10节和3.10.3节中讲到的变量作用域和作用域链的概念）。函数对象可以通过作用域

链相互关联起来，函数体内部的变量都可以保存在函数作用域内，这种特性在计算机科学文献中称为“闭包”[注4]。

从技术的角度讲，所有的JavaScript函数都是闭包：它们都是对象，它们都关联到作用域链。定义大多数函数时的作用域链在调用函数时依然有效，但这并不影响闭包。当调用函数时闭包所指向的作用域链和定义函数时的作用域链不是同一个作用域链时，事情就变得非常微妙。当一个函数嵌套了另外一个函数，外部函数将嵌套的函数对象作为返回值返回的时候往往会发生这种事情。有很多强大的编程技术都利用到了这类嵌套的函数闭包，以至于这种编程模式在JavaScript中非常常见。当你第一次碰到闭包时可能会觉得非常让人费解，一旦你理解掌握了闭包之后，就能非常自如地使用它了，了解这一点至关重要。

理解闭包首先要了解嵌套函数的词法作用域规则。看一下这段代码（这段代码和你刚在3.10节中看到的代码非常类似）：

```
var scope = "global scope";             // 全局变量
function checkscope() {
    var scope = "local scope";          // 局部变量
    function f() { return scope; }      // 在作用域中返回这个值
    return f();
}
checkscope()                            // => "local scope"
```

checkscope()函数声明了一个局部变量，并定义了一个函数f()，函数f()返回了这个变量的值，最后将函数f()的执行结果返回。你应当非常清楚为什么调用checkscope()会返回“local scope”。现在我们对这段代码做一点改动。你知道这段代码返回什么吗？

```
var scope = "global scope";             // 全局变量
function checkscope() {
    var scope = "local scope";          // 局部变量
    function f() { return scope; }      // 在作用域中返回这个值
    return f;
}
checkscope()()                          // 返回值是什么?
```

在这段代码中，我们将函数内的一对圆括号移动到了checkscope()之后。checkscope()现在仅仅返回函数内嵌套的一个函数对象，而不是直接返回结果。在定义函数的作用域外面，调用这个嵌套的函数（包含最后一行代码的最后一对圆括号）会发生什么事情呢？

回想一下词法作用域的基本规则：JavaScript函数的执行用到了作用域链，这个作用域链

注4：这个术语非常古老，是指函数变量可以被隐藏于作用域链之内，因此看起来是函数将变量“包裹”了起来。

是函数定义的时候创建的。嵌套的函数f()定义在这个作用域链里，其中的变量scope一定是局部变量，不管在何时何地执行函数f()，这种绑定在执行f()时依然有效。因此最后一行代码返回“local scope”，而不是“global scope”。简言之，闭包的这个特性强大到让人吃惊：它们可以捕捉到局部变量（和参数），并一直保存下来，看起来像这些变量绑定到了在其中定义它们的外部函数。

实现闭包

如果你理解了词法作用域的规则，你就能很容易地理解闭包：函数定义时的作用域链到函数执行时依然有效。然而很多程序员觉得闭包非常难理解，因为他们在深入学习闭包的实现细节时将自己搞得晕头转向。他们觉得在外部函数中定义的局部变量在函数返回后就不存在了[译注7]，那么嵌套的函数如何能调用不存在的作用域链呢？如果你想搞清楚这个问题，你需要更深入地了解类似C语言这种更底层的编程语言，并了解基于栈的CPU架构：如果一个函数的局部变量定义在CPU的栈中，那么当函数返回时它们的确就不存在了。

但回想一下在3.10.3节中是如何定义作用域链的。我们将作用域链描述为一个对象列表，不是绑定的栈。每次调用JavaScript函数的时候，都会为之创建一个新的对象用来保存局部变量，把这个对象添加至作用域链中。当函数返回的时候，就从作用域链中将这个绑定变量的对象删除。如果不存在嵌套的函数，也没有其他引用指向这个绑定对象，它就会被当做垃圾回收掉。如果定义了嵌套的函数，每个嵌套的函数都各自对应一个作用域链，并且这个作用域链指向一个变量绑定对象。但如果这些嵌套的函数对象在外部函数中保存下来，那么它们也会和所指向的变量绑定对象一样当做垃圾回收。但是如果这个函数定义了嵌套的函数，并将它作为返回值返回或者存储在某处的属性里，这时就会有一个外部引用指向这个嵌套的函数。它就不会被当做垃圾回收，并且它所指向的变量绑定对象也不会被当做垃圾回收[译注8]。

在8.4.1节中定义了uniqueInteger()函数，这个函数使用自身的一个属性来保存每次返回的值，以便每次调用都能跟踪上次的返回值。但这种做法有一个问题，就是恶意代码可能将计数器重置或者把一个非整数赋值给它，导致uniquenterger()函数不一定能产生“唯一”的“整数”。而闭包可以捕捉到单个函数调用的局部变量，并将这些局部变量用做私有状态。我们可以利用闭包这样来重写uniqueInteger()函数：

译注7：之所以有这种想法是因为很多人以为函数执行结束后，与之相关的作用域链似乎也不存在了，但在JavaScript中并非如此。

译注8：作者在这里清楚地解释了闭包和垃圾回收之间的关系，如果使用不慎，闭包很容易造成“循环引用”，当DOM对象和JavaScript对象之间存在循环引用时需要格外小心，在某些浏览器下会造成内存泄漏。

```
var uniqueInteger = (function() {              // 定义函数并立即调用
                        var counter = 0;       // 函数的私有状态
                        return function() { return counter++; };
                    }());
```

你需要仔细阅读这段代码才能理解其含义。粗略来看，第一行代码看起来像将函数赋值给一个变量uniqueInteger，实际上，这段代码定义了一个立即调用的函数（函数的开始带有左圆括号），因此是这个函数的返回值赋值给变量uniqueInteger。现在，我们来看函数体，这个函数返回另外一个函数，这是一个嵌套的函数，我们将它赋值给变量uniqueInteger，嵌套的函数是可以访问作用域内的变量的，而且可以访问外部函数中定义的counter变量。当外部函数返回之后，其他任何代码都无法访问counter变量，只有内部的函数才能访问到它。

像counter一样的私有变量不是只能用在一个单独的闭包内，在同一个外部函数内定义的多个嵌套函数也可以访问它，这多个嵌套函数都共享一个作用域链，看一下这段代码：

```
function counter() {
    var n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}

var c = counter(), d = counter();      // 创建两个计数器
c.count()                              // => 0
d.count()                              // => 0: 它们互不干扰
c.reset()                              // reset() 和 count() 方法共享状态
c.count()                              // => 0: 因为我们重置了c
d.count()                              // => 1: 而没有重置d
```

counter()函数返回了一个“计数器”对象，这个对象包含两个方法：count()返回下一个整数，reset()将计数器重置为内部状态。首先要理解，这两个方法都可以访问私有变量n。再者，每次调用counter()都会创建一个新的作用域链和一个新的私有变量。因此，如果调用counter()两次，则会得到两个计数器对象，而且彼此包含不同的私有变量，调用其中一个计数器对象的count()或reset()不会影响到另外一个对象。

从技术角度看，其实可以将这个闭包合并为属性存取器方法getter和setter。下面这段代码所示的counter()函数的版本是6.6节中代码的变种，所不同的是，这里私有状态的实现是利用了闭包，而不是利用普通的对象属性来实现：

```
function counter(n) { // 函数参数n是一个私有变量
    return {
        // 属性getter方法返回并给私有计数器var递增1
        get count() { return n++; },
```

```
        // 属性setter不允许n递减
        set count(m) {
        if (m >= n) n = m;
        else throw Error("count can only be set to a larger value");
        }
    };
}

var c = counter(1000);
c.count              // => 1000
c.count              // => 1001
c.count = 2000
c.count              // => 2000
c.count = 2000       // => Error!
```

需要注意的是，这个版本的counter()函数并未声明局部变量，而只是使用参数n来保存私有状态，属性存取器方法可以访问n。这样的话，调用counter()的函数就可以指定私有变量的初始值了。

例8-4是这种使用闭包技术来共享的私有状态的通用做法。这个例子定义了addPrivateProperty()函数，这个函数定义了一个私有变量，以及两个嵌套的函数用来获取和设置这个私有变量的值。它将这些嵌套函数添加为所指定对象的方法：

例8-4：利用闭包实现的私有属性存取器方法

```
// 这个函数给对象o增加了属性存取器方法
// 方法名称为get<name>和set<name>。如果提供了一个判定函数
// setter方法就会用它来检测参数的合法性，然后在存储它
// 如果判定函数返回false，setter方法抛出一个异常
//
// 这个函数有一个非同寻常之处，就是getter和setter函数
// 所操作的属性值并没有存储在对象o中
// 相反，这个值仅仅是保存在函数中的局部变量中
// getter和setter方法同样是局部函数，因此可以访问这个局部变量
// 也就是说，对于两个存取器方法来说这个变量是私有的
// 没有办法绕过存取器方法来设置或修改这个值
function addPrivateProperty(o, name, predicate) {
    var value; // 这是一个属性值

    // getter方法简单地将其返回
    o["get" + name] = function() { return value; };

    // setter方法首先检查值是否合法，若不合法就抛出异常
    // 否则就将其存储起来
    o["set" + name] = function(v) {
        if (predicate && !predicate(v))
            throw Error("set" + name + ": invalid value " + v);
        else
            value = v;
    };
}

// 下面的代码展示了addPrivateProperty()方法
```

```
var o = {}; // 设置一个空对象

// 增加属性存取器方法getName()和setName()
// 确保只允许字符串值
addPrivateProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank");       // 设置属性值
console.log(o.getName()); // 得到属性值
o.setName(0);             // 试图设置一个错误类型的值
```

我们已经给出了很多例子，在同一个作用域链中定义两个闭包，这两个闭包共享同样的私有变量或变量。这是一种非常重要的技术，但还是要特别小心那些不希望共享的变量往往不经意间共享给了其他的闭包，了解这一点也很重要。看一下下面这段代码：

```
// 这个函数返回一个总是返回v的函数
function constfunc(v) { return function() { return v; }; }

// 创建一个数组用来存储常数函数
var funcs = [];
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// 在第5个位置的元素所表示的函数返回值为5
funcs[5]() // => 5
```

这段代码利用循环创建了很多个闭包，当写类似这种代码的时候往往会犯一个错误：那就是试图将循环代码移入定义这个闭包的函数之内，看一下这段代码：

```
// 返回一个函数组成的数组，它们的返回值是0～9
function constfuncs() {
    var funcs = [];
    for(var i = 0; i < 10; i++)
        funcs[i] = function() { return i; };
    return funcs;
}

var funcs = constfuncs();
funcs[5]() // 返回值是什么?
```

上面这段代码创建了10个闭包，并将它们存储到一个数组中。这些闭包都是在同一个函数调用中定义的，因此它们可以共享变量i。当`constfuncs()`返回时，变量i的值是10，所有的闭包都共享这一个值，因此，数组中的函数的返回值都是同一个值，这不是我们想要的结果。关联到闭包的作用域链都是“活动的”，记住这一点非常重要。嵌套的函数不会将作用域内的私有成员复制一份，也不会对所绑定的变量生成静态快照（static snapshot）。

书写闭包的时候还需注意一件事情，this是JavaScript的关键字，而不是变量。正如之前讨

论的，每个函数调用都包含一个this值，如果闭包在外部函数里是无法访问this的[译注9]，除非外部函数将this转存为一个变量：

```
var self = this; // 将this保存至一个变量中，以便嵌套的函数能够访问它
```

绑定arguments的问题与之类似。arguments并不是一个关键字，但在调用每个函数时都会自动声明它，由于闭包具有自己所绑定的arguments，因此闭包内无法直接访问外部函数的参数数组，除非外部函数将参数数组保存到另外一个变量中：

```
var outerArguments = arguments;  //保存起来以便嵌套的函数能使用它
```

在本章接下来讲到的例8-5中就利用了这种编程技巧来定义闭包，以便在闭包中可以访问外部函数的this和arguments值。

8.7 函数属性、方法和构造函数

我们看到在JavaScript程序中，函数是值。对函数执行typeof运算会返回字符串“function”，但是函数是JavaScript中特殊的对象。因为函数也是对象，它们也可以拥有属性和方法，就像普通的对象可以拥有属性和方法一样。甚至可以用Function()构造函数来创建新的函数对象。接下来几节就会着重介绍函数属性和方法以及Function()构造函数。在第三部分也有关于这些内容的讲解。

8.7.1 length属性

在函数体里，arguments.length表示传入函数的实参的个数。而函数本身的length属性则有着不同含义。函数的length属性是只读属性，它代表函数实参的数量，这里的参数指的是“形参”而非“实参”，也就是在函数定义时给出的实参个数，通常也是在函数调用时期望传入函数的实参个数。

下面的代码定义一个名叫check()的函数，从另外一个函数给它传入arguments数组，它比较arguments.length（实际传入的实参个数）和arguments.callee.length（期望传入的实参个数）来判断所传入的实参个数是否正确。如果个数不正确，则抛出异常。check()函数之后定义一个测试函数f()，用来展示check()的用法：

译注9：严格讲，闭包内的逻辑是可以使用this的，但这个this和当初定义函数时的this不是同一个，即便是同一个this，this的值是随着调用栈的变化而变化的，而闭包里的逻辑所取到的this的值也是不确定的，因此外部函数内的闭包是可以使用this的，但要非常小心地使用才行，作者在这里提到的将this转存为一个变量的做法就可以避免this的不确定性带来的歧义。

```
// 这个函数使用arguments.callee，因此它不能在严格模式下工作
function check(args) {
    var actual = args.length;                  //实参的真实个数
    var expected = args.callee.length;         //期望的实参个数
    if (actual !== expected)                   //如果不同则抛出异常
        throw Error("Expected " + expected + "args; got " + actual);
}

function f(x, y, z) {
    check(arguments);          // 检查实参个数和期望的实参个数是否一致
    return x + y + z;          // 再执行函数的后续逻辑
}
```

8.7.2 prototype属性

每一个函数都包含一个prototype属性，这个属性是指向一个对象的引用，这个对象称做“原型对象”（prototype object）。每一个函数都包含不同的原型对象。当将函数用做构造函数的时候，新创建的对象会从原型对象上继承属性。6.1.3节讨论了原型和prototype属性，在第9章里会有进一步讨论。

8.7.3 call()方法和apply()方法

我们可以将call()和apply()看做是某个对象的方法，通过调用方法的形式来间接调用（见8.2.4节）函数（比如在例6-4我们使用了call()方法来调用一个对象的Object.prototype.toString方法，用以输出对象的类）。call()和apply()的第一个实参是要调用函数的母对象，它是调用上下文，在函数体内通过this来获得对它的引用。要想以对象o的方法来调用函数f()，可以这样使用call()和apply()：

```
f.call(o);
f.apply(o);
```

每行代码和下面代码的功能类似（假设对象o中预先不存在名为m的属性）。

```
o.m = f;      // 将f存储为o的临时方法
o.m();        // 调用它，不传入参数
delete o.m;   //将临时方法删除
```

在ECMAScript 5的严格模式中，call()和apply()的第一个实参都会变为this的值，哪怕传入的实参是原始值甚至是null或undefined。在ECMAScript 3和非严格模式中，传入的null和undefined都会被全局对象代替，而其他原始值则会被相应的包装对象（wrapper object）所替代。

对于call()来说，第一个调用上下文实参之后的所有实参就是要传入待调用函数的值。比如，以对象o的方法的形式调用函数f()，并传入两个参数，可以使用这样的代码：

```
f.call(o, 1, 2);
```

apply()方法和call()类似，但传入实参的形式和call()有所不同，它的实参都放入一个数组当中：

```
f.apply(o,[1,2]);
```

如果一个函数的实参可以是任意数量，给apply()传入的参数数组可以是任意长度的。比如，为了找出数组中最大的数值元素，调用Math.max()方法的时候可以给apply()传入一个包含任意个元素的数组：

```
var biggest = Math.max.apply(Math, array_of_numbers);
```

需要注意的是，传入apply()的参数数组可以是类数组对象也可以是真实数组。实际上，可以将当前函数的arguments数组直接传入（另一个函数的）apply()来调用另一个函数，参照如下代码：

```
// 将对象o中名为m()的方法替换为另一个方法
// 可以在调用原始的方法之前和之后记录日志消息
function trace(o, m) {
    var original = o[m];       //在闭包中保存原始方法
    o[m] = function() {        // 定义新的方法
        console.log(new Date(), "Entering:", m);         //输出日志消息
        var result = original.apply(this, arguments);    // 调用原始函数
        console.log(new Date(), "Exiting:", m);          //输出日志消息
        return result;                                   // 返回结果
    };
}
```

trace()函数接收两个参数，一个对象和一个方法名，它将指定的方法替换为一个新方法，这个新方法是“包裹”原始方法的另一个泛函数[译注10]。这种动态修改已有方法的做法有时称做“monkey-patching”。

8.7.4 bind()方法

bind()是在ECMAScript 5中新增的方法，但在ECMAScript 3中可以轻易模拟bind()。从名字就可以看出，这个方法的主要作用就是将函数绑定至某个对象。当在函数f()上调用bind()方法并传入一个对象o作为参数，这个方法将返回一个新的函数。（以函数调用的方式）调用新的函数将会把原始的函数f()当做o的方法来调用。传入新函数的任何实参都将传入原始函数，比如：

译注10：泛函数也叫泛函，在这里特指一种变换，以函数为输入，输出可以是值也可以是另一个函数，泛函的概念可以参照：*http://zh.wikipedia.org/wiki/泛函数*。

```
function f(y) { return this.x + y; }   //这个是待绑定的函数
var o = { x : 1 };                     //将要绑定的对象
var g = f.bind(o);                     // 通过调用g(x)来调用o.f(x)
g(2) // => 3
```

可以通过如下代码轻易地实现这种绑定：

```
// 返回一个函数，通过调用它来调用o中的方法f()，传递它所有的实参
function bind(f, o) {
    if (f.bind) return f.bind(o);      // 如果bind()方法存在的话，使用bind()方法
    else return function() {           //否则，这样绑定
        return f.apply(o, arguments);
    };
}
```

ECMAScript 5中的bind()方法不仅仅是将函数绑定至一个对象，它还附带一些其他应用：除了第一个实参之外，传入bind()的实参也会绑定至this，这个附带的应用是一种常见的函数式编程技术，有时也被称为“柯里化”（currying）。参照下面这个例子中的bind()方法的实现：

```
var sum = function(x,y) { return x + y }; //返回两个实参的和值
// 创建一个类似sum的新函数，但this的值绑定到null
// 并且第一个参数绑定到1，这个新的函数期望只传入一个实参
var succ = sum.bind(null, 1);
succ(2) // => 3: x绑定到1，并传入2作为实参y

function f(y,z) { return this.x + y + z };      //另外一个做累加计算的函数
var g = f.bind({x:1}, 2);                       //绑定this和y
g(3)    // => 6: this.x绑定到1，y绑定到2，z绑定到3
```

我们可以绑定this的值并在ECMAScript 3中实现这个附带的应用。例8-5中的示例代码就模拟实现了标准的bind()方法。

注意，我们将这个方法另存为Function.prototype.bind，以便所有的函数对象都继承它，这种技术在9.4节中有详细介绍：

例8-5：ECMAScript 3版本的Function.bind()方法

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, args */) {
        // 将this和arguments的值保存至变量中
        // 以便在后面嵌套的函数中可以使用它们
        var self = this, boundArgs = arguments;

        // bind()方法的返回值是一个函数
        return function() {
            // 创建一个实参列表，将传入bind()的第二个及后续的实参都传入这个函数
            var args = [], i;
            for(i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]);
            for(i = 0; i < arguments.length; i++) args.push(arguments[i]);
```

```
            // 现在将self作为o的方法来调用，传入这些实参
            return self.apply(o, args);
        };
    };
}
```

我们注意到，bind()方法返回的函数是一个闭包，在这个闭包的外部函数中声明了self和boundArgs变量，这两个变量在闭包里用到。尽管定义闭包的内部函数已经从外部函数中返回，而且调用这个闭包逻辑的时刻要在外部函数返回之后（在闭包中照样可以正确访问这两个变量）。

ECMAScript 5定义的bind()方法也有一些特性是上述ECMAScript 3代码无法模拟的。首先，真正的bind()方法返回一个函数对象，这个函数对象的length属性是绑定函数的形参个数减去绑定实参的个数（length的值不能小于零）。再者，ECMAScript 5的bind()方法可以顺带用做构造函数。如果bind()返回的函数用做构造函数，将忽略传入bind()的this，原始函数就会以构造函数的形式调用，其实参也已经绑定[译注11]。由bind()方法所返回的函数并不包含prototype属性（普通函数固有的prototype属性是不能删除的），并且将这些绑定的函数用做构造函数时所创建的对象从原始的未绑定的构造函数中继承prototype。同样，在使用instanceof运算符时，绑定构造函数和未绑定构造函数并无两样。

8.7.5 toString()方法

和所有的JavaScript对象一样，函数也有toString()方法，ECMAScript规范规定这个方法返回一个字符串，这个字符串和函数声明语句的语法相关。实际上，大多数（非全部）的toString()方法的实现都返回函数的完整源码。内置函数往往返回一个类似“[native code]”的字符串作为函数体。

8.7.6 Function()构造函数

不管是通过函数定义语句还是函数直接量表达式，函数的定义都要使用function关键字。但函数还可以通过Function()构造函数来定义，比如：

```
var f = new Function("x", "y", "return x*y;");
```

这一行代码创建一个新的函数，这个函数和通过下面代码定义的函数几乎等价：

```
var f = function(x, y) { return x*y; }
```

译注11：作者的意思是在运行时将bind()所返回的函数用做构造函数时，所传入实参会原封不动的传入原始函数。

Function()构造函数可以传入任意数量的字符串实参，最后一个实参所表示的文本就是函数体；它可以包含任意的JavaScript语句，每两条语句之间用分号分隔。传入构造函数的其他所有的实参字符串是指定函数的形参名字的字符串。如果定义的函数不包含任何参数，只须给构造函数简单地传入一个字符串——函数体——即可。

注意，Function()构造函数并不需要通过传入实参以指定函数名。就像函数直接量一样，Function()构造函数创建一个匿名函数。

关于Function()构造函数有几点需要特别注意：

- Function()构造函数允许JavaScript在运行时动态地创建并编译函数。
- 每次调用Function()构造函数都会解析函数体，并创建新的函数对象。如果是在一个循环或者多次调用的函数中执行这个构造函数，执行效率会受影响。相比之下，循环中的嵌套函数和函数定义表达式则不会每次执行时都重新编译。
- 最后一点，也是关于Function()构造函数非常重要的一点，就是它所创建的函数并不是使用词法作用域，相反，函数体代码的编译总是会在顶层函数[译注12]执行，正如下面代码所示：

```
var scope = "global";
function constructFunction() {
    var scope = "local";
    return new Function("return scope"); //无法捕获局部作用域
}
// 这一行代码返回global，因为通过Function()构造函数
// 所返回的函数使用的不是局部作用域
constructFunction()(); // => "global"
```

我们可以将Function()构造函数认为是在全局作用域中执行的eval()（参照4.12.2节），eval()可以在自己的私有作用域内定义新变量和函数，Function()构造函数在实际编程过程中很少会用到。

8.7.7 可调用的对象

我们在7.11节中提到“类数组对象”并不是真正的数组，但大部分场景下可以将其当做数组来对待。对于函数也存在类似的情况。“可调用的对象”（callable object）是一个对象，可以在函数调用表达式中调用这个对象。所有的函数都是可调用的，但并非所有的可调用对象都是函数。

截至目前，可调用对象在两个JavaScript实现中不能算作函数。首先，IE Web浏览器（IE8及之前的版本）实现了客户端方法（诸如Window.alert()和Document.

译注12：也就是全局作用域。

getElementsById()) [译注13]，使用了可调用的宿主对象，而不是内置函数对象。IE中的这些方法在其他浏览器中也都存在，但它们本质上不是Function对象。IE9将它们实现为真正的函数，因此这类可调用的对象将越来越罕见。

另外一个常见的可调用对象是RegExp对象（在众多浏览器中均有实现），可以直接调用RegExp对象，这比调用它的exec()方法更快捷一些。在JavaScript中这是一个彻头彻尾的非标准特性，最开始是由Netscape提出，后被其他浏览器厂商所复制，仅仅是为了和Netscape兼容。代码最好不要对可调用的RegExp对象有太多依赖，这个特性在不久的将来可能会废弃并删除。对RegExp执行typeof运算的结果并不统一，在有些浏览器中返回“function”，在有些中返回“object”。

如果想检测一个对象是否是真正的函数对象（并且具有函数方法），可以参照例6-4中的代码检测它的*class*属性（见6.8.2节）：

```
function isFunction(x) {
    return Object.prototype.toString.call(x) === "[object Function]";
}
```

注意，这里的isFunction()函数和7.10节的isArray()函数极其类似。

8.8 函数式编程

和Lisp、Haskell不同，JavaScript并非函数式编程语言，但在JavaScript中可以像操控对象一样操控函数，也就是说可以在JavaScript中应用函数式编程技术。ECMAScript 5中的数组方法（诸如map()和reduce()）就可以非常适合用于函数式编程风格。接下来的几节将会着重介绍JavaScript中的函数式编程技术。对JavaScript函数的探讨会让人倍感兴奋，你会体会到JavaScript函数非常强大，而不仅仅是学习一种编程风格而已[注5]。

8.8.1 使用函数处理数组

假设有一个数组，数组元素都是数字，我们想要计算这些元素的平均值和标准差。若使用非函数式编程风格的话，代码会是这样：

```
var data = [1,1,3,5,5]; // 这里是待处理的数组

// 平均数是所有元素的累加和值除以元素个数
var total = 0;
```

译注13：作者给出的这个例子有误，应当是getElementById()。

注5：如果你对这部分内容感兴趣，推荐你使用一下（至少阅读一下）奥利弗·斯蒂尔（Oliver Steele）的函数式JavaScript库，请参照：*http://osteele.com/sources/javascript/functional/*。

```
for(var i = 0; i < data.length; i++) total += data[i];
var mean = total/data.length;// 平均数是3

//计算标准差，首先计算每个数据减去平均数之后偏差的平方然后求和
total = 0;
for(var i = 0; i < data.length; i++) {
    var deviation = data[i] - mean;
    total += deviation * deviation;
}
var stddev = Math.sqrt(total/(data.length-1)); // 标准差的值是 2
```

可以使用数组方法map()和reduce()来实现同样的计算，这种实现极其简洁（参照7.9节来查看这些方法）：

```
//首先定义两个简单的函数
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };

// 然后将这些函数和数组方法配合使用计算出平均数和标准差
var data = [1,1,3,5,5];
var mean = data.reduce(sum)/data.length;
var deviations = data.map(function(x) {return x-mean;});
var stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
```

如果我们基于ECMAScript 3来如何实现呢？因为ECMAScript 3中并不包含这些数组方法，如果不存在内置方法的话我们可以自定义map()和reduce()函数：

```
// 对于每个数组元素调用函数f()，并返回一个结果数组
// 如果Array.prototype.map定义了的话，就使用这个方法
var map = Array.prototype.map
    ? function(a, f) { return a.map(f); }      // 如果已经存在map()方法，就直接使用它
    : function(a, f) {                         // 否则，自己实现一个
        var results = [];
        for (var i = 0, len = a.length; i < len; i++) {
            if (i in a) results[i] = f.call(null, a[i], i, a);
        }
        return results;
    };

// 使用函数f()和可选的初始值将数组a减至一个值
// 如果Array.prototype.reduce存在的话，就使用这个方法
var reduce = Array.prototype.reduce
    ? function(a, f, initial) {        //如果reduce()方法存在的话
    if (arguments.length > 2)
        return a.reduce(f, initial);   // 如果传入了一个初始值
        else return a.reduce(f); // 否则没有初始值
    }
    : function(a, f, initial) { // 这个算法来自ES5规范
        var i = 0, len = a.length, accumulator;

        // 以特定的初始值开始，否则第一个值取自a
        if (arguments.length > 2) accumulator = initial;
```

```
        else { //找到数组中第一个已定义的索引
             if (len == 0) throw TypeError();
             while (i < len) {
                 if (i in a) {
                     accumulator = a[i++];
                     break;
                 }
                 else i++;
             }
             if (i == len) throw TypeError();
        }

        // 对于数组中剩下的元素依次调用f()
        while (i < len) {
             if (i in a)
                 accumulator = f.call(undefined, accumulator, a[i], i, a);
             i++;
        }

        return accumulator;
    };
```

使用定义的map()和reduce()函数，计算平均值和标准差的代码看起来像这样：

```
var data = [1,1,3,5,5];
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };
var mean = reduce(data, sum)/data.length;
var deviations = map(data, function(x) {return x-mean;});
var stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1));
```

8.8.2 高阶函数

所谓高阶函数（higher-order function）就是操作函数的函数，它接收一个或多个函数作为参数，并返回一个新函数，来看这个例子：

```
// 这个高阶函数返回一个新的函数，这个新函数将它的实参传入f()
// 并返回f的返回值的逻辑非
function not(f) {
    return function() {                          //返回一个新的函数
        var result = f.apply(this, arguments);  // 调用f()
        return !result;                          // 对结果求反
    };
}

var even = function(x) { // 判断a是否为偶数的函数
    return x % 2 === 0;
};

var odd = not(even);           //一个新函数，所做的事情和even()相反
[1, 1, 3, 5, 5].every(odd);    // => true: 每个元素都是奇数
```

上面的not()函数就是一个高阶函数，因为它接收一个函数作为参数，并返回一个新函数。另外一个例子，来看下面的mapper()函数，它也是接收一个函数作为参数，并返回一个新函数，这个新函数将一个数组映射到另一个使用这个函数的数组上。这个函数使用了之前定义的map()函数，但要首先理解这两个函数有哪里不同，理解这一点至关重要：

```
// 所返回的函数的参数应当是一个实参数组，并对每个数组元素执行函数f()
// 并返回所有计算结果组成的数组
// 可以对比一下这个函数和上文提到的map()函数
function mapper(f) {
    return function(a) { return map(a, f); };
}

var increment = function(x) { return x+1; };
var incrementer = mapper(increment);
incrementer([1,2,3]) // => [2,3,4]
```

这里是一个更常见的例子，它接收两个函数f()和g()，并返回一个新的函数用以计算f(g())：

```
// 返回一个新的可以计算f(g(...))的函数
// 返回的函数h()将它所有的实参传入g()，然后将g()的返回值传入f()
// 调用f()和g()时的this值和调用h()时的this值是同一个this
function compose(f,g) {
    return function() {
        // 需要给f()传入一个参数，所以使用f()的call()方法
        // 需要给g()传入很多参数，所以使用g()的apply()方法
        return f.call(this, g.apply(this, arguments));
    };
}

var square = function(x) { return x*x; };
var sum = function(x,y) { return x+y; };
var squareofsum = compose(square, sum);
squareofsum(2,3) // => 25
```

本章后续几节中定义了partial()和memoize()函数，这两个函数是非常重要的高阶函数。

8.8.3 不完全函数

函数f()（见8.7.4节）的bind()方法返回一个新函数，给新函数传入特定的上下文和一组指定的参数，然后调用函数f()。我们说它把函数“绑定至”对象并传入一部分参数。bind()方法只是将实参放在（完整实参列表的）左侧[译注14]，也就是说传入bind()的实参

译注14：作者在本节讨论的是一种函数变换技巧，即把一次完整的函数调用拆成多次函数调用，每次传入的实参都是完整实参的一部分，每个拆分开的函数叫做不完全函数（partial function），每次函数调用叫做不完全调用（partial application），这种函数变换的特点是每次调用都返回一个函数，直到得到最终运行结果为止，举一个简单的例子，将对函数f(1,2,3,4,5,6)的调用修改为等价的f(1,2)(3,4)(5,6)，后者包含三次调用，和每次调用相关的函数就是“不完全函数”。

都是放在传入原始函数的实参列表开始的位置，但有时我们期望将传入bind()的实参放在（完整实参列表的）右侧：

```
// 实现一个工具函数将类数组对象（或对象）转换为真正的数组
// 在后面的示例代码中用到了这个方法将arguments对象转换为真正的数组
function array(a, n) { return Array.prototype.slice.call(a, n || 0);}

// 这个函数的实参传递至左侧
function partialLeft(f /*, ...*/) {
    var args = arguments;      //保存外部的实参数组
    return function() {        // 并返回这个函数
        var a = array(args, 1);                 // 开始处理外部的第1个args
        a = a.concat(array(arguments));         // 然后增加所有的内部实参
        return f.apply(this, a);                // 然后基于这个实参列表调用f()
    };
}

// 这个函数的实参传递至右侧
function partialRight(f /*, ...*/) {
    var args = arguments;      // 保存外部实参数组
    return function() {        // 返回这个函数
        var a = array(arguments);       //从内部参数开始
        a = a.concat(array(args, 1));  //然后从外部第1个args开始添加
        return f.apply(this, a);        // 最后基于这个实参列表调用f()
    };
}

// 这个函数的实参被用做模板
// 实参列表中的undefined值都被填充
function partial(f /*, ... */) {
    var args = arguments; //保存外部实参数组
    return function() {
        var a = array(args, 1);         //从外部args开始
        var i = 0, j = 0;
        // 遍历args，从内部实参填充undefined值
        for (; i < a.length; i++)
            if (a[i] === undefined) a[i] = arguments[j++];
        // 现在将剩下的内部实参都追加进去
        a = a.concat(array(arguments, j))
        return f.apply(this, a);
    };
}

// 这个函数带有三个实参
var f = function(x, y, z) { return x * (y - z);};
// 注意这三个不完全调用之间的区别
partialLeft(f, 2)(3, 4)        // => -2:绑定第一个实参: 2 * (3 - 4)
partialRight(f, 2)(3, 4)       // => 6: 绑定最后一个实参:3 * (4 - 2)
partial(f, undefined, 2)(3, 4) // => -6: 绑定中间的实参: 3 * (2 - 4)
```

利用这种不完全函数的编程技巧，可以编写一些有意思的代码，利用已有的函数来定义新的函数，参照下面这个例子：

```
var increment = partialLeft(sum, 1);
var cuberoot = partialRight(Math.pow, 1/3);
String.prototype.first = partial(String.prototype.charAt, 0);
String.prototype.last = partial(String.prototype.substr, -1, 1);
```

当将不完全调用和其他高阶函数整合在一起的时候，事情就变得格外有趣了。比如，这里的例子定义了not()函数，它用到了刚才提到的不完全调用：

```
var not = partialLeft(compose, function(x) { return !x; });
var even = function(x) { return x % 2 === 0; };
var odd = not(even);
var isNumber = not(isNaN)
```

我们也可以使用不完全调用的组合来重新组织求平均数和标准差的代码，这种编码风格是非常纯粹的函数式编程：

```
var data = [1, 1, 3, 5, 5];                    // 我们要处理的数据
var sum = function(x, y) { return x + y; };    // 两个初等函数
var product = function(x, y) { return x * y; };
var neg = partial(product, -1);                // 定义其他函数
var square = partial(Math.pow, undefined, 2);
var sqrt = partial(Math.pow, undefined, .5);
var reciprocal = partial(Math.pow, undefined, -1);

// 现在计算平均值和标准差，所有的函数调用都不带运算符
// 这段代码看起来很像lisp代码
var mean = product(reduce(data, sum), reciprocal(data.length));
var stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))),
                                 sum),
                          reciprocal(sum(data.length, -1))));
```

8.8.4 记忆

在8.4.1节中定义了一个阶乘函数，它可以将上次的计算结果缓存起来。在函数式编程当中，这种缓存技巧叫做“记忆”（memorization）。下面的代码展示了一个高阶函数，memorize()接收一个函数作为实参，并返回带有记忆能力的函数[译注15]。

```
// 返回f()的带有记忆功能的版本
// 只有当f()的实参的字符串表示都不相同时它才会工作
function memorize(f) {
    var cache = {}; //将值保存在闭包内
```

译注15：需要注意的是，记忆只是一种编程技巧，本质上是牺牲算法的空间复杂度以换取更优的时间复杂度，在客户端JavaScript中代码的执行时间复杂度往往成为瓶颈，因此在大多数场景下，这种牺牲空间换取时间的做法以提升程序执行效率的做法是非常可取的。

```
        return function() {
          // 将实参转换为字符串形式，并将其用做缓存的键
          var key = arguments.length + Array.prototype.join.call(arguments,",");
          if (key in cache) return cache[key];
          else return cache[key] = f.apply(this, arguments);
        };
    }
```

memorize()函数创建一个新的对象，这个对象被当做缓存（的宿主）并赋值给一个局部变量，因此对于返回的函数来说它是私有的（在闭包中）。所返回的函数将它的实参数组转换成字符串，并将字符串用做缓存对象的属性名。如果在缓存中存在这个值，则直接返回它。

否则，就调用既定的函数对实参进行计算，将计算结果缓存起来并返回，下面的代码展示了如何使用memorize()：

```
// 返回两个整数的最大公约数
// 使用欧几里德算法:http://en.wikipedia.org/wiki/Euclidean_algorithm
function gcd(a,b) {                    // 这里省略对a和b的类型检查
    var t;                             // 临时变量用来存储交换数值
    if (a < b) t=b, b=a, a=t;          // 确保 a >= b
    while(b != 0) t=b, b = a%b, a=t;   // 这是求最大公约数的欧几里德算法
    return a;
}

var gcdmemo = memorize(gcd);
gcdmemo(85, 187) // => 17

// 注意，当我们写一个递归函数时，往往需要实现记忆功能
// 我们更希望调用实现了记忆功能的递归函数，而不是原递归函数
var factorial = memorize(function(n) {
                           return (n <= 1) ? 1 : n * factorial(n-1);
                         });
factorial(5)          // => 120.对于4～1的值也有缓存
```