JavaScript
函数编程基础

随书赠送

目　录

JavaScript 函数编程基础

对于学习 JavaScript 语言来说，掌握函数是非常重要的。JavaScript 语言中的很多功能，及其灵活性和表达能力都来自函数。例如，JavaScript 通过函数来实现面向对象特性的。因此，我们特意为初学者开设一个 JavaScript 函数专题，帮助读者打好 Web 开发的语言基础。

【学习重点】

- 如何定义和使用函数。
- 如何向函数传递参数。
- 调用预定义函数。
- 了解 JavaScript 中的变量作用域。
- 理解函数也是数据的本质。
- 匿名函数的调用。
- 回调函数。
- 即时函数。
- 内嵌函数。

E.1 再说函数

所谓函数，本质上是一种代码的分组形式。我们可以通过这种形式赋予某组代码一个名字，以便于之后的调用。

【示例】下面示例声明一下函数。

```
function sum(a, b) {
    var c = a + b;
    return c;
```

```
}
```

一般来说，函数声明通常由以下几部分组成。

- 关键词 function。
- 函数名称，即这里的 sum。
- 数所需的参数，即这里的 a、b。一个函数通常都具有 0 个或多个参数。参数之间用逗号分隔。
- 函数所要执行的代码块，我们称之为函数体。
- return 子句。函数通常都会有返回值，如果某个函数没有显式的返回值，我们就会默认它的返回值为 undefined。

注意，一个函数只能有一个返回值，如果我们需要同时返回多个值，可以考虑将其放进一个数组里，以数组元素的形式返回。

在 JavaScript 中，函数声明只是创建函数的方法之一，之后还会介绍其他方法。

E.1.1 调用函数

如果要使用一个函数，就必须调用它。调用的方式很简单，只需在函数名后面加一对小括号即可。有时也称之为请求函数。

【示例】以上节示例定义的函数为例，调用 sum()函数。先将两个参数传递给该函数，然后再将函数的返回值赋值给变量 result。

```
var result = sum(1, 2);
> result;          //3
```

E.1.2 函数的参数

在定义一个函数的同时，往往会设置该函数所需的参数。当然，也可以不给它设定参数，但如果设定了，而又在调用时忘了传递相关的参数值，

JavaScript 引擎就会自动将其设定为 undefined。

【示例 1】在下面这个调用中，函数返回的是 NaN，因为这里试图将 1 与 undefined 相加。

```
sum(1);          //NaN
```

从技术角度来说，参数可分为形参（形式参数）与实参（实际参数）两种，一般不需要严格区分它们。形参是指定义函数时所用的那些参数，而实参则指的是在调用函数时所传递的那些参数。例如：

```
function sum(a, b){
    return a + b;
}
sum(1, 2);
```

在这里，a 和 b 是形参，而 1 和 2 是实参。

对于那些已经传递进来的参数，JavaScript 会忽略，不会抛出异常。例如，向 sum()传递再多的参数，多余的那部分也只会被默默地忽略掉。

```
sum(1, 2, 3, 4, 5);          //3
```

实际上，我们可以利用函数内部的 arguments 变量，该变量为内建变量，每个函数中都能调用。它能返回函数所接收的所有参数。例如：

```
function args() {
    return arguments;
}
args();                                  // []
args( 1, 2, 3, 4, true, 'ninja');        // [1, 2, 3, 4, true, "ninja"]
```

【示例 2】通过变量 arguments，可以进一步完善 sum()函数的功能，使之能对任意数量的参数执行求和运算。

```
function sum() {
    var i,
    res = 0,
    number_of_params = arguments.length;
    for (i = 0; i < number_of_params; i++) {
```

```
        res += arguments[i];
    }
    return res;
}
```

下面用不同数量的参数（包括没有参数）来测试该函数，看看它是否能按照预计的方式工作。

```
sum(1, 1, 1);                    //3
sum(1, 2, 3, 4);                 //10
sum(1, 2, 3, 4, 4, 3, 2, 1);     //20
sum(5);                          //5
sum();                           //0
```

其中，表达式 arguments.length 返回的是函数被调用时所接收的参数数量。arguments 实际上不是一个数组（虽然它有很多数组的特性），而是一个类似数组的对象。

E.2 预定义函数

JavaScript 引擎中有一组可供随时调用的内建函数。下面就来了解一下这些函数。

E.2.1 parseInt()

parseInt()会试图将其收到的任何输入值（通常是字符串）转换成整数类型输出。如果转换失败就返回 NaN。例如：

```
parseInt('123');                 // 123
parseInt('abc123');              // NaN
parseInt('1abc23');              //1
```

```
parseInt('123abc');                    //123
```

除此之外，该函数还有个可选的第二参数：基数（radix），它负责设定函数所期望的数字类型：十进制、十六进制、二进制等。

【示例 1】在下面的例子中，如果试图以十进制输出字符串"FF"，结果就会为 NaN。而改为十六进制，就会得到 255。

```
parseInt('FF', 10);                    //NaN
 parseInt('FF', 16);                   //255
```

【示例 2】再来看一个将字符串转换为十进制和八进制的例子。

```
parseInt('0377', 10);                  //377
 parseInt('0377', 8);                  //255
```

如果在调用 parseInt()时没有指定第二参数，函数就会将其默认为十进制，但有两种情况例外。

- 如果首参数字符串是 0x 开头，第二参数就会被默认指定为 16（也就是默认其为十六进制数）。
- 如果首参数以 0 开头，第二参数就会被默认指定为 8（也就是默认其为八进制数）。

```
parseInt('377');                       //377
parseInt('0377');                      //255
parseInt('0x377');                     //887
```

当然，明确指定 radix 值是最安全的。如果省略了它，尽管 99%的情况下依然能够正常运作（毕竟最常用的还是十进制数），但我们偶尔还是会在调试时发现一些小问题。例如，当我们从日历中读取日期时，对于 08 这样的数据，如果不设定 radix 参数可能就会导致意想不到的结果。

注意，ECMAScript 5 移除了八进制的默认表示法，这避免了其在 parseInt()中与十进制的混淆。

E.2.2 parseFloat()

parseFloat()的功能与 parseInt()基本相同，只不过它仅支持将输入值转换为十进制数。因此，该函数只有一个参数。例如：

```
parseFloat('123');                  //123
parseFloat('1.23');                 //1.23
parseFloat('1.23abc.00');           //1.23
parseFloat('a.bc1.23');             //NaN
```

与 parseInt()相同的是，parseFloat()在遇到第一个异常字符时就会放弃，无论剩余的那部分字符串是否可用。

```
parseFloat('a123.34');              //NaN
parseFloat('12a3.34');              //12
```

此外，parseFloat()还可以接受指数形式的数据（这点与 parseInt()不同）。

```
parseFloat('123e-2');               //1.23
parseFloat('1e10');                 //10000000000
parseInt('1e10');                   //1
```

E.2.3 isNaN()

通过 isNaN()可以确定某个输入值是否是一个可以参与算术运算的数字。因而，该函数也可以用来检测 parseInt()和 parseFloat()的调用成功与否。例如：

```
isNaN(NaN);                         //true
isNaN(123);                         //false
isNaN(1.23);                        //false
isNaN(parseInt('abc123'));          //true
```

该函数也会始终试图将其所接收的输入转换为数字，例如：

```
isNaN('1.23');          //false
isNaN('a1.23');         //true
```

isNaN()函数是非常有用的，因为 NaN 自己不存在等值的概念，也就是说表达式 NaN=== NaN 返回的是 false，这确实让人匪夷所思。

E.2.4 isFinite()

isFinite()可以用来检查输入是否是一个既非 Infinity，也非 NaN 的数字。例如：

```
isFinite(Infinity);     //false
isFinite(-Infinity);    //false
isFinite(12);           //true
isFinite(1e308);        //true
isFinite(1e309);        //false
```

JavaScript 中的最大数字为 1.7976931348623157e+308，因此 1e309 会被视作为无穷数。

E.2.5 URI 编码/解码

在 URL（Uniform Resource Locator，统一资源定位符）或 URI（Uniform Resource Identifier，统一资源标识符）中，有一些字符是具有特殊含义的。如果想“转义”这些字符，就可以去调用函数 encodeURI() 或 encodeURIComponent()。前者会返回一个可用的 URL，而后者则会认为我们所传递的仅仅是 URL 的一部分。

【示例】对于下面这个查询字符串来说，这两个函数所返回的字符编码分别是：

```
var url = 'http://www.packtpub.com/scr ipt.php?q=this and that';
```

```
encodeURI(url);
// "http://www.packtpub.com/scr%20ipt.php?q=this%20and%20that"
encodeURIComponent(url);
//
"http%3A%2F%2Fwww.packtpub.com%2Fscr%20ipt.php%3Fq%3Dthis%20and%20that"
```

encodeURI()和 encodeURIComponent()分别都有各自对应的反编码函数：decodeURI() 和 decodeURIComponent()。

注意，在一些遗留代码中会看到相似的编码函数和反编码函数 escape()和 unescape()，但不赞成使用这些函数来执行相关的操作，它们的编码规则也不尽相同。

E.2.6 eval()

eval()会将其输入的字符串当作 JavaScript 代码来执行。例如：

```
eval('var ii = 2;');
ii;          // 2
```

这里的 eval('var ii = 2;')与表达式 var ii = 2;的执行效果是相同的。

尽管 eval()在某些情况下是很有用的，但如果有选择的话，应该尽量避免使用它。因为 eval()是这样一种函数。

- 在安全性方面，它拥有的功能很强大，但这也意味着很大的不确定性，如果对放在 eval()函数中的代码没有太多把握，最好还是不要这样使用。
- 在性能方面，它是一种由函数执行的“动态”代码，所以要比直接执行脚本要慢。

E.2.7 alert()

alert()不是 JavaScript 核心的一部分，它没有包括在 ECMA 标准中，而是由宿主环境—浏览器所提供的，其作用是显示一个带文本的消息对话框。这对于某些调试很有帮助。

当然，在使用这个函数之前，必须要明白这样做会阻塞当前的浏览器线程。也就是说，在 alert()的执行窗口关闭之前，当前所有的代码都会暂停执行。因此，对于一个忙碌的 Ajax 应用程序来说，alert()通常不是一个好的选择。

E.3 作用域

在 JavaScript 中，变量的定义并不是以代码块作为作用域的，而是以函数作为作用域。也就是说，如果变量是在某个函数中定义的，那么它在函数以外的地方是不可见的。而如果该变量是定义在 if 或者 for 这样的代码块中的，它在代码块之外是可见的。

E.3.1 全局变量和局部变量

在 JavaScript 中，全局变量指的是定义在所有函数之外的变量（也就是定义在全局代码中的变量），与之相对的是局部变量，所指的则是在某个函数中定义的变量。其中，函数内的代码可以像访问自己的局部变量那样访问全局变量，反之则不行。

如果声明一个变量时没有使用 var 语句，该变量就会被默认为全局变量。

【示例】下面是一个具体示例，函数 f()可以访问变量 global，在函数

f()以外，变量 local 是不存在的。

```
var global = 1;
function f() {
    var local = 2;
    global++;
    return global;
}
```

测试一下：

```
f();            // 2
f();            // 3
local;          //ReferenceError: local is not defined
```

首先，在函数 f()中定义了一个变量 local。在该函数被调用之前，这个变量是不存在的。该变量会在函数首次被调用时创建，并被赋予局部作用域。这使得我们可以在该函数以内的地方访问它。但是在函数外访问，将抛出异常。

【建议】

- 尽量将全局变量的数量降到最低，以避免命名冲突。因为如果有两个人在同一段脚本的不同函数中使用了相同的全局变量名，就很容易导致不可预测的结果和难以察觉的 bug。
- 最好总是使用 var 语句来声明变量。
- 可以考虑使用“单一 var”模式，即仅在函数体内的第一行使用一个 var 来定义这个作用域中所有需要的变量。这样一来，我们就能很轻松地找到相关变量的定义，并且在很大程度上避免了不小心污染全局变量的情况。

E.3.2 变量提升

【示例】下面是一个很有趣的例子，它显示了关于局部和全局作用域

的另一个重要问题。

```
var a = 123;
function f() {
    alert(a);
    var a = 1;
    alert(a);
}
f();
```

在上面示例中，第一个 alert()实际上显示的是 undefined，这是因为函数域始终优先于全局域，所以局部变量 a 会覆盖掉所有与它同名的全局变量，尽管在 alert()第一次被调用时，a 还没有被正式定义（即该值为 undefined），但该变量本身已经存在于本地空间了。这种特殊的现象叫作提升（hoisting）。

也就是说，当 JavaScript 执行过程进入新的函数时，这个函数内被声明的所有变量都会被移动（或者说提升）到函数最开始的地方。这个概念很重要，必须牢记。另外需要注意的是，被提升的只有变量的声明，这意味着，只有函数体内声明的这些变量在该函数执行开始时就存在，而与之相关的赋值操作并不会被提升，它还在其原来的位置上。

例如，在前面的例子中，局部变量本身被提升到了函数开始处，但并没有在开始处就被赋值为 1。上面示例可以被等价地改写为：

```
var a = 123;
function f() {
    var a;                  // same as: var a = undefined;
    alert(a);               // undefined
    a = 1;
    alert(a); // 1
}
```

当然，我们也可以采用在最佳实践中提到过的单一 var 模式。在这个例子中，可以手动提升变量声明的位置，这样一来代码就不会被 JavaScript

的提升行为所混淆了。

E.4 数据类型

在 JavaScript 中，函数实际上也是一种数据。这概念对于我们日后的学习至关重要。也就是说，我们可以把一个函数赋值给一个变量。例如：

```
var f = function() {
    return 1;
};
```

上面这种定义方式通常被叫作函数标识记法（function literal notation）。

function(){ return 1;}是一个函数表达式。函数表达式可以被命名，称为命名函数表达式（named function expression，NFE）。所以以下这种情况也是合法的，虽然我们不常用到（在这里，myFunc 是函数的名字，而不是变量；IE 会错误地创建 f 和 myFunc 两个变量。

```
var f = function myFunc() {
    return 1;
};
```

这样看起来，命名函数表达式与函数声明没有什么区别。但它们其实是不同的。两者的差别表现于它们所在的上下文。函数声明只会出现在程序代码里（在另一个函数的函数体中，或者在程序主体中）。

【示例 1】如果对函数变量调用 typeof，操作符返回的字符串将会是"function"。

```
function define() {
    return 1;
}
var express = function () {
    return 1;
};
```

```
typeof define;                  // "function"
typeof express;                 // "function"
```

所以，JavaScript 中的函数也是一种数据，只不过这种特殊的数据类型有两个重要的特性。

- 它们所包含的是代码。
- 它们是可执行的（或者说是可调用的）。

要调用某个函数，只需要在它的名字后面加一对括号即可。

【示例 2】下面这段代码工作与函数的定义方式无关，它演示的是如何像变量那样使用函数。

```
var sum = function(a, b) {
    return a + b;
};
var add = sum;
typeof add;                     //"function"
add(1, 2);                      //3
```

由于函数也是赋值给变量的一种数据，所以函数的命名规则与一般变量相同，即函数名不能以数字开头，并且可以由任意的字母、数字、下划线和美元符号组合而成。

E.4.1 匿名函数

我们可以这样定义一个函数。

```
var f = function(a){
    return a;
};
```

通过这种方式定义的函数常被称为匿名函数（即没有名字的函数），特别是当它不被赋值给变量单独使用的时候。在这种情况下，此类函数有两种优雅的用法。

- 可以将匿名函数作为参数传递给其他函数，这样，接收方函数就能利用传递的函数来完成某些事情。
- 可以定义某个匿名函数来执行某些一次性任务。

E.4.2 回调函数

既然函数与任何可以被赋值给变量的数据是相同的，那么它当然可以像其他数据那样被定义、删除、拷贝，以及当成参数传递给其他函数。

【示例】下面示例定义了一个函数，这个函数有两个函数类型的参数，然后它会分别执行这两个参数所指向的函数，并返回它们的返回值之和。

```
function invokeAdd(a, b){
    return a() + b();
}
```

下面简单定义两个参与加法运算的函数（使用函数声明模式），它们只是单纯地返回一个固定值。

```
function one() {
    return 1;
}
function two() {
    return 2;
}
```

将这两个函数传递给目标函数 invokeAdd()，就可以得到执行结果了。

```
invokeAdd(one, two);                    //3
```

我们也可以直接用匿名函数（即函数表达式）来代替 one()和 two()，以作为目标函数的参数，例如：

```
invokeAdd(function () {return 1; }, function () {return 2; });        //3
```

可以换一种可读性更高的写法。

```
invokeAdd(
```

```
        function () { return 1; },
        function () { return 2; }
    );                                          //3
```

也可以这样写。

```
    invokeAdd(
        function () {
            return 1;
        },
        function () {
        return 2;
        }
    );                                          //3
```

当我们将函数 A 传递给函数 B，并由 B 来执行 A 时，A 就成了一个回调函数（callback functions）。如果这时 A 还是一个无名函数，我们就称它为匿名回调函数。

什么时候使用回调函数呢？

- 可以在不做命名的情况下传递函数（这意味着可以节省变量名的使用）。
- 可以将一个函数调用操作委托给另一个函数（这意味着可以节省一些代码编写工作）。
- 有助于提升性能。

E.4.3 应用回调函数

在编程过程中，我们通常需要将一个函数的返回值传递给另一个函数。

【示例】在下面示例中，我们定义了两个函数：第一个是 multiplyByTwo()，该函数会通过一个循环将其所接受的三个参数分别乘以 2，并以数组的形式返回结果；第二个函数 addOne()只接受一个值，然后将

它加 1 并返回。

```
function multiplyByTwo(a, b, c) {
    var i, ar = [];
    for(i = 0; i < 3; i++) {
        ar[i] = arguments[i] * 2;
    }
    return ar;
}
function addOne(a) {
    return a + 1;
}
```

现在测试一下这两个函数，结果如下。

```
multiplyByTwo(1, 2, 3);                    //[2, 4, 6]
addOne(100);                               //101
```

如果实现三个元素在两个函数之间的传递。这需要定义另一个数组，用于存储来自第一步的结果。我们先从 multiplyByTwo()的调用开始。

```
var myarr = [];
myarr = multiplyByTwo(10, 20, 30);      //[20, 40, 60]
```

然后，使用循环遍历每个元素，并将它们分别传递给 addOne()。

```
for (var i = 0; i < 3; i++) {
    myarr[i] = addOne(myarr[i]);
}
myarr;                                     //[21, 41, 61]
```

进一步优化代码，对 multiplyByTwo()函数做一些改动，使其接受一个回调函数，并在每次迭代操作中调用它。

```
function multiplyByTwo(a, b, c, callback) {
    var i, ar = [];
    for(i = 0; i < 3; i++) {
        ar[i] = callback(arguments[i] * 2);
```

```
        }
        return ar;
    }
```

优化之后，之前的工作只需要一次函数调用就够了。

```
    myarr = multiplyByTwo(1, 2, 3, addOne);                    //[3, 5, 7]
```

同样，我们还可以用匿名函数来代替 addOne()，这样做可以节省一个额外的全局变量。

```
    multiplyByTwo(1, 2, 3, function (a){
        return a + 1;
    });                                                        //[3, 5, 7]
```

使用匿名函数更易于随时根据需求调整代码。例如：

```
    multiplyByTwo(1, 2, 3, function(a){
        return a + 2;
    });                                                        //[4, 6, 8]
```

E.4.4 即时函数

即时函数是匿名函数的一种特殊应用形式，即这种函数在定义后立即调用。例如：

```
    (
    function(){
        alert('boo');
    }
    )();
```

这种语法其实很简单：只需将匿名函数的定义放进一对括号中，然后外面再紧跟一对括号即可。其中，第二对括号起到的是“立即调用”的作用，

同时它也是我们向匿名函数传递参数的地方。

```
(
function(name){
    alert('Hello ' + name + '!');
}
)('dude');
```

另外，也可以将第一对括号闭合于第二对括号之后。这两种做法都有效。

```
(function () {
    // …
} () );
// 等于
(functioin () {
    // …
})();
```

使用即时（自调）匿名函数的好处是不会产生任何全局变量。当然，缺点在于这样的函数是无法重复执行的（除非将它放在某个循环或其他函数中）。这也使得即时函数非常适合于执行一些一次性的或初始化的任务。

如果需要的话，即时函数也可以有返回值，虽然并不常见。

```
var result = (function () {
    // something complex with
    // temporary local variables…
    // …
    // return something;
}());
```

当然在这个例子中，将整个函数表达式用括号包起来是不必要的，我们只要在函数最后使用一对括号来执行这个函数即可。所以上例又可以改为：

```
var result = function () {
    // something complex with
```

```
    // temporary local variables…
    // return something;
}();
```

虽然这种写法也有效，但可读性稍微差了点：不读到最后，无法知道 result 到底是一个函数，还是一个即时函数的返回值。

E.4.5 私有函数

函数与其他类型的值本质上是一样的，因此可以在一个函数内部定义另一个函数。例如：

```
function outer(param) {
    function inner(theinput) {
        return theinput * 2;
    }
    return 'The result is ' + inner(param);
}
```

我们也可以改用函数标识记法来写这段代码。

```
var outer = function (param) {
    var inner = function (theinput) {
        return theinput * 2;
    };
    return 'The result is ' + inner(param);
};
```

当调用全局函数 outer()时，本地函数 inner()也会在其内部被调用。由于 inner()是本地函数，它在 outer()以外的地方是不可见的，所以将它称为私有函数。

```
outer(2);          //"The result is 4"
outer(8);          //"The result is 16"
```

```
inner(2);                //ReferenceError: inner is not defined
```

下面列举出使用私有函数的优点。

- 有助于确保全局名字空间的纯净性。
- 确保私有性，这样可以选择只将一些必要的函数暴露给“外部世界”，而保留属于自己的函数，使它们不为该应用程序的其他部分所用。

E.4.6 返回函数

函数一般都会有一个返回值，即便不是显式返回，它也会隐式返回一个 undefined。既然函数能返回一个唯一值，那么这个值就也有可能是另一个函数。例如：

```
function a() {
    alert('A!');
    return function(){
        alert('B!');
    };
}
```

在这个例子中，函数 a()会在执行它的工作（弹出'A!'）之后返回另一个函数。而所返回的函数又会去执行另外一些事情（弹出'B!'）。我们只需将该返回值赋值给某个变量，然后就可以像使用一般函数那样调用它了。

```
var newFunc = a();
newFunc();
```

上面第一行执行的是 alert('A!')，第二行才是 alert ('B!')。

如果想让返回的函数立即执行，也可以不用将它赋值给变量，直接在该调用后面再加一对括号即可，效果是一样的。

```
a()();
```

E.4.7 重写自己

由于一个函数可以返回另一个函数，因此我们可以用新的函数来覆盖旧的。例如，在上面示例中，我们也可以通过 a()的返回值来重写 a()函数。

```
a = a();
```

当前这句依然只会执行 alert ('A!')，但如果我们再次调用 a()，它就会执行 alert ('B!')了。这对于要执行某些一次性初始化工作的函数来说会非常有用。这样一来，该函数可以在第一次被调用后重写自己，从而避免了每次调用时重复一些不必要的操作。

在上面示例中，我们在外面重定义该函数，将函数返回值赋值给函数本身。我们也可以让函数从内部重写自己。例如：

```
function a() {
    alert('A!');
    a = function(){
        alert('B!');
    };
}
```

这样，当我们第一次调用该函数时会有如下情况发生。

```
alert ('A!')
```

将会被执行（可以视之为一次性的准备操作）。全局变量 a 将会被重定义，并被赋予新的函数。而如果该函数再被调用的话，被执行的就将是 alert ('B!')了。

【示例】下面是一个组合型的应用示例。

```
var a = (function () {
    function someSetup () {
        var setup = 'done';
    }
    function actualWork() {
```

```
            alert('Worky-worky');
        }
        someSetup();
        return actualWork;
    }() );
```

在上面示例中有如下情况。

第 1 点，使用了私有函数 someSetup()和 actualWork()。

第 2 点，使用了即时函数：函数 a()的定义后面有一对括号，因此它会执行自调。

第 3 点，当该函数第一次被调用时，它会调用 someSetup()，并返回函数变量 actualWork 的引用。注意，返回值中是不带括号的，因此该结果仅仅是一个函数引用，并不会产生函数调用。

第 4 点，由于这里的执行语句是以 var a = ...开头的，因而该自调函数所返回的值会重新赋值给 a。

如果想测试一下自己对上述内容的理解，可以尝试回答一下这个问题：上面的代码在以下情景中分别会 alert()什么内容？

- 当它最初被载入时。
- 之后再次调用 a()时。

这项技术对于某些浏览器相关的操作会相当有用。因为在不同浏览器中，实现相同任务的方法可能是不同的，我们都知道浏览器的特性不可能因为函数调用而发生任何改变，因此，最好的选择就是让函数根据其当前所在的浏览器来重定义自己。这就是所谓的“浏览器兼容性探测”技术。

E.5 闭包

闭包是 JavaScript 语言的一个技术难点。在学习闭包之前，最好先回顾一下 JavaScript 作用域，然后再进行延伸扩展。

E.5.1 作用域链

JavaScript 只有有函数作用域，也就是说，在某函数内定义的所有变量在该函数外是不可见的。但如果该变量是在某代码块中被定义的（如在某个 if 或 for 语句中），那它在代码块外是可见的。例如：

```
var a = 1;
function f() {
    var b = 1;
    return a;
}
f();                    // 1
b;                      // ReferenceError: b is not defined
```

在上面代码中，变量 a 是属于全局域，而变量 b 的作用域就在函数 f()内。所以在 f()内，a 和 b 都是可见的；在 f()外，a 是可见的，b 则不可见。

【示例】在下面示例中，如果在函数 outer()中定义了另一个函数 inner()，那么，在 inner()中可以访问的变量既来自它自身的作用域，也可以来自其“父级”作用域。这就形成了一条作用域链（scope chain），该链的长度（或深度）则取决于我们的需要。

```
var global = 1;
function outer(){
    var outer_local = 2;
    function inner() {
        var inner_local = 3;
        return inner_local + outer_local + global;
    }
    return inner();
}
```

测试一下 inner()是否真的可以访问所有变量：

```
outer();                    //6
```

E.5.2 利用闭包突破作用域链

下面通过代码的方式来介绍一下闭包的概念。

```
var a = "global variable";
var F = function () {
    var b = "local variable";
    var N = function () {
        var c = "inner local";
    };
};
```

首先，当然就是全局作用域 G，我们可以将其视为包含一切的宇宙。其中可以包含各种全局变量（如 a1，a2）和函数（如 F）。

每个函数也都会拥有一块属于自己的私用空间，用以存储一些别的变量（例如 b）以及内部函数（例如 N）。

在上面示例中，如果我们在 a 点，那么就位于全局空间中。而如果是在 b 点，我们就在函数 F 的空间里，在这里我们既可以访问全局空间，也可以访问 F 空间。如果我们在 c 点，那就位于函数 N 中，我们可以访问的空间包括全局空间、F 空间和 N 空间。其中，a 和 b 之间是不连通的，因为 b 在 F 以外是不可见的。

但如果愿意的话，我们是可以将 c 点和 b 点连通起来的，或者说将 N 与 b 连通起来。当我们将 N 的空间扩展到 F 以外，并止步于全局空间以内时，就产生了一件有趣的东西——闭包。

N 将会和 a 一样置身于全局空间。而且由于函数还记得它在被定义时所设定的环境，因此它依然可以访问 F 空间并使用 b。因为现在 N 和 a 同处于一个空间，但 N 可以访问 b，而 a 不能。

那么，N 究竟是如何突破作用域链的呢？

我们只需要将它们升级为全局变量（不使用 var 语句）或通过 F 传递（或返回）给全局空间即可。

1. 闭包 1

首先，我们先来看一个函数。这个函数与之前所描述的一样，只不过在 F 中多了返回 N，而在函数 N 中多了返回变量 b，N 和 b 都可通过作用域链进行访问。

```
var a = "global variable";
var F = function () {
    var b = "local variable";
    var N = function () {
        var c = "inner local";
        return b;
    };
    return N;
};
```

函数 F 中包含了局部变量 b，因此后者在全局空间里是不可见的。

```
b;                          // ReferenceError: b is not defined
```

函数 N 有自己的私有空间，同时也可以访问 f()的空间和全局空间，所以 b 对它来说是可见的。因为 F()是可以在全局空间中被调用的（它是一个全局函数），所以我们可以将它的返回值赋值给另一个全局变量，从而生成一个可以访问 F()私有空间的新全局函数。

```
var inner = F();
inner();                    //"local variable"
```

2. 闭包 2

下面示例的最终结果与之前相同，但在实现方法上存在一些细微的不同。在这里 F()不再返回函数了，而是直接在函数体内创建一个新的全局函数 inner()。

首先，需要声明一个全局函数的占位符。尽管这种占位符不是必需的，但最好还是声明一下，然后，我们就可以将函数 F()定义如下。

```
var inner; // placeholder
var F = function (){
    var b = "local variable";
    var N = function () {
        return b;
    };
    inner = N;
};
```

现在，F()被调用时会发生什么。

```
F();
```

在 F()中定义了一个新的函数 N()，并且将它赋值给了全局变量 inner。由于 N()是在 F()内部定义的，它可以访问 F()的作用域，所以即使该函数后来升级成了全局函数，但它依然可以保留对 F()作用域的访问权。

```
inner();                                    //"local variable".
```

3. 闭包 3

事实上，每个函数都可以被认为是一个闭包。因为每个函数都在其所在域（即该函数的作用域）中维护了某种私有联系。但在大多数时候，该作用域在函数体执行完之后就自行销毁了。除非发生一些有趣的事，导致作用域被保持。

如果一个函数会在其父级函数返回之后留住对父级作用域的链接的话，相关闭包就会被创建起来。但其实每个函数本身就是一个闭包，因为每个函数至少都有访问全局作用域的权限，而全局作用域是不会被破坏的。

例如，下面示例创建了一个函数，该函数将返回一个子函数，而这个子函数返回的则是其父函数的参数。

```
function F(param) {
    var N = function(){
```

```
            return param;
        };
        param++;
        return N;
    }
```

然后调用它：

```
    var inner = F(123);
    inner();                ///124
```

注意，当返回函数被调用时，param++已经执行过一次递增操作了。所以 inner()返回的是更新后的值。由此可以看出，函数所绑定的是作用域本身，而不是在函数定义时该作用域中的变量或变量当前所返回的值。

4. 循环中的闭包

下面示例设计一个三次的循环操作，在每次迭代中都会创建一个返回当前循环序号的新函数。该新函数会被添加到一个数组中，并最终返回。具体代码如下。

```
    function F() {
        var arr = [], i;
        for (i = 0; i < 3; i++) {
            arr[i] = function () {
                return i;
            };
        }
        return arr;
    }
```

下面运行一下函数，并将结果赋值给数组 arr。

```
    var arr = F();
```

然后在每个数组元素后面加一对括号来调用它们。按通常的估计，它们应该会依照循环顺序分别输出 0、1 和 2。

```
arr[0]();          //3
arr[1]();          //3
arr[2]();          //3
```

显然，这并不是我们想要的结果。究竟是怎么回事呢？原来我们在这里创建了三个闭包，而它们都指向了一个共同的局部变量 i。但是，闭包并不会记录它们的值，它们所拥有的只是相关域在创建时的一个连接（即引用）。在这个例子中，变量 i 恰巧存在于定义这三个函数域中。对这三个函数中的任何一个而言，当它要去获取某个变量时，它会从其所在的域开始逐级寻找那个距离最近的 i 值。由于循环结束时 i 的值为 3，所以这三个函数都指向了这一共同值。

那么，如何纠正这种行为呢？换一种闭包形式。

```
function F() {
    var arr = [], i;
    for(i = 0; i < 3; i++) {
        arr[i] = (function (x){
            return function () {
                return x;
            }
        }(i));
    }
    return arr;
}
```

这样就能获得预期的结果了。

```
var arr = F();
arr[0]();          // 0
arr[1]();          // 1
arr[2]();          // 2
```

这里不再直接创建一个返回 i 的函数，而是将 i 传递给了另一个即时函数。在该函数中，i 就被赋值给了局部变量 x，这样一来，每次迭代中的

x 就会拥有各自不同的值了。

也可以定义一个内部函数（不使用即时函数）来实现相同的功能。要点是在每次迭代操作中，我们要在中间函数内将 i 的值“本地化”。

```
function F() {
    function binder(x) {
        return function(){
            return x;
        };
    }
    var arr = [], i;
    for(i = 0; i < 3; i++) {
        arr[i] = binder(i);
    }
    return arr;
}
```

E.5.3 getter 和 setter

下面介绍闭包的应用：getter 和 setter。

假设现在有一个变量，它所表示的是某类特定值，或某特定区间内的值。我们不想将该变量暴露给外部。所以，需要将它保护在相关函数的内部，然后提供两个额外的函数：一个用于获取变量值，另一个用于给变量重新赋值。并在函数中引入某种验证措施，以便在赋值之前给予变量一定的保护。另外，为简洁起见，我们对该类中的验证部分进行了简化：即这里只处理数字值。

我们将 getter 和 setter 这两种函数放在一个共同的函数中，并在该函数中定义 secret 变量，这使得两个函数能够共享同一作用域。具体代码如下。

```
var getValue, setValue;
(function() {
    var secret = 0;
    getValue = function(){
        return secret;
    };
    setValue = function (v) {
        if (typeof v === "number") {
            secret = v;
        }
    };
}());
```

这里一切都是通过一个即时函数来实现的，我们在其中定义了全局函数 setValue()和 getValue()，并以此来确保局部变量 secret 的不可直接访问性。

```
getValue();                    // 0
setValue(123);
getValue();                    //123
setValue(false);
getValue();                    //123
```

E.5.4 迭代器

下面介绍闭包的应用：闭包在实现迭代器方面的功能。

通常情况下，我们都知道如何用循环来遍历一个简单的数组，但是有时候需要面对更为复杂的数据结构，它们通常会有着与数组截然不同的序列规则。这时候就需要将一些“谁是下一个”的复杂逻辑封装成易于使用的 next()函数，然后，我们只需要简单地调用 next()就能实现对于相关的遍历操作了。

在下面示例中，我们将依然通过简单数组，而不是复杂的数据结构来说明问题。该例子是一个接受数组输入的初始化函数，我们在其中定义了一个私有指针 i，该指针会始终指向数组中的下一个元素。

```
function setup(x) {
    var i = 0;
    return function(){
        return x[i++];
    };
}
```

现在，我们只需用一组数据来调用一下 setup()，就可创建出我们所需要的 next()函数，具体如下。

```
var next = setup(['a', 'b', 'c']);
```

这是一种既简单又好玩的循环形式：我们只需重复调用一个函数，就可以不停地获取下一个元素。

```
next();                    //"a"
next();                    //"b"
next();                    //"c"
```