《征服 Ajax：Web2.0 开发技术详解》试读版	
作者：王沛　冯曼菲	出版社：人民邮电出版社
全国各大书店均有销售	
网上购书地址：http://www.dearbook.com.cn/book/106884 http://www.china-pub.com/computers/common/info.asp?id=29998	

第 6 章　JavaScript 面向对象程序设计

在传统的 Web 开发模式中，JavaScript 是一种点缀的作用，完成很有限的功能，诸如表单验证之类。其语言本身也一直被当作过程化的语言使用，很难完成复杂的功能。而 Ajax 的出现使得复杂脚本成为必需的组成部分，这就对 JavaScript 程序设计提出了新的要求，很多 Ajax 应用开始利用 JavaScript 面向对象的性质进行开发，使逻辑更加清晰。事实上，JavaScript 提供了完善的机制来实现面向对象的开发思想。

本章假设读者已经了解面向对象思想的基本概念，熟悉对象、类、继承等基本术语。以此为基础，将重点介绍如何在 JavaScript 中使用面向对象的思想，包括实现的原理、机制和技巧。

6.1　JavaScript 中支持面向对象的基础

6.1.1　用定义函数的方式定义类

在面向对象的思想中，最核心的概念之一就是类。一个类表示了具有相似性质的一类事物的抽象，通过实例化一个类，可以获得属于该类的一个实例（即对象）。

在 JavaScript 中定义一个类的方法如下：

```
function class1(){
    //类成员的定义及构造函数
}
```

这里 class1 既是一个函数也是一个类。作为函数，它可以理解为类的构造函数，负责初始化的工作。

6.1.2　使用 new 操作符获得一个类的实例

在前面介绍基本对象时，已经用过 new 操作符，例如：

```
new Date();
```

表示创建一个日期对象，而 Date 就是表示日期的类，只是这个类是由 JavaScript 内部提供的，而不是由用户定义的。

new 操作符不仅对内部类有效，对用户定义的类也是同样的用法，对于上节定义的 class1，也可以用 new 来获取一个实例：

```
function class1(){
    //类成员的定义及构造函数
}
var obj1=new class1();
```

抛开类的概念，从代码的形式上来看，class1 就是一个函数，那么是不是所有的函数都可以用 new 来操作呢？答案是肯定的。在 JavaScript 中，函数和类就是一个概念，当 new 一个函数时，就会返回一个对象。如果这个函数中没有初始化类成员，那就会返回一个空的对象。例如：

```
//定义一个 hello 函数
function hello(){
```

```
        alert(“hello”);
}
//通过 new 一个函数获得一个对象
var obj=new hello();
alert(typeof(obj));
```

从运行结果看，执行了 hello 函数，同时 obj 也获得了一个对象的引用。事实上，当 new 一个函数时，这个函数就是所代表类的构造函数，其中的所有代码都可以看作为了初始化一个对象而工作。用于表示类的函数也称之为构造器。

6.1.3 使用方括号（[]）引用对象的属性和方法

在 JavaScript 中，每个对象可以看作是多个属性（方法）的集合，引用一个属性（方法）很简单，即：

```
对象名.属性（方法）名
```

除此之外，还可以用方括号的形式来引用：

```
对象名[“属性（方法）名”]
```

注意，这里的方法名和属性名是一个字符串，而非原先点号后面的标识符，例如：

```
var arr=new Array();
//为数组添加一个元素
arr[“push”](“abc”);
//获得数组的长度
var len=arr[“length”];
//输出数组的长度
alert(len);
```

图 4.1 显示了执行的结果。

图 4.1　引用对象属性示例

由此可见，上面的代码等价于：

```
var arr=new Array();
//为数组添加一个元素
arr.push( “abc”);
//获得数组的长度
var len=arr.length;
//输出数组的长度
alert(len);
```

这种引用属性（方法）的方式和数组类似，也体现出一个 JavaScript 对象就是一组属性（方法）的集合这个性质。

这种用法适合不确定具体要引用哪个属性（方法）的场合，例如：一个对象用于表示用户资料，这时一个字符串表示要使用哪个属性，那就可以用这种方式来引用：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义了一个 User 类，包括两个成员 age 和 sex，并指定了初始值。
```

```
function User(){
	this.age=21;
	this.sex="male";
}
//创建 user 对象
var user=new User();
//根据下拉列表框显示用户的信息
function show(slt){
	if(slt.selectedIndex!=0){
		alert(user[slt.value]);
	}
}
//-->
</script>
<!--下拉列表框用于选择用户信息-->
<select onchange="show(this)">
	<option>请选择需要查看的信息：</option>
	<option value="age">年龄</option>
	<option value="sex">性别</option>
</select>
```

在这段代码中，使用一个下拉列表框让用户选择查看哪个信息，每个选项的 value 就表示用户对象的属性名称。这时如果不采用方括号的形式，就必须使用如下代码来达到预期效果：

```
function show(slt){
	if(slt.selectedIndex!=0){
		if(slt.value==”age”)alert(user.age);
		if(slt.value==”sex”)alert(user.sex);
	}
}
```

而使用方括号语法，则只需写为：

```
alert(user[slt.value]);
```

由此可见，方括号语法更像一种参数语法，可用一个变量来表示引用对象的哪个属性。如果不采用这种方法，又不想用条件判断，可以使用 eval 函数：

```
alert(eval(“user.”+slt.value));
```

这里利用 eval 函数的性质，执行了一段动态生成的代码，并返回了结果。

实际上，在前面讲述 document 的集合对象时，就有类似方括号的用法，比如引用页面中一个名为“theForm”的表单对象，曾经的用法是：

```
document.forms[“theForm”];
```

其实也可以写为：

```
document.forms.theForm;
```

但这里的 forms 对象是一个内部对象，和自定义对象不同的是，它还可以用索引来引用其中的一个属性。

6.1.4 动态添加、修改、删除对象的属性和方法

上一节介绍了如何引用一个对象的属性和方法，现在介绍如何为一个对象添加、修改或者删除属性和方法。

在其他语言中，对象一旦生成，就不可更改了，要为一个对象添加修改成员必须要在对应的类中修改，并重新实例化，而且程序必须经过重新编译。JavaScript 中却非如此，它提

供了灵活的机制来修改对象的行为，可以动态添加、修改、删除属性和方法。例如首先使用类 Object 来创建一个空对象 user：

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

1. 添加属性

这时 user 对象没有任何属性和方法，显然没有任何用途。但可以为它动态的添加属性和方法，例如：

```
user.name=”jack”;
user.age=21;
user.sex=”male”;
```

通过上述语句，user 对象便具有了三个属性：name、age 和 sex。下面输出这三个语句：

```
alert(user.name);
alert(user.age);
alert(user.sex);
```

由代码运行效果可知，三个属性已经完全属于 user 对象了。

2. 添加方法

添加方法的过程和属性类似：

```
user.alert=function(){
    alert(“my name is:”+this.name);
}
```

这就为 user 对象添加了一个方法“alert”，通过执行它，可以弹出一个对话框显示自己的名字介绍：

```
user.alert();
```

图 4.2 显示了执行的结果。

图 4.2　为 user 对象添加 alert 方法示例

3. 修改属性

修改一个属性的过程就是用新的属性替换旧的属性，例如：

```
user.name=”tom”;
user.alert=function(){
    alert(“hello,”+this.name);
}
```

这样就修改了 user 对象 name 属性的值和 alert 方法，它从显示“my name is”变为了显示“hello”。

4. 删除属性

删除一个属性的过程也很简单，就是将其置为 undefined：

```
user.name=undefined;
user.alert=undefined;
```

这样就删除了 name 属性和 alert 方法。在之后的代码中，这些属性变的不可用。

在添加、修改或者删除属性时，和引用属性相同，也可以采用方括号（[]）语法：

```
user[“name”]=”tom”;
```

使用这种方式还有一个额外的特点，就是可以使用非标识符字符串作为属性名称，例如标识符中不允许以数字开头或者出现空格，但在方括号（[]）语法中却可以使用：

```
user[“my name”]=”tom”;
```

需要注意，在使用这种非标识符作为名称的属性时，仍然要用方括号语法来引用：

```
alert(user[“my name”]);
```

而不能写为：

```
alert(user.my name);
```

利用对象的这种性质，甚至可以很容易实现一个简单的哈希表，在本书的后面将会看到其应用。此可见，JavaScript 中的每个对象都是动态可变的，这给编程带来了很大的灵活性，也和其他语言产生了很大的区别，读者可以体会这种性质。

6.1.5 使用大括号（{}）语法创建无类型对象

传统的面向对象语言中，每个对象都会对应到一个类。而上一节讲 this 指针时提到，JavaScript 中的对象其实就是属性（方法）的一个集合，并没有严格意义的类的概念。所以它提供了另外一种简单的方式来创建对象，即大括号（{}）语法：

```
{
	property1:statement,
	property2:statement2,
	…,
	propertyN:statmentN
}
```

通过大括号括住多个属性或方法及其定义（这些属性或方法用逗号隔开），来实现对象的定义，这段代码就直接定义个了具有 n 个属性或方法的对象，其中属性名和其定义之间用冒号（:）隔开。例如：

```
<script language="JavaScript" type="text/javascript">
<!--
var obj={};                    //定义了一个空对象
var user={
	name:"jack",           //定义了 name 属性，初始化为 jack
	favoriteColor:["red","green","black","white"],//定义了颜色喜好数组
	hello:function(){      //定义了方法 hello
		alert("hello,"+this.name);
	},
	sex:"male"             //定义了性别属性 sex，初始化为 sex
}

//调用 user 对象的方法 hello
user.hello();
//-->
</script>
```

第一行定义了一个无类型对象 obj，它等价于：

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

接着定义了一个对象 user 及其属性和方法。注意，除了最后一个属性（方法）定义，其他的必须以逗号（,）结尾。其实，使用动态增减属性的方法也可以定义一个完全相同的 user 对象，读者不妨使用前面介绍的方法做一个尝试。

使用这种方式来定义对象，还可以使用字符串作为属性（方法）名，例如：

```
var obj={“001”:”abc”}
```

这就给对象 obj 定义了一个属性“001”，这并不是一个有效的标识符，所以要引用这个属性必须使用方括号语法：

```
obj[“001”];
```

由此可见，无类型对象提供了一种创建对象的简便方式，它以紧凑和清晰的语法将一个对象体现为一个完整的实体。而且也有利于减少代码的体积，这对 JavaScript 代码来说尤其重要，因为要通过网络来下载，减少体积意味着提高了访问速度。

6.1.6 prototype 原型对象

prototype 对象是实现面向对象的一个重要机制。每个函数（function）其实也是一个对象，它们对应的类是“Function”，但它们身份特殊，每个函数对象都具有一个子对象 prototype。顾名思义，prototype 表示了该函数的原型，而函数也即是类，prototype 实际上就是表示了一个类的成员的集合。当通过 new 来获取一个类的对象时，prototype 对象的成员都会成为实例化对象的成员。

既然 prototype 是一个对象，可以使用前面两节介绍的方法对其进行动态的修改，这里先给出一个简单的例子：

```
//定义了一个空类
function class1(){
	//empty
}
//对类的 prototype 对象进行修改，增加方法 method
class1.prototype.method=function(){
	alert(“it’s a test method”);
}
//创建类 class1 的实例
var obj1=new class1();
//调用 obj1 的方法 method
obj1.method();
```

图 4.3 显示了执行的结果。

图 4.3　使用 prototype 示例

6.2 深入认识 JavaScript 中的函数

6.2.1 概述

函数是进行模块化程序设计的基础，然而到现在为止对 JavaScript 中函数的介绍还停留在一个很基础的阶段。尽管这已经足以应付一般的应用开发，但是对于复杂的 Ajax 应用，必须对函数有更深入的了解。JavaScript 中的函数不同于其它的语言，每个函数都是作为一

个对象被维护和运行的。通过函数对象的性质，可以很方便的将一个函数赋值给一个变量或者将函数作为参数传递。在继续讲述之前，先看一下函数的使用语法：

```
function func1(…){…}
var func2=function(…){…};
var func3=function func4(…){…};
var func5=new Function();
```

或许现在这些代码看上去有点匪夷所思，但这些都是声明函数的正确语法。它们和其它语言中常见的函数，或者之前介绍的函数定义方式有着很大的区别。那么在 JavaScript 中为什么能这么写？它所遵循的语法是什么呢？这些正是下面要介绍的内容。

6.2.2 认识函数对象（Function Object）

可以用 function 关键字定义一个函数，对于每个函数可以为其指定一个函数名，通过函数名来进行调用。这些都是代码给用户的印象，而在 JavaScript 解释执行的时候，实际上每个函数都是被维护为一个对象，这就是本小节将要介绍的函数对象（Function Object）。

函数对象与其它用户所定义的对象有着本质的区别，这一类对象被称之为内部对象，例如日期对象（Date）、数组对象（Array）、字符串对象（String）都是属于内部对象。换句话说，这些内置对象的构造器是由 JavaScript 本身所定义的：通过执行 new Array()这样的语句返回一个对象，JavaScript 内部有一套机制来初始化返回的对象，而不是由用户来指定对象的构造方式。

在 JavaScript 中，函数对象对应的类型是 Function，正如数组对象对应的类型是 Array，日期对象对应的类型是 Date 一样，可以通过 new Function()来创建一个函数对象，也可以通过 function 关键字来创建一个对象。为了便于理解，将函数对象的创建和数组对象的创建来比较。先看数组对象：下面两行代码的作用是一样的，都是创建一个数组对象 myArray：

```
var myArray=[];
//等价于
var myArray=new Array();
```

同样，下面的两段代码也是等价的，都是创建一个函数 myFunction：

```
function myFunction(a,b){
        return a+b;
}
//等价于
var myFunction=new Function("a","b","return a+b");
```

现在上面的代码还有些难以理解，但是通过和构造数组对象语句的比较，可以清楚的看到函数的对象本质，前面介绍的函数声明是上述代码的第一种方式，而在解释器内部，当遇到这种语法时，就会自动构造一个 Function 对象，将函数作为一个内部的对象来存储和运行。从这里也可以看到，一个函数对象名称（函数变量）和一个普通变量名称具有同样的规范，都可以通过变量名来引用这个变量，但是函数变量名后面可以跟上括号和参数列表来进行函数调用。

也许不会有人通过 new Function()的形式来创建一个函数，因为一个函数体通常会有多条语句，如果将它们以一个字符串的形式作为参数传递，那么代码的可读性会非常的差。下面介绍一下其使用语法：

```
var funcName=new Function(p1,p2,...,pn,body);
```

参数的类型都是字符串，p1 到 pn 表示所创建函数的参数名称列表，body 表示所创建函数的函数体语句，而 funcName 就是所创建函数的名称了。可以不指定任何参数创建一个空函数，不指定 funcName 创建一个无名函数，当然那样的函数什么用处都没有。

需要注意的是，前面说 p1 到 pn 是参数名称的列表，这意味着 p1 不仅仅只能代表一个参数，它也可以是一个逗号格开的参数列表，例如下面的定义是等价的：

```
new Function("a", "b", "c", "return a+b+c")
new Function("a, b, c", "return a+b+c")
new Function("a,b", "c", "return a+b+c")
```

JavaScript 引入 Function 类型并提供 new Function()这样的语法来创建函数并不是毫无意义的，在后面可以看到，函数作为一个对象，它本身就可以具有一些方法和属性，而为函数对象添加属性和方法就必须借助于 Function 这个类型。

现在已经认识到了函数的本质，它其实是一个内部对象，由 JavaScript 解释器决定其运行方式。通过上述代码创建的函数，在程序中可以使用函数名进行调用。于是在本节开头列出的函数定义问题也得到了解释：它们都是创建函数对象的正确语法。注意直接在函数声明后面加上括号就表示创建完成后立即进行函数调用，例如：

```
var i=function (a,b){
    return a+b;
}(1,2);
alert(i);
```

这段代码会显示变量 i 的值等于 3。i 是表示返回的值，而不是创建的函数，因为括号“(”比等号“=”有更高的优先级。这样的代码可能并不常用，但当用户想在很长的代码段中进行模块化设计或者想避免命名冲突，这是一个不错的解决办法。

需要注意的是，尽管下面两种创建函数的方法是等价的：

```
function funcName(){
    //函数体
}
//等价于
var funcName=function(){
    //函数体
}
```

但前面一种方式创建的是有名函数，而后面是创建了一个无名函数，只是让一个变量指向了这个无名函数。在使用上仅有一点区别，就是：对于有名函数，它可以出现在调用之后再定义；而对于无名函数，它必须是在调用之前就已经定义。例如：

```
<script language="JavaScript" type="text/javascript">
<!--
func();
var func=function(){
    alert(1)
}
//-->
</script>
```

这段语句将产生 func 未定义的错误，而：

```
<script language="JavaScript" type="text/javascript">
<!--
func();
function func(){
    alert(1)
}
//-->
</script>
```

则能够正确执行，甚至下面的语句也能正确执行：

```
<script language="JavaScript" type="text/javascript">
<!--
```

```
func();
var someFunc=function func(){
        alert(1)
}
//-->
</script>
```

由此可见，尽管 JavaScript 是一门解释型的语言，但它会在进行函数调用时，检查整个代码中是否存在相应的函数定义，这个函数名只有是通过 function funcName()形式定义的才会有效，而不能是匿名函数。

6.2.3 函数对象和其他内部对象的关系

除了函数对象，还有很多内部对象，比如：Object、Array、Date、RegExp、Math、Error。更准确的说，这些名称实际上表示一个类型，可以通过 new 操作符返回一个对象。然而函数对象和其它对象不同，当用 typeof 得到一个函数对象的类型时，它仍然会返回字符串“function”，而 typeof 一个数组对象或其它的对象时，它会返回字符串“object”。下面的代码示例了 typeof 不同类型的情况：

```
alert(typeof(Function)));
alert(typeof(new Function()));
alert(typeof(Array));
alert(typeof(Object));
alert(typeof(new Array()));
alert(typeof(new Date()));
alert(typeof(new Object()));
```

运行这段代码可以发现：前面 4 条语句都会显示“function”，而后面 3 条语句则显示“object”，可见 new 一个 function 实际上是返回一个函数。这与其它的对象有很大的不同。其它的类型 Array、Object 等都会通过 new 操作符返回一个普通对象。尽管函数本身也是一个对象，但它与普通的对象还是有区别的，因为它同时也是对象构造器，也就是说，可以 new 一个函数来返回一个对象，这在前面已经有过描述。所有 typeof 返回“function”的对象都是函数对象。也称这样的对象为构造器（constructor），因而，所有的构造器都是对象，但不是所有的对象都是构造器。

既然函数本身也是一个对象，它们的类型是 function，联想到 C++、Java 等面向对象语言的类定义，可以猜测到 Function 类型的作用所在，那就是可以给函数对象本身定义一些方法和属性，借助于函数的 prototype 对象，可以很方便的修改和扩充 Function 类型的定义，例如下面先扩展了函数类型 Function，为其增加了 method1 方法，作用是弹出对话框显示"function"：

```
Function.prototype.method1=function(){
        alert("function");
}
function func1(a,b,c){
        return a+b+c;
}
func1.method1();
func1.method1.method1();
```

注意最后一个语句：func1.method1.mehotd1()，它调用了 method1 这个函数对象的 method1 方法。虽然看上去有点容易混淆，但仔细观察一下语法还是很明确的：这是一个递归的定义。因为 method1 本身也是一个函数，所以它同样具有函数对象的属性和方法，所有

对 Function 类型的方法扩充都具有这样的递归性质。

Function 是所有函数对象的基础，而 Object 则是所有对象（包括函数对象）的基础。在 JavaScript 中，任何一个对象都是 Object 的实例，因此，可以修改 Object 这个类型来让所有的对象具有一些通用的属性和方法，修改 Object 类型是通过 prototype 来完成的：

```
Object.prototype.getType=function(){
    return typeof(this);
}
var array1=new Array();
function func1(a,b){
    return a+b;
}
alert(array1.getType());
alert(func1.getType());
```

上面的代码为所有的对象添加了 getType 方法，作用是返回该对象的类型。两条 alert 语句分别会显示“object”和“function”。

6.2.4 将函数作为参数传递

在前面已经介绍了函数的对象本质，每个函数在都被表示为一个特殊的对象，可以方便的将其赋值给一个变量，再通过这个变量名进行函数调用。作为一个变量，它可以以参数的形式传递给另一个函数，这在前面介绍 JavaScript 事件处理机制中已经看到过这样的用法，或许前面还对语法难以理解，但到了现在一切都变得顺其自然，例如下面的程序将 func1 作为参数传递给 func2：

```
function func1(theFunc){
    theFunc();
}
function func2(){
    alert("ok");
}
func1(func2);
```

在最后一条语句中，func2 作为一个对象传递给了 func1 的形参 theFunc，再由 func1 内部进行 theFunc 的调用。事实上，将函数作为参数传递，或者是将函数赋值给其它变量是所有事件机制的基础。

例如，如果需要在页面载入时进行一些初始化工作，可以先定义一个 init 的初始化函数，再通过 window.onload=init;语句将其绑定到页面载入完成的事件。这里的 init 就是一个函数对象，它可以加入 window 的 onload 事件列表，也正源于它的对象本质，这和 C 语言、C++语言的函数指针概念有很大的相似性。

6.2.5 传递给函数的隐含参数：arguments

当进行函数调用时，除了指定的参数外，还创建一个隐含的对象——arguments。arguments 是一个类似数组但不是数组的对象，说它类似是因为它具有数组一样的访问性质，可以用 arguments[index]这样的语法取值，拥有数组长度属性 length。arguments 对象存储的是实际传递给函数的参数，而不只局限于函数声明所定义的参数列表，例如：

```
function func(a,b){
    alert(a);
```

```
    alert(b);
    for(var i=0;i<arguments.length;i++){
        alert(arguments[i]);
    }
}
func(1,2,3);
```

代码运行时会依次显示：1，2，1，2，3。因此，在定义函数的时候，即使不指定参数列表，仍然可以通过 arguments 引用到所获得的参数，这给编程带来了很大的灵活性。arguments 对象的另一个属性是 callee，它表示对函数对象本身的引用，这有利于实现无名函数的递归或者保证函数的封装性，例如使用递归来计算 1 到 n 的自然数之和：

```
var sum=function(n){
    if(1==n)return 1;
    else return n+sum(n-1);
}
alert(sum(100));
```

其中函数内部包含了对 sum 自身的调用，然而对于 JavaScript 来说，函数名仅仅是一个变量名，在函数内部调用 sum 即相当于调用一个全局变量，不能很好的体现出是调用自身，所以使用 arguments.callee 属性会是一个较好的办法：

```
var sum=function(n){
    if(1==n)return 1;
    else return n+arguments.callee(n-1);
}
alert(sum(100));
```

callee 属性并不是 arguments 不同于数组对象的唯一特征，下面的代码说明了 arguments 不是由 Array 类型创建：

```
Array.prototype.p1=1;
alert(new Array().p1);
function func(){
    alert(arguments.p1);
}
func();
```

运行代码可以发现，第一个 alert 语句显示为 1，即表示数组对象拥有属性 p1，而 func 调用则显示为“undefined”，即 p1 不是 arguments 的属性，由此可见，arguments 并不是一个数组对象。

6.2.6 函数的 apply、call 方法和 length 属性

JavaScript 为函数对象定义了两个方法：apply 和 call，它们的作用都是将函数绑定到另外一个对象上去运行，两者仅在定义参数的方式有所区别：

```
Function.prototype.apply(thisArg,argArray);
Function.prototype.call(thisArg[,arg1[,arg2…]]);
```

从函数原型可以看到，第一个参数都被取名为 thisArg，也就是说，所有函数内部的 this 指针都会被赋值为 thisArg，这就达到了将函数作为另外一个对象的方法运行的目的。两个方法除了 thisArg 参数，都是为 Function 对象传递的参数。下面的代码说明了 apply 和 call 方法的工作方式：

```
//定义一个函数 func1，具有属性 p 和方法 A
function func1(){
    this.p="func1-";
```

```
        this.A=function(arg){
                alert(this.p+arg);
        }
}
//定义一个函数 func2，具有属性 p 和方法 B
function func2(){
        this.p="func2-";
        this.B=function(arg){
                alert(this.p+arg);
        }
}
var obj1=new func1();
var obj2=new func2();
obj1.A("byA");                          //显示 func1-byA
obj2.B("byB");                          //显示 func2-byB
obj1.A.apply(obj2,["byA"]);             //显示 func2-byA，其中[“byA”]是仅有一个元素的数组，下同
obj2.B.apply(obj1,["byB"]);             //显示 func1-byB
obj1.A.call(obj2,"byA");                //显示 func2-byA
obj2.B.call(obj1,"byB");                //显示 func1-byB
```

可以看出，obj1 的方法 A 被绑定到 obj2 运行后，整个函数 A 的运行环境就转移到了 obj2，即 this 指针指向了 obj2。同样 obj2 的函数 B 也可以绑定到 obj1 对象去运行。代码的最后 4 行显示了 apply 和 call 函数参数形式的区别。

与 arguments 的 length 属性不同，函数对象的还有一个参数相关的属性 length，它表示函数定义时所指定参数的个数，而非调用时实际传递的参数个数。例如下面的代码将显示 2：

```
function sum(a,b){
        return a+b;
}
alert(sum.length);
```

6.2.7 深入认识 JavaScript 中的 this 指针

this 指针是面向对象程序设计中的一项重要概念，它表示当前运行的对象。在实现对象的方法时，可以使用 this 指针来获得该对象自身的引用。

和传统意义的面向对象的语言不同，JavaScript 中的 this 指针是一个动态的变量，一个方法内的 this 指针并不是始终指向定义该方法的对象的，在上一节讲函数的 apply 和 call 方法时已经有过这样的例子。为了方便大家理解，再来看下面的例子：

```
<script language="JavaScript" type="text/javascript">
<!--
//创建两个空对象
var obj1=new Object();
var obj2=new Object();
//给两个对象都添加属性 p，并分别等于 1 和 2
obj1.p=1;
obj2.p=2;
//给 obj1 添加方法，用于显示 p 的值
obj1.getP=function(){
        alert(this.p);      //表面上 this 指针指向的是 obj1
}
//调用 obj1 的 getP 方法
obj1.getP();
```

```
//使 obj2 的 getP 方法等于 obj1 的 getP 方法
obj2.getP=obj1.getP;
//调用 obj2 的 getP 方法
obj2.getP();
//-->
</script>
```

从代码的执行结果看，分别弹出对话框 1 和 2。由此可见，getP 函数仅定义了一次，在不同的场合运行，显示了不同的运行结果，这是有 this 指针的变化所决定的。在 obj1 的 getP 方法中，this 就指向了 obj1 对象，而在 obj2 的 getP 方法中，this 就指向了 obj2 对象，并通过 this 指针引用到了两个对象都具有的属性 p。

由此可见，JavaScript 中的 this 指针是一个动态变化的变量，它表明了当前运行该函数的对象。由 this 指针的性质，也可以更好的理解 JavaScript 中对象的本质：一个对象就是由一个或多个属性（方法）组成的集合。每个集合元素不是仅能属于一个集合，而是可以动态的属于多个集合。这样，一个方法（集合元素）由谁调用，this 指针就指向谁。实际上，前面介绍的 apply 方法和 call 方法都是通过强制改变 this 指针的值来实现的，使 this 指针指向参数所指定的对象，从而达到将一个对象的方法作为另一个对象的方法运行的效果。

同时，每个对象集合的元素（即属性或方法）也是一个独立的部分，全局函数和作为一个对象方法定义的函数之间没有任何区别，因为可以把全局函数和变量看作为 window 对象的方法和属性。也可以使用 new 操作符来操作一个对象的方法来返回一个对象，这样一个对象的方法也就可以定义为类的形式，其中的 this 指针则会指向新创建的对象。在后面可以看到，这时对象名可以起到一个命名空间的作用，这是使用 JavaScript 进行面向对象程序设计的一个技巧。例如：

```
var namespace1=new Object();
namespace1.class1=function(){
    //初始化对象的代码
}
var obj1=new namespace1.class1();
```

这里就可以把 namespace1 看成一个命名空间。

由于对象属性（方法）的动态变化特性，一个对象的两个属性（方法）之间的互相引用，必须要通过 this 指针，而在传统语言中，this 关键字是可以省略的。但是上面的例子中：

```
obj1.getP=function(){
    alert(this.p);      //表面上 this 指针指向的是 obj1
}
```

这里的 this 关键字是不可省略的，即不能写成 alert(p)的形式。这将使得 getP 函数去引用上下文环境中的 p 变量，而不是 obj1 的属性。

6.3 类的实现

6.3.1 理解类的实现机制

在前面已经讲过，在 JavaScript 中可以使用 function 关键字来定义一个“类”。现在介绍如何为类添加成员。其过程很简单，在函数内通过 this 指针引用的变量或者方法都会成为类的成员，例如：

```
function class1(){
    var s=”abc”;
```

```
    this.p1=s;
    this.method1=function(){
        alert("this is a test method");
    }
}
var obj1=new class1();
```

当通过 new class1()获得对象 obj1 时，这个对象便自动获得了属性 p1 和方法 method1。

在 JavaScript 中，function 本身的定义就是类的构造函数，结合前面介绍过的对象的性质以及 new 操作符的用法，下面来看使用 new 创建对象的过程。

（1）当解释器遇到 new 操作符时便创建一个空对象；

（2）开始运行 class1 这个函数，并将其中的 this 指针都指向这个新建的对象；

（3）因为当给对象不存在的属性赋值时，解释器就会为对象创建该属性，例如在 class1 中，当执行到 this.p1=s 这条语句时，就会添加一个属性 p1，并把变量 s 的值赋给它，这样函数执行就是初始化这个对象的过程，即实现构造函数的作用；

（4）当函数执行完后，new 操作符就返回初始化后的对象。

通过这整个过程，JavaScript 中就实现了面向对象的基本机制。由此可见，在 JavaScript 中，function 的定义实际上就是实现一个对象的构造器，是通过函数来完成的。这种方式的缺点是：

- 将所有的初始化语句、成员定义都放到一起，代码逻辑不够清晰，要实现复杂的功能时往往力不从心。
- 因为每创建一个类的实例，都要执行一次构造函数。所以实际上构造函数中定义的属性和方法总被重复的创建，例如：

```
this.method1=function(){
        alert("this is a test method");
    }
```

这里的 method1 每创建一个 class1 的实例，都会被创建一次，造成了内存的浪费。下面一节，将介绍另外一种类定义的机制：prototype 对象，可以解决构造函数中定义类成员带来的缺点。

6.3.2 使用 prototype 对象定义类成员

上一节介绍了类的实现机制以及构造函数的实现，现在介绍另一种为类添加成员的机制：prototype 对象。当 new 一个 function 时，该对象的成员将自动赋给所创建的对象，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义一个只有一个属性 prop 的类
function class1(){
    this.prop=1;
}
//使用函数的 prototype 属性给类定义新成员
class1.prototype.showProp=function(){
    alert(this.prop);
}
//创建 class1 的一个实例
var obj1=new class1();
//调用通过 prototype 原型对象定义的 showProp 方法
obj1.showProp();
```

```
//-->
</script>
```

因为 prototype 是一个 JavaScript 对象，所以可以为 prototype 对象添加、修改、删除方法和属性。从而为一个类添加成员定义。

了解了函数的 prototype 对象，现在再来看 new 的执行过程。

（1）创建一个新的对象，并让 this 指针指向它；

（2）将函数的 prototype 对象的所有成员都赋给这个新对象；

（3）执行函数体，对这个对象进行初始化的操作；

（4）返回（1）中创建的对象。

和上一节介绍的 new 的执行过程相比，现在是多了用 prototype 来初始化对象的过程，这也和 prototype 的字面意思相符，它是所对应类的实例的原型。这个初始化过程发生在函数体（构造器）执行之前，所以可以在函数体内部调用 prototype 中定义的属性和方法，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义一个只有一个属性 prop 的类
function class1(){
	this.prop=1;
	this.showProp();
}
//使用函数的 prototype 属性给类定义新成员
class1.prototype.showProp=function(){
	alert(this.prop);
}
//创建 class1 的一个实例
var obj1=new class1();
//-->
</script>
```

和上一段代码相比，这里在 class1 的内部调用了 prototype 中定义的方法 showProp，从而在对象的构造过程中就弹出了对话框，显示 prop 属性的值为 1。

需要注意，原型对象的定义必须在创建类实例的语句之前，否则它将不会起作用，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义一个只有一个属性 prop 的类
function class1(){
	this.prop=1;
	this.showProp();
}
//创建 class1 的一个实例
var obj1=new class1();
//在创建实例的语句之后使用函数的 prototype 属性给类定义新成员，只会对后面创建的对象有效
class1.prototype.showProp=function(){
	alert(this.prop);
}
//-->
</script>
```

这段代码将会产生运行时错误，显示对象没有 showProp 方法，就是因为该方法的定义是在实例化一个类的语句之后。

由此可见，prototype 对象专用于设计类的成员，它是和一个类紧密相关的，除此之外，prototype 还有一个重要的属性：constructor，表示对该构造函数的引用，例如：

```
function class1(){
    alert(1);
}
class1.prototype.constructor();  //调用类的构造函数
```

这段代码运行后将会出现对话框，在上面显示文字“1”，从而更容易看出一个 prototype 是和一个类的定义紧密相关的。实际上：class1.prototype.constructor===class1。

6.3.3 一种 JavaScript 类的设计模式

尽管前面介绍了如何定义一个类，如何初始化一个类的实例，但既可以在 function 定义的函数体中添加成员，又可以用 prototype 定义类的成员，代码显的很混乱，和面向对象语言类的实现之间有着很大的区别。那么，如何以一种清晰的方式来定义类呢？下面给出了一种类的实现模式，并将它们对应到面向对象语言类的实现上。

类的构造函数用来初始化一个实例，是每个类必不可少的一部分。在传统意义的面向对象中，类的构造函数的名称和类的名称一致，同时它们的定义方式和类成员的定义是类似而又相互独立的。而在 JavaScript 中，由于对象的灵活的性质，在构造函数中也可以为类添加成员，在增加灵活性的同时，也增加了代码的复杂度。为了提高代码的可读性和开发效率，完全可以抛弃这种定义成员的方式，而使用 prototype 对象来替代，这样 function 的定义就是类的构造函数，符合传统意义类的实现：类名和构造函数名是相同的。例如：

```
function class1(){
    //构造函数
}
//成员定义
class1.prototype.someProperty=”sample”;
class1.prototype.someMethod=function(){
    //方法实现代码
}
```

虽然上面的代码对于类的定义已经清晰了很多，每定义一个属性或方法，都需要使用一次 class1.prototype，不仅代码体积变大，而且易读性还不够。为了进一步改进，可以使用无类型对象的构造方法来指定 prototype 对象，从而实现类的成员定义：

```
//定义一个类 class1
function class1(){
    //构造函数
}
//通过指定 prototype 对象来实现类的成员定义
class1.prototype={
    someProperty:”sample”,
    someMethod:function(){
        //方法代码
    },
    …//其他属性和方法.
}
```

上面的代码用一种很清晰的方式定义了 class1，构造函数直接用类名来实现，而成员使用无类型对象来定义，以列表的方式实现了所有属性和方法，并且可以在定义的同时初始化属性的值。这也更类似于传统意义面向对象语言中类的实现。只是构造函数和类的成员定义被分为了两个部分，不妨将其看成 JavaScript 中定义类的一种固定模式，这样在使用时会更加容易理解。

注意：在一个类的成员之间互相引用，必须通过 this 指针来进行，例如在上面例子中的 someMethod 方法中，如果要使用属性 someProperty，必须通过 this.someProperty 的形式，因为在 JavaScript 中每个属性和方法都是独立的，它们通过 this 指针联系在一个对象上。

6.4 公有成员、私有成员和静态成员

6.4.1 实现类的公有成员

事实上，在前面定义的任何类成员都属于公有成员的范畴，该类的任何实例都对外公开这些属性和方法。为了完整起见，安排了这一小节，其实现方式不再遨述，可参考前面一节的内容。

6.4.2 实现类的私有成员

私有成员即在类的内部实现中可以共享的成员，但是并不对外公开。JavaScript 中并没有特殊的机制来定义私有成员，但可以用一些技巧来实现这个功能。

这个技巧主要是通过变量的作用域性质来实现的，在 JavaScript 中，一个函数内部定义的变量称为局部变量，该变量不能够被函数外的程序所访问，却可以被函数内部定义的嵌套函数所访问。在实现私有成员的过程中，正是利用了这一性质。

前面提到，在类的构造函数中可以为类添加成员，通过这种方式定义的类成员，实际上共享了在构造函数内部定义的局部变量，这些变量就可以看作类的私有成员，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
function class1(){
    var pp=" this is a private property";   //私有属性成员 pp
    function pm(){          //私有方法成员 pm，显示 pp 的值
        alert(pp);
    }
    this.method1=function(){
        //在公有成员中改变私有属性的值
        pp="pp has been changed";
    }
    this.method2=function(){
        pm();       //在公有成员中调用私有方法
    }
}
var obj1=new class1();
obj1.method1();       //调用公有方法 method1
obj1.method2();       //调用公有方法 method2
//-->
</script>
```

图 4.4 显示了运行的结果。

图 4.4　私有成员使用示例

这样，就实现了私有属性 pp 和私有方法 pm。运行完 class1 以后，尽管看上去 pp 和 pm 这些局部变量应该随即消失，但实际上因为 class1 是通过 new 来运行的，它所属的对象还没消失，所以仍然可以通过公开成员来对它们进行操作。

> 注意：这些局部变量（私有成员），被所有在构造函数中定义的公有方法所共享，而且仅被在构造函数中定义的公有方法所共享。这意味着，在 prototype 中定义的类成员将不能访问在构造体中定义的局部变量（私有成员）。

要使用私有成员，是以牺牲代码可读性为代价的。而且这种实现更多的是一种 JavaScript 技巧，看上去也比较勉强，因为它并不是语言本身具有的机制。但这种利用变量作用域性质的技巧，却是值得借鉴的。

6.4.3　实现静态成员

和私有成员的勉强相比，静态成员则显得“正统”的多。静态成员即属于一个类的成员，它可以通过“类名.静态成员名”的方式访问。在 JavaScript 中，可以给一个函数对象直接添加成员来实现静态成员，因为函数也是一个对象，所以对象的相关操作，对函数同样适用。例如：

```
function class1(){//构造函数
}
//静态属性
class1.staticProperty=”sample”;
//静态方法
class1.staticMethod=function(){
        alert(class1.staticProperty);
}
//调用静态方法
class1.staticMethod();
```

通过上面的代码，就为类 class1 添加了一个静态属性和静态方法，并且在静态方法中引用了该类的静态属性。

如果要给每个函数对象都添加通用的静态方法，还可以通过函数对象所对应的类 Function 来实现，例如：

```
//给类 Function 添加原型方法：showArgsCount
Function.prototype.showArgsCount=function(){
        alert(this.length);      //显示函数定义的形参的个数
}
function class1(a){
        //定义一个类
}
//调用通过 Function 的 prototype 定义的类的静态方法 showArgsCount
class1. showArgsCount ();
```

由此可见，通过 Function 的 prototype 原型对象，可以给任何函数都加上通用的静态成员，这在实际开发中可以起到很大的作用，比如在著名的 prototype-1.3.1.js 框架中，就给所有的函数定义了以下两个方法：

```
//将函数作为一个对象的方法运行
Function.prototype.bind = function(object) {
    var __method = this;
    return function() {
        __method.apply(object, arguments);
    }
}
//将函数作为事件监听器
Function.prototype.bindAsEventListener = function(object) {
    var __method = this;
    return function(event) {
        __method.call(object, event || window.event);
    }
}
```

这两个方法在 prototype-1.3.1 框架中起了很大的作用，具体含义及用法将在后面章节介绍。

6.5 使用 for(...in...)实现反射机制

6.5.1 什么是反射机制

反射机制指的是程序在运行时能够获取自身的信息。例如一个对象能够在运行时知道自己有哪些方法和属性，这属于高级的面向对象程序设计的功能。在 C#中，利用程序集中的元数据来实现反射，能够在运行时动态判断和调用自己的属性或方法。

6.5.2 在 JavaScript 中利用 for(...in...)语句实现反射

在 JavaScript 中有一个很方便的语法来实现反射，即 for(…in…)语句，其语法如下：

```
for(var p in obj){
        //语句
}
```

这里 var p 表示声明的一个变量，用以存储对象 obj 的属性（方法）名称，有了对象名和属性（方法）名，就可以使用方括号语法来调用一个对象的属性（方法）：

```
for(var p in obj){
        if(typeof(obj[p]==”function”){
                obj[p]();
        }else{
                alert(obj[p]);
        }
}
```

这段语句就遍历 obj 对象的所有属性和方法，遇到属性则弹出它的值，遇到方法则立刻执行。该语句就是遍历一个集合内的所有成员。在后面可以看到，在面向对象的 JavaScript 程序设计中，反射机制是很重要的一种技术，它在实现类的继承中发挥了很大的作用。

6.5.3 使用反射来传递样式参数

在 Ajax 编程中，经常要能动态的改变界面元素的样式，可以通过对象的 style 属性来改变，比如要改变背景色为红色，可以这样写：

```
element.style.backgroundColor=”#ff0000”;
```

其中 style 对象有很多属性，基本上 CSS 里拥有的属性在 JavaScript 中都能够使用。现在考虑，如果一个函数接收参数，用以指定一个界面元素的样式，那就需要设计参数的实现方式，显然一个或几个参数是不能符合要求的，下面是一种实现：

```
function setStyle(_style){
    //得到要改变样式的界面对象
    var element=getElement();
    element.style=_style;
}
```

这样，直接将整个 style 对象作为参数传递了进来，一个 style 对象可能的形式是：

```
var style={
    color:#ffffff,
    backgroundColor:#ff0000,
    borderWidth:2px
}
```

这时可以这样调用函数：

```
setStyle(style);
```

或者直接写为：

```
setStyle({ color:#ffffff,backgroundColor:#ff0000,borderWidth:2px});
```

这段代码看上去没有任何问题，但实际上，在 setStyle 函数内部使用参数_style 为 element.style 赋值时，如果 element 原先已经有了一定的样式，例如曾经执行过：

```
element.style.height=”20px”;
```

而_style 中却没有包括对 height 的定义，因此 element 的 height 样式就丢失了，不是最初所要的结果。要解决这个问题，可以用反射机制来重写 setStyle 函数：

```
function setStyle(_style){
    //得到要改变样式的界面对象
    var element=getElement();
    for(var p in _style){
        element.style[p]=_style[p];
    }
}
```

程序中遍历_style 的每个属性，得到属性名称，然后再使用方括号语法将 element.style 中的对应的属性赋值为_style 中的相应属性的值。从而，element 中仅改变指定的样式，而其他样式不会改变，得到了所要的结果。

6.6 类的继承

6.6.1 利用共享 prototype 实现继承

继承是面向对象开发的又一个重要概念，它可以将现实生活的概念对应带程序逻辑中。例如水果是一个类，具有一些公共的性质；而苹果也是一类，但它们属于水果，所以苹果应

该继承于水果。

虽然在 JavaScript 中没有专门的机制来实现类的继承，但可以通过拷贝一个类的 prototype 到另外一个类来实现继承。一种简单的实现如下：

```
fucntion class1(){
	//构造函数
}

function class2(){
	//构造函数
}
class2.prototype=class1.prototype;
class2.prototype.moreProperty1=”xxx”;
class2.prototype.moreMethod1=function(){
	//方法实现代码
}
var obj=new class2();
```

这样，首先是 class2 具有了和 class1 一样的 prototype，不考虑构造函数，两个类是等价的。随后，又通过 prototype 给 class2 赋予了两个额外的方法。所以 class2 是在 class1 的基础上增加了属性和方法，这就实现了类的继承。

JavaScript 提供了 instanceof 操作符来判断一个对象是否是某个类的实例，对于上面创建的 obj 对象，下面两条语句都是成立的：

```
obj instanceof class1
obj instanceof class2
```

表面上看，上面的实现完全可行，而且 JavaScript 也能够正确的理解这种继承关系，obj 同时是 class1 和 class2 的实例。但实质上则不然，JavaScript 的这种理解实际上是基于一种很简单的策略。看下面的代码，先使用 prototype 让 class2 继承于 class1，再在 class2 中重复定义 method 方法：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义 class1
function class1(){
	//构造函数
}
//定义 class1 的成员
class1.prototype={
	m1:function(){
		alert(1);
	}
}
//定义 class2
function class2(){
	//构造函数
}
//让 class2 继承于 class1
class2.prototype=class1.prototype;
//给 class2 重复定义方法 method
class2.prototype.method=function(){
	alert(2);
}
//创建两个类的实例
var obj1=new class1();
```

```
var obj2=new class2();
//分别调用两个对象的 method 方法
obj1.method();
obj2.method();
//-->
</script>
```

从代码执行结果看，弹出了两次对话框“2”。由此可见，当对 class2 进行 prototype 的改变时，class1 的 prototype 也随之改变，即使对 class2 的 prototype 增减一些成员，class1 的成员也随之改变。所以 class1 和 class2 仅仅是构造函数不同的两个类，它们保持着相同的成员定义。说到这里，相信读者已经发现了其中的奥妙：class1 和 class2 的 prototype 是完全相同的，是对同一个对象的引用。其实从这条赋值语句就可以看出来：

```
//让 class2 继承于 class1
class2.prototype=class1.prototype;
```

在 JavaScript 中，除了基本的数据类型（数字、字符串、布尔等），所有的赋值以及函数参数都是引用传递，而不是值传递。所以上面的语句仅仅是让 class2 的 prototype 对象引用 class1 的 prototype，造成了类成员定义始终保持一致的效果。从这里也看到了 instanceof 操作符的执行机制，它就是判断一个对象是否是一个 prototype 的实例，因为这里的 obj1 和 obj2 都是对应于同一个 prototype，所以它们 instanceof 的结果都是相同的。

由此可见，使用 prototype 引用拷贝实现继承不是一种正确的办法。但在要求不严格的情况下，却也是一种合理的方法，唯一的约束是不允许类成员的覆盖定义。下面一节，将利用反射机制和 prototype 来实现正确的类继承。

6.6.2 利用反射机制和 prototype 实现继承

前面一节介绍的共享 prototype 来实现类的继承，并不是一种很好的方法，毕竟两个类是共享的一个 prototype，任何对成员的重定义都会互相影响，不是严格意义的继承。但在这个思想的基础上，可以利用反射机制来实现类的继承，思路如下：利用 for(…in…)语句枚举出所有基类 prototype 的成员，并将其赋值给子类的 prototype 对象。例如：

```
<script language="JavaScript" type="text/javascript">
<!--
function class1(){
    //构造函数
}
class1.prototype={
    method:function(){
        alert(1);
    },
    method2:function(){
        alert("method2");
    }
}
function class2(){
    //构造函数
}
//让 class2 继承于 class1
for(var p in class1.prototype){
    class2.prototype[p]=class1.prototype[p];
}
```

```
//覆盖定义 class1 中的 method 方法
class2.prototype.method=function(){
    alert(2);
}
//创建两个类的实例
var obj1=new class1();
var obj2=new class2();
//分别调用 obj1 和 obj2 的 method 方法
obj1.method();
obj2.method();
//分别调用 obj1 和 obj2 的 method2 方法
obj1.method2();
obj2.method2();
//-->
</script>
```

从运行结果可见，obj2 中重复定义的 method 已经覆盖了继承的 method 方法，同时 method2 方法未受影响。而且 obj1 中的 method 方法仍然保持了原有的定义。这样，就实现了正确意义的类的继承。为了方便开发，可以为每个类添加一个共有的方法，用以实现类的继承：

```
//为类添加静态方法 inherit 表示继承于某类
Function.prototype.inherit=function(baseClass){
    for(var p in baseClass.prototype){
        this.prototype[p]=baseClass.prototype[p];
    }
}
```

这里使用所有函数对象（类）的共同类 Function 来添加继承方法，这样所有的类都会有一个 inherit 方法，用以实现继承，读者可以仔细理解这种用法。于是，上面代码中的：

```
//让 class2 继承于 class1
for(var p in class1.prototype){
    class2.prototype[p]=class1.prototype[p];
}
```

可以写为：

```
//让 class2 继承于 class1
class2.inherit(class1)
```

这样代码逻辑变的更加清楚，也更容易理解。通过这种方法实现的继承，有一个缺点，就是在 class2 中添加类成员定义时，不能给 prototype 直接赋值，而只能对其属性进行赋值，例如不能写为：

```
class2.prototype={
    //成员定义
}
```

而只能写为：

```
class2.prototype.propertyName=someValue;
class2.prototype.methodName=function(){
    //语句
}
```

由此可见，这样实现继承仍然要以牺牲一定的代码可读性为代价，在下一节将介绍 prototype-1.3.1 框架（一个 JavaScript 组件库）中实现的类的继承机制，不仅基类可以用对象直接赋值给 property，而且在派生类中也可以同样实现，使代码逻辑更加清晰，也更能体现面向对象的语言特点。

6.6.3 prototype-1.3.1 框架中的类继承实现机制

prototype 框架是一个开源的 JavaScript 组件，它扩展了 JavaScript 内置对象的方法以及提供了新的工具类，简化了动态 Web 开发。此框架的详细介绍将在附录中说明。

在 prototype-1.3.1 框架中，首先为每个对象都定义了一个 extend 方法：

```
//为 Object 类添加静态方法：extend
Object.extend = function(destination, source) {
   for (property in source) {
      destination[property] = source[property];
   }
   return destination;
}
//通过 Object 类为每个对象添加方法 extend
Object.prototype.extend = function(object) {
   return Object.extend.apply(this, [this, object]);
}
```

Object.extend 方法很容易理解，它是 Object 类的一个静态方法，用于将参数中 source 的所有属性都赋值到 destination 对象中，并返回 destination 的引用。下面解释一下 Object.prototype.extend 的实现，因为 Object 是所有对象的基类，所以这里是为所有的对象都添加一个 extend 方法，函数体中的语句如下：

```
Object.extend.apply(this,[this,object]);
```

这一句是将 Object 类的静态方法作为对象的方法运行，第一个参数 this 是指向对象实例自身；第二个参数是一个数组，包括两个元素：对象本身和传进来的对象参数 object。函数功能是将参数对象 object 的所有属性和方法赋值给调用该方法的对象自身，并返回自身的引用。有了这个方法，下面看类继承的实现：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义 extend 方法
Object.extend = function(destination, source) {
   for (property in source) {
      destination[property] = source[property];
   }
   return destination;
}
Object.prototype.extend = function(object) {
   return Object.extend.apply(this, [this, object]);
}
//定义 class1
function class1(){
      //构造函数
}
//定义类 class1 的成员
class1.prototype={
      method:function(){
            alert("class1");
      },
      method2:function(){
            alert("method2");
      }
```

```
}
//定义 class2
function class2(){
	//构造函数
}
//让 class2 继承于 class1 并定义新成员
class2.prototype=(new class1()).extend({
	method:function(){
		alert("class2");
	}
});

//创建两个实例
var obj1=new class1();
var obj2=new class2();
//试验 obj1 和 obj2 的方法
obj1.method();
obj2.method();
obj1.method2();
obj2.method2();
//-->
</script>
```

从运行结果可以看出，继承被正确的实现了，而且派生类的额外成员也可以以列表的形式加以定义，提高了代码的可读性。下面解释继承的实现：

```
//让 class2 继承于 class1 并定义新成员
class2.prototype=(new class1()).extend({
	method:function(){
		alert("class2");
	}
});
```

看上去这段代码也可以写为：

```
//让 class2 继承于 class1 并定义新成员
class2.prototype=class1.prototype.extend({
	method:function(){
		alert("class2");
	}
});
```

但因为 extend 方法会改变调用该方法对象本身，所以上述调用会改变 class1 的 prototype 的值，犯了和 6.6.1 节中一样的错误。在 prototype-1.3.1 框架中，巧妙的利用 new class1()来创建一个实例对象，并将实例对象的成员赋值给 class2 的 prototype。其本质相当于创建了 class1 的 prototype 的一个拷贝，在这个拷贝上进行操作自然不会影响原有类中 prototype 的定义了。

6.7 实现抽象类

6.7.1 抽象类和虚函数

虚函数是类成员中的概念，即只做了一个声明而未实现的方法，具有虚函数的类就称之

为抽象类，这些虚函数在派生类中才被实现。抽象类是不能实例化的，因为其中的虚函数并不是一个完整的函数，不能被调用。所以抽象类一般只作为基类被派生以后再使用。

和类的继承一样，JavaScript 并没有任何机制用于支持抽象类，其实继承都没有，更何况抽象类了。但利用 JavaScript 语言本身的性质，可以实现自己的抽象类。

6.7.2 在 JavaScript 实现抽象类

在传统面向对象语言中，抽象类中的虚方法必须先被声明，但可以在其他方法中被调用。而在 JavaScript 中，虚方法就可以看作该类中没有定义的方法，但已经通过 this 指针使用了。和传统面向对象不同的是，这里虚方法不需经过声明，而直接使用了。这些方法将在派生类中实现，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义 extend 方法
Object.extend = function(destination, source) {
   for (property in source) {
      destination[property] = source[property];
   }
   return destination;
}
Object.prototype.extend = function(object) {
   return Object.extend.apply(this, [this, object]);
}
//定义一个抽象基类 base，无构造函数
function base(){}
base.prototype={
      initialize:function(){
            this.oninit();      //调用了一个虚方法
      }
}
//定义 class1
function class1(){
      //构造函数
}
//让 class1 继承于 base 并实现其中的 oninit 方法
class1.prototype=(new base()).extend({
      oninit:function(){      //实现抽象基类中的 oninit 虚方法
            //oninit 函数的实现
      }
});
//-->
</script>
```

这样，当在 class1 的实例中调用继承得到的 initialize 方法时，就会自动执行派生类中的 oninit()方法。从这里也可以看到解释型语言执行的特点，它们只有在运行到某一个方法调用时，才会检查该方法是否存在，而不会向编译型语言一样在编译阶段就检查方法存在与否。JavaScript 中则避免了这个问题。当然，如果希望在基类中添加虚方法的一个定义，也是可以的，只要在派生类中覆盖此方法即可。例如：

```
//定义一个抽象基类 base，无构造函数
function base(){}
```

```
base.prototype={
        initialize:function(){
                this.oninit();      //调用了一个虚方法
        },
        oninit:function(){}     //虚方法是一个空方法，由派生类实现
}
```

6.7.3 使用抽象类的示例

仍然以 prototype-1.3.1 为例，其中定义了一个类的创建模型：

```
//Class 是一个全局对象，有一个方法 create，用于返回一个类
var Class = {
  create: function() {
    return function() {
      this.initialize.apply(this, arguments);
    }
  }
}
```

这里 Class 是一个全局对象，具有一个方法 create，用于返回一个函数（类），从而声明一个类，可以用如下语法：

```
var class1=Class.create();
```

这样和函数的定义方式区分开来，使 JavaScript 语言能够更具备面向对象语言的特点。现在来看这个返回的函数（类）：

```
function(){
        this.initialize.apply(this, arguments);
}
```

这个函数也是一个类的构造函数，当 new 这个类时便会得到执行。它调用了一个 initialize 方法，从名字来看，是类的构造函数。而从类的角度来看，它是一个虚方法，是未定义的。但这个虚方法的实现并不是在派生类中实现的，而是创建完一个类后，在 prototype 中定义的，例如 prototype 可以这样写：

```
var class1=Class.create();
class1.prototype={
        initialize:function(userName){
                        alert("hello,"+userName);
        }
}
```

这样，每次创建类的实例时，initialize 方法都会得到执行，从而实现了将类的构造函数和类成员一起定义的功能。其中，为了能够给构造函数传递参数，使用了这样的语句：

```
function(){
        this.initialize.apply(this, arguments);
}
```

实际上，这里的 arguments 是 function()中所传进来的参数，也就是 new class1(args)中传递进来的 args，现在要把 args 传递给 initialize，巧妙的使用了函数的 apply 方法，注意不能写成：

```
this.initialize(arguments);
```

这是将 arguments 数组作为一个参数传递给 initialize 方法，而 apply 方法则可以把 arguments 数组对象的元素作为一组参数传递过去，不得不说是一种很巧妙的实现。

尽管这个例子在 prototype-1.3.1 中不是一个抽象类的概念，而是类的一种设计模式。但

实际上可以把 Class.create()返回的类看作所有类的共同基类，它在构造函数中调用了一个虚方法 initialize，所有继承于它的类都必须实现这个方法，完成构造函数的功能。它们得以实现的本质都是对于 prototype 的操作。

6.8 事件设计模式

6.8.1 事件设计概述

事件机制可以使程序逻辑更加符合现实世界，在 JavaScript 中很多对象都有自己的事件，例如按钮就有 onclick 事件，下拉列表框就有 onchange 事件，通过这些事件可以方便编程。那么对于自己定义的类，是否也实现事件机制呢？答案是肯定的，通过事件机制，可以将类设计为独立的模块，通过事件对外通信，提高了程序的开发效率。本节就将详细介绍 JavaScript 中的事件设计模式以及可能遇到的问题。

6.8.2 最简单的事件设计模式

最简单的一种模式是将一个类方法成员定义为事件，这不需要任何特殊的语法，通常是一个空方法，例如：

```
function class1(){
    //构造函数
}
class1.prototype={
    show:function(){
        //show 函数的实现
        this.onShow();        //触发 onShow 事件
    },
    onShow:function(){}  //定义事件接口
}
```

上面的代码中，就定义了一个方法：show()，同时该方法中调用了 onShow()方法，这个 onShow()方法就是对外提供的事件接口，其用法如下：

```
//创建 class1 的实例
var obj=new class1();
//创建 obj 的 onShow 事件处理程序
obj.onShow=function(){
    alert("onshow event");
}
//调用 obj 的 show 方法
obj.show();
```

代码执行结果如图 4.4 所示。由此可见，obj.onShow 方法在类的外部被定义，而在类的内部方法 show()中被调用，这就实现了事件机制。

图 4.4　事件设计示例

上述方法虽然很简单，但在实际的开发中却是经常可以使用的，常用来解决一些简单的事件功能。说它简单，因为它有以下两个缺点：

- 不能够给事件处理程序传递参数，因为是在 show()这个内部方法中调用事件处理程序的，无法知道外部的参数；
- 每个事件接口仅能够绑定一个事件处理程序，而内部方法则可以使用 attachEvent 或者 addEventListener 方法绑定多个处理程序。

在下面两小节将着重解决这个问题。

6.8.3　给事件处理程序传递参数

给事件处理程序传递参数不仅是自定义事件中存在的问题，也是系统内部对象的事件机制中存在的问题，因为事件机制仅传递一个函数的名称，不带有任何参数的信息，所以无法传递参数进去。例如：

```
//定义类 class1
function class1(){
    //构造函数
}
class1.prototype={
    show:function(){
        //show 函数的实现
        this.onShow();        //触发 onShow 事件
    },
    onShow:function(){}  //定义事件接口
}
//创建 class1 的实例
var obj=new class1();
//创建 obj 的 onShow 事件处理程序
function objOnShow(userName){
    alert("hello,"+userName);
}
//定义变量 userName
var userName="jack";
//绑定 obj 的 onShow 事件
obj.onShow=objOnShow;        //无法将 userName 这个变量传递进去
//调用 obj 的 show 方法
obj.show();
```

注意上面的 obj.onShow=objOnShow 事件绑定语句，不能够为了传递 userName 进去而写成：

```
obj.onShow=objOnShow(userName);
```

或者：

```
obj.onShow="objOnShow(userName)";
```

前者是将 objOnShow(userName)的运行结果赋给了 obj.onShow，而后者是将字符串“objOnShow(userName)”赋给了 obj.onShow。

要解决这个问题，可以从相反的思路去考虑，不考虑怎么把参数传进去，而是考虑如何构建一个无需参数的事件处理程序，该程序是根据有参数的事件处理程序创建的，是一个外层的封装。现在自定义一个通用的函数来实现这种功能：

```
//将有参数的函数封装为无参数的函数
function createFunction(obj,strFunc){
    var args=[];            //定义 args 用于存储传递给事件处理程序的参数
    if(!obj)obj=window;   //如果是全局函数则 obj=window;
    //得到传递给事件处理程序的参数
    for(var i=2;i<arguments.length;i++)args.push(arguments[i]);
    //用无参数函数封装事件处理程序的调用
    return function(){
        obj[strFunc].apply(obj,args);     //将参数传递给指定的事件处理程序
    }
}
```

该方法将一个有参数的函数封装为一个无参数的函数，不仅对全局函数适用，对作为对象方法存在的函数同样适用。该方法首先接收两个参数：obj 和 strFunc，obj 表示事件处理程序所在的对象；strFunc 表示事件处理程序的名称。除此以外，程序中还利用 arguments 对象处理第二个参数以后的隐式参数（即未定义形参的参数），并在调用事件处理程序时将这些参数传递进去。例如一个事件处理程序是：

```
someObject.eventHandler=function(_arg1,_arg2){
    //事件处理代码
}
```

应该调用：

```
createFunction(someObject,”eventHandler”,arg1,arg2);
```

这就返回一个无参数的函数，在返回的函数中已经包括了传递进去的参数。如果是全局函数作为事件处理程序，事实上它是 window 对象的一个方法，所以可以传递 window 对象作为 obj 参数，要是为了更清晰一点，也可以指定 obj 为 null，createFunction 函数内部会自动认为该函数是全局函数，从而自动把 obj 赋值为 window。下面来看应用的例子：

```
<script language="JavaScript" type="text/javascript">
<!--
//将有参数的函数封装为无参数的函数
function createFunction(obj,strFunc){
    var args=[];
    if(!obj)obj=window;
    for(var i=2;i<arguments.length;i++)args.push(arguments[i]);
    return function(){
        obj[strFunc].apply(obj,args);
    }
}
//定义类 class1
function class1(){
    //构造函数
}
class1.prototype={
    show:function(){
        //show 函数的实现
        this.onShow();          //触发 onShow 事件
    },
```

```
        onShow:function(){}  //定义事件接口
}
//创建 class1 的实例
var obj=new class1();
//创建 obj 的 onShow 事件处理程序
function objOnShow(userName){
        alert("hello,"+userName);
}
//定义变量 userName
var userName="jack";
//绑定 obj 的 onShow 事件
obj.onShow=createFunction(null,"objOnShow",userName);
//调用 obj 的 show 方法
obj.show();
//-->
</script>
```

在这段代码中，就将变量 userName 作为参数传递给了 objOnShow 事件处理程序。事实上，obj.onShow 得到的事件处理程序并不是 objOnShow，而是由 createFunction 返回的一个无参函数。

通过 createFunction 这的封装，就可以用一种通用的方案实现参数传递了。这不仅适用于自定义的事件，也适用于系统提供的事件，其原理是完全相同的。

6.8.4 使自定义事件支持多绑定

可以用 attachEvent 或者 addEventListener 方法来实现多个事件处理程序的同时绑定，不会互相冲突，而自定义事件怎样来实现多订阅呢？下面介绍这种实现。要实现多订阅，必定需要一个机制用于存储绑定的多个事件处理程序，在事件发生时同时调用这些事件处理程序。从而达到多订阅的效果，其实现如下：

```
<script language="JavaScript" type="text/javascript">
<!--
//定义类 class1
function class1(){
        //构造函数
}
//定义类成员
class1.prototype={
        show:function(){
                //show 的代码
                //...

                //如果有事件绑定则循环 onshow 数组，触发该事件
                if(this.onshow){
                        for(var i=0;i<this.onshow.length;i++){
                                this.onshow[i]();        //调用事件处理程序
                        }
                }
        },
        attachOnShow:function(_eHandler){
                if(!this.onshow)this.onshow=[];  //用数组存储绑定的事件处理程序引用
                this.onshow.push(_eHandler);
```

```
        }
}
var obj=new class1();
//事件处理程序 1
function onShow1(){
        alert(1);
}
//事件处理程序 2
function onShow2(){
        alert(2);
}
//绑定两个事件处理程序
obj.attachOnShow(onShow1);
obj.attachOnShow(onShow2);
//调用 show，触发 onshow 事件
obj.show();
//-->
</script>
```

从代码的执行结果可以看到，绑定的两个事件处理程序都得到了正确的运行。如果要绑定有参数的事件处理程序，只需加上 createFunction 方法即可，在上一节有过描述。

这种机制基本上说明了处理多事件处理程序的基本思想，但还有改进的余地。例如如果类有多个事件，可以定义一个类似于 attachEvent 的方法，用于统一处理事件绑定。在添加了事件绑定后如果想删除，还可以定义一个 detachEvent 方法用于取消绑定。这些实现的基本思想都是对数组的操作，这里不再赘述。读者不妨动手实践一下，以加深理解。

6.9 实例：使用面向对象思想处理 cookie

如果读者对 cookie 不熟悉，可以在第七章学习它的使用方法，虽然在那里创建了几个通用函数用于 cookie 的处理，但这些函数彼此分离，没有体现出是一个整体。联想到 JavaScript 中 Math 对象的功能，它其实就是通过 Math 这个全局对象，把所有的数学计算相关的常量和方法都联系到一起，作为一个整体使用，提高了封装性和使用效率。现在对 cookie 的处理事实上也可以按照这种方法来进行。

6.9.1 需求分析

对于 cookie 的处理，事实上只是封装一些方法，每个对象不会有状态，所以不需要创建一个 cookie 处理类，而只用一个全局对象来联系这些 cookie 操作。对象名可以理解为命名空间。下面考虑 cookie 操作有哪些经常的操作：

（1）设置 cookie 包括了添加和修改功能，事实上如果原有 cookie 名称已经存在，那么添加此 cookie 就相当于修改了此 cookie。在设置 cookie 的时候可能还会有一些可选项，用于指定 cookie 的声明周期、访问路径以及访问域。为了让 cookie 中能够存储中文，该方法中还需要对存储的值进行编码。

（2）删除一个 cookie，删除 cookie 只需将一个 cookie 的过期事件设置为过去的一个时间即可，它接收一个 cookie 的名称为参数，从而删除此 cookie。

（3）取一个 cookie 的值，该方法接收 cookie 名称为参数，返回该 cookie 的值。因为在

存储该值的时候已经进行了编码，所以取值时应该能自动解码，然后返回。

针对这些需求，下一小节将实现这些功能。

6.9.2 创建 Cookie 对象

因为是作为类名或者命名空间的作用，所以和 Math 对象类似，这里使用 Cookie 来表示该对象：

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

6.9.3 实现设置 Cookie 的方法

方法原型为：setCookie(name,value,option);其中 name 是要设置 cookie 的名称；value 是设置 cookie 的值；option 包括了其他选项，是一个对象作为参数。其实现如下：

```
Cookie.setCookie=function(name,value,option){
    //用于存储赋值给 document.cookie 的 cookie 格式字符串
    var str=name+"="+escape(value);
    if(option){
        //如果设置了过期时间
        if(option.expireDays){
            var date=new Date();
            var ms=option.expireDays*24*3600*1000;
            date.setTime(date.getTime()+ms);
            str+="; expires="+date.toGMTString();
        }
        if(option.path)str+="; path="+path;                    //设置访问路径
        if(option.domain)str+="; domain"+domain; //设置访问主机
        if(option.secure)str+="; true";                        //设置安全性
    }
    document.cookie=str;
}
```

6.9.4 实现取 Cookie 值的方法

方法原型为：getCookie(name);其中 name 是指定 cookie 的名称，从而根据名称返回相应的值。实现如下：

```
Cookie.getCookie=function(name){
    var cookieArray=document.cookie.split("; ");     //得到分割的 cookie 名值对
    var cookie=new Object();
    for(var i=0;i<cookieArray.length;i++){
        var arr=cookieArray[i].split("=");                    //将名和值分开
        if(arr[0]==name)return unescape(arr[1]);   //如果是指定的 cookie，则返回它的值
    }
    return "";
}
```

6.9.5 实现删除 Cookie 的方法

方法原型为：deleteCookie(name);其中 name 是指定 cookie 的名称，从而根据这个名称删除相应的 cookie。在本实现中，删除 cookie 是通过调用 setCookie 来完成的，将 option 的 expireDays 属性指定为负数即可：

```
Cookie.deleteCookie=function(name){
    this.setCookie(name,"",{expireDays:-1});  //将过期时间设置为过去来删除一个 cookie
}
```

通过上面的代码，整个 Cookie 对象就创建完毕，也可以将其放到一个大括号中来定义，例如：

```
var Cookie={
    setCookie:function(){},
    getCookie:function(){},
    deleteCookie:function(){}
}
```

通过这种形式，可以让 Cookie 的功能更加清晰，它作为一个全局对象，大大方便了对 Cookie 的操作，例如：

```
Cookie.setCookie("user","jack");
alert(Cookie.getCookie("user"));
Cookie.deleteCookie("user");
alert(Cookie.getCookie("user"));
```

上面的代码就先建立了一个名为 user 的 cookie，然后删除了该 cookie。两次 alert 输出语句显示了执行的效果。

本节通过建立一个 Cookie 对象来处理 cookie，方便了操作，也体现了面向对象的编程思想：把相关的功能封装在一个对象中。考虑到 JavaScript 语言的特点，本章没有选择需要创建类的面向对象编程的例子，那和一般面向对象语言没有大的不同。而是以 JavaScript 中可以直接创建对象为特点介绍了 Cookie 对象的实现及其工作原理。事实上这也和 JavaScript 内部对象 Math 的工作原理是类似的。

第 7 章　JavaScript 高级技术

JavaScript 有很多高级特性，包括框架编程、cookie 技术、正则表达式、window 对象、异常处理等，本章将详细讲解。

7.1 框架编程

7.1.1 框架编程概述

一个 Html 页面可以有一个或多个子框架，这些子框架以<iframe>来标记，用来显示一个独立的 Html 页面。这里所讲的框架编程包括框架的自我控制以及框架之间的互相访问，例如从一个框架中引用另一个框架中的 JavaScript 变量、调用其他框架内的函数、控制另一个框架中表单的行为等。

7.1.2 框架间的互相引用

一个页面中的所有框架以集合的形式作为 window 对象的属性提供，例如：window.frames 就表示该页面内所有框架的集合，这和表单对象、链接对象、图片对象等是类似的，不同的是，这些集合是 document 的属性。因此，要引用一个子框架，可以使用如下语法：

```
window.frames[“frameName”];
window.frames.frameName
window.frames[index]
```

其中，window 字样也可以用 self 代替或省略，假设 frameName 为页面中第一个框架，则以下的写法是等价的：

```
self.frames[“frameName”]
self.frames[0]
frames[0]
frameName
```

了解了如何引用一个框架，那么引用的这个框架到底是什么呢？其实，每个框架都对应一个 HTML 页面，所以这个框架也是一个独立的浏览器窗口，它具有窗口的所有性质，所谓对框架的引用也就是对 window 对象的引用。有了这个 window 对象，就可以很方便地对其中的页面进行操作，例如使用 window.document 对象向页面写入数据、使用 window.location 属性来改变框架内的页面等。

下面分别介绍不同层次框架间的互相引用：

1. 父框架到子框架的引用

知道了上述原理，从父框架引用子框架变的非常容易，即：

```
window.frames[“frameName”];
```

这样就引用了页面内名为 frameName 的子框架。如果要引用子框架内的子框架，根据引用的框架实际就是 window 对象的性质，可以这样实现：

```
window.frames[“frameName”].frames[“frameName2”];
```

这样就很引用到了二级子框架，以此类推，可以实现多层框架的引用。

2. 子框架到父框架的引用

每个 window 对象都有一个 parent 属性，表示它的父框架。如果该框架已经是顶层框架，则 window.parent 还表示该框架本身。

3. 兄弟框架间的引用

如果两个框架同为一个框架的子框架，它们称为兄弟框架，可以通过父框架来实现互相

引用，例如一个页面包括 2 个子框架：

```
<frameset rows="50%,50%">
    <frame src="1.html" name="frame1" />
    <frame src="2.html" name="frame2" />
</frameset>
```

在 frame1 中可以使用如下语句来引用 frame2：

```
self.parent.frames[“frame2”];
```

4. 不同层次框架间的互相引用

框架的层次是针对顶层框架而言的。当层次不同时，只要知道自己所在的层次以及另一个框架所在的层次和名字，利用框架引用的 window 对象性质，可以很容易地实现互相访问，例如：

```
self.parent.frames[“childName”].frames[“targetFrameName”];
```

5. 对顶层框架的引用

和 parent 属性类似，window 对象还有一个 top 属性。它表示对顶层框架的引用，这可以用来判断一个框架自身是否为顶层框架，例如：

```
//判断本框架是否为顶层框架
if(self==top){
    //dosomething
}
```

7.1.3 改变框架的载入页面

前面已经讲到，对框架的引用就是对 window 对象的引用，利用 window 对象的 location 属性，可以改变框架的导航，例如：

```
window.frames[0].location=”1.html”;
```

这就将页面中第一个框架的页面重定向到 1.html，利用这个性质，甚至可以使用一条链接来更新多个框架。

```
<frameset rows="50%,50%">
    <frame src="1.html" name="frame1" />
    <frame src="2.html" name="frame2" />
</frameset>
<!--somecode-->
<a href=”frame1.location=’3.html;frame2.location=’4.html’” onclick=””>link</a>
<!--somecode-->
```

7.1.4 引用其他框架内的 JavaScript 变量和函数

在介绍引用其他框架内 JavaScript 变量和函数的技术之前，先来看以下代码：

```
<script language="JavaScript" type="text/javascript">
<!--
function hello(){
    alert("hello,ajax!");
}
window.hello();
//-->
</script>
```

如果运行了这段代码，会弹出“hello,ajax!”的窗口，这正是执行 hello()函数的结果。

那为什么 hello()变成了 window 对象的方法呢？事实上，在一个页面内定义的所有全局变量和全局函数都是作为 window 对象的成员。例如：

```
var a=1;
alert(window.a);
```

就会弹出对话框显示为 1。同样的原理，在不同框架之间共享变量和函数，就是要通过 window 对象来调用。

为了方便大家的理解，下面举例说明。一个商品浏览页面由两个子框架组成，左侧表示商品分类的链接；当用户单击分类链接时，右侧显示相应的商品列表；用户可以单击商品旁的【购买】链接将商品加入购物车。

在这个例子中，可以利用左侧导航页面来存储用户希望购买的商品，因为当用户单击导航链接时，变化的是另外一个页面，即商品展示页面，而导航页面本身是不变的，因此其中的 JavaScript 变量不会丢失，可以用来存储全局数据。其实现原理如下：

假设左侧页面为 links.html，右侧页面为 show.html，页面结构如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title> New Document </title>
</head>
<frameset cols="20%,80%">
    <frame src="link.html" name="link" />
    <frame src="show.html" name="show" />
</frameset>
</html>
```

在 show.html 中展示的商品旁边可以加入这样一条语句：

```
<a href=”void(0)” onclick=”self.parent.link.addToOrders(32068)”>加入购物车</a>
```

其中 link 表示导航框架，在 link.html 页面中定义了 arrOrders 数组来存储商品的 id，函数 addToOrders()用来响应商品旁边【购买】链接的单击事件，它接收的参数 id 表示商品的 id，例子中是一个 id 为 32068 的商品：

```
<script language="JavaScript" type="text/javascript">
<!--
var arrOrders=new Array();
function addToOrders(id){
    arrOrders.push(id);
}
//-->
</script>
```

这样，在结帐页面或是购物车浏览页面就可以用 arrOrders 来获取所有准备购买的商品。

框架可以使一个页面划分为功能独立的多个模块，每个模块之间彼此独立，但又可以通过 window 对象的引用来建立联系，是 web 开发中的一个重要机制。在 Ajax 开发中，还可以利用隐藏框架实现各种技巧，在后面介绍 Ajax 实例编程时可以发现，无刷新上传文件以及解决 Ajax 的前进后退问题都会用到这种技术。

7.2 使用 cookie

7.2.1 cookie 概述

在上一节，曾经利用一个不变的框架来存储购物栏数据，而商品显示页面是不断变化的，尽管这样能达到一个模拟全局变量的功能，但并不严密。例如在导航框架页面内右击，殉葬快捷菜单中的【刷新】命令，则所有的 JavaScript 变量都会丢失。因此，要实现严格的跨页面全局变量，这种方式是不行的，JavaScript 中的另一个机制：cookie，则可以达到真正全局变量的要求。

准确的说，cookie 是浏览器提供的一种机制，它将 document 对象的 cookie 属性提供给 JavaScript。可以由 JavaScript 对其进行控制，而并不是 JavaScript 本身的性质。简单的来讲，cookie 是存于用户硬盘的一个文件，这个文件通常对应于一个域名，当浏览器再次访问这个域名时，便使这个 cookie 可用。因此，cookie 可以跨越一个域名下的多个网页，但不能跨越多个域名使用。

不同的浏览器对 cookie 的实现也不一样，但其性质是相同的。例如在 windows 2000 以及 windows xp 中，cookie 文件存储于 documents and settings\userName\cookie\文件夹下。通常的命名格式为：userName@domain.txt。

cookie 机制将信息存储于用户硬盘，因此可以作为全局变量，这是它最大的一个优点。一般来讲，它可以用于以下几种场合：

- 保存用户登录状态。例如将用户 id 存储于一个 cookie 内，这样当用户下次访问该页面时就不需要重新登录了，现在很多论坛和社区都提供这样的功能。cookie 还可以设置过期时间，当超过时间期限后，cookie 就会自动消失。因此，系统往往可以提示用户保持登录状态的时间：常见选项有一个月、三个月、一年等。
- 跟踪用户行为。例如一个天气预报网站，能够根据用户选择的地区显示当地的天气情况。如果每次都需要选择所在地是烦琐的，当利用了 cookie 后就会显得很人性化了，系统能够记住上一次访问的地区，当下次再打开该页面时，它就会自动显示上次用户所在地区的天气情况。因为一切都是在后台完成，所以这样的页面就像为某个用户所定制的一样，使用起来非常方便。
- 定制页面。如果网站提供了换肤或更换布局的功能，那么可以使用 cookie 来记录用户的选项，例如：背景色、分辨率等。当用户下次访问时，仍然可以保存上一次访问的界面风格。
- 创建购物车。正如在前面的例子中使用 cookie 来记录用户需要购买的商品，在结帐的时候可以统一提交。例如淘宝网就使用 cookie 记录了用户曾经浏览过的商品，方便随时进行比较。

当然，上述应用仅仅是 cookie 能完成的部分应用，还有更多的功能需要全局变量。任何事情都有正反两面，cookie 也不例外，它的缺点主要集中于安全性和隐私保护。主要包括以下几种：

- cookie 可能被禁用。当用户非常注重个人隐私保护时，他很可能禁用浏览器的 cookie 功能。
- cookie 是浏览器相关的。这意味着即使访问的是同一个页面，不同浏览器之间所保存的 cookie 也是不能互相访问的。
- cookie 可能被删除。因为每个 cookie 都是硬盘上的一个文件，因此很有可能被用

户删除。

- cookie 安全性不够高。所有的 cookie 都是以纯文本的形式记录于文件中，因此如果要保存用户名密码等信息时，最好事先经过加密处理。

7.2.2 设置 cookie

每个 cookie 都是一个名/值对，可以把下面这样一个字符串赋值给 document.cookie：

```
document.cookie=”userId=828”;
```

如果要一次存储多个名/值对，可以使用分号加空格（; ）隔开，例如：

```
document.cookie=”userId=828; userName=hulk”;
```

在 cookie 的名或值中不能使用分号（;）、逗号（,）、等号（=）以及空格。在 cookie 的名中做到这点很容易，但要保存的值是不确定的。如何来存储这些值呢？方法是用 escape()函数进行编码，它能将一些特殊符号使用十六进制表示，例如空格将会编码为“20%”，从而可以存储于 cookie 值中，而且使用此种方案还可以避免中文乱码的出现。例如：

```
document.cookie=”str=”+escape(“I love ajax”);
```

相当于：

```
document.cookie=”str=I%20love%20ajax”;
```

当使用 escape()编码后，在取出值以后需要使用 unescape()进行解码才能得到原来的 cookie 值，这在前面已经介绍过。

尽管 document.cookie 看上去就像一个属性，可以赋不同的值。但它和一般的属性不一样，改变它的赋值并不意味着丢失原来的值，例如连续执行下面两条语句：

```
document.cookie=”userId=828”;
document.cookie=”userName=hulk”;
```

这时浏览器将维护两个 cookie，分别是 userId 和 userName，因此给 document.cookie 赋值更像执行类似这样的语句：

```
document.addCookie(“userId=828”);
document.addCookie(“userName=hulk”);
```

事实上，浏览器就是按照这样的方式来设置 cookie 的，如果要改变一个 cookie 的值，只需重新赋值，例如：

```
document.cookie=”userId=929”;
```

这样就将名为 userId 的 cookie 值设置为了 929。

7.2.3 获取 cookie 的值

下面介绍如何获取 cookie 的值。cookie 的值可以由 document.cookie 直接获得：

```
var strCookie=document.cookie;
```

这将获得以分号隔开的多个名/值对所组成的字符串，这些名/值对包括了该域名下的所有 cookie。例如：

```
<script language="JavaScript" type="text/javascript">
<!--
document.cookie="userId=828";
document.cookie="userName=hulk";
var strCookie=document.cookie;
alert(strCookie);
//-->
</script>
```

图 7.1 显示了输出的 cookie 值。由此可见，只能够一次获取所有的 cookie 值，而不能指定 cookie 名称来获得指定的值，这正是处理 cookie 值最麻烦的一部分。用户必须自己分析这个字符串，来获取指定的 cookie 值，例如，要获取 userId 的值，可以这样实现：

```
<script language="JavaScript" type="text/javascript">
<!--
//设置两个 cookie
document.cookie="userId=828";
document.cookie="userName=hulk";
//获取 cookie 字符串
var strCookie=document.cookie;
//将多 cookie 切割为多个名/值对
var arrCookie=strCookie.split("; ");
var userId;
//遍历 cookie 数组，处理每个 cookie 对
for(var i=0;i<arrCookie.length;i++){
    var arr=arrCookie[i].split("=");
    //找到名称为 userId 的 cookie，并返回它的值
    if("userId"==arr[0]){
        userId=arr[1];
        break;
    }
}
alert(userId);
//-->
</script>
```

这样就得到了单个 cookie 的值，图 7.2 显示了获得的 userId 的值。

图 7.1　cookie 值示例

图 7.2　获取的单个 cookie 值

用类似的方法，可以获取一个或多个 cookie 的值，其主要的技巧仍然是字符串和数组的相关操作。

7.2.4　给 cookie 设置终止日期

到现在为止，所有的 cookie 都是单会话 cookie，即浏览器关闭后这些 cookie 将会丢失，事实上这些 cookie 仅仅是存储在内存中，而没有建立相应的硬盘文件。

在实际开发中，cookie 常常需要长期保存，例如保存用户登录的状态。这可以用下面的选项来实现：

```
document.cookie=”userId=828; expires=GMT_String”;
```

其中 GMT_String 是以 GMT 格式表示的时间字符串，这条语句就是将 userId 这个 cookie 设置为 GMT_String 表示的过期时间，超过这个时间，cookie 将消失，不可访问。例如：如果要将 cookie 设置为 10 天后过期，可以这样实现：

```
<script language="JavaScript" type="text/javascript">
```

```
<!--
//获取当前时间
var date=new Date();
var expireDays=10;
//将 date 设置为 10 天以后的时间
date.setTime(date.getTime()+expireDays*24*3600*1000);
//将 userId 和 userName 两个 cookie 设置为 10 天后过期
document.cookie="userId=828; userName=hulk; expire="+date.toGMTString();
//-->
</script>
```

7.2.5 删除 cookie

为了删除一个 cookie，可以将其过期时间设定为一个过去的时间，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
//获取当前时间
var date=new Date();
//将 date 设置为过去的时间
date.setTime(date.getTime()-10000);
//将 userId 这个 cookie 删除
document.cookie="userId=828; expire="+date.toGMTString();
//-->
</script>
```

7.2.6 指定可访问 cookie 的路径

默认情况下，如果在某个页面创建了一个 cookie，那么该页面所在目录中的其他页面也可以访问该 cookie。如果这个目录下还有子目录，则在子目录中也可以访问。例如在 www.xxxx.com/html/a.html 中所创建的 cookie，可以被 www.xxxx.com/html/b.html 或 www.xxx.com/html/some/c.html 所访问，但不能被 www.xxxx.com/d.html 访问。

为了控制 cookie 可以访问的目录，需要使用 path 参数设置 cookie，语法如下：

```
document.cookie=”name=value; path=cookieDir”;
```

其中 cookieDir 表示可访问 cookie 的目录。例如：

```
document.cookie=”userId=320; path=/shop”;
```

就表示当前 cookie 仅能在 shop 目录下使用。

如果要使 cookie 在整个网站下可用，可以将 cookie_dir 指定为根目录，例如：

```
document.cookie=”userId=320; path=/”;
```

7.2.7 指定可访问 cookie 的主机名

和路径类似，主机名是指同一个域下的不同主机，例如：www.google.com 和 gmail.google.com 就是两个不同的主机名。默认情况下，一个主机中创建的 cookie 在另一个主机下是不能被访问的，但可以通过 domain 参数来实现对其的控制，其语法格式为：

```
document.cookie=”name=value; domain=cookieDomain”;
```

以 google 为例，要实现跨主机访问，可以写为：

```
document.cookie=”name=value;domain=.google.com”;
```

这样，所有 google.com 下的主机都可以访问该 cookie。

7.2.8 综合示例：构造通用的 cookie 处理函数

cookie 的处理过程比较复杂，但都具有一定的相似性。因此可以定义几个函数来完成 cookie 的通用操作，从而实现代码的复用。下面列出了常用的 cookie 操作及其函数实现。

1. 添加一个 cookie：addCookie(name,value,expireHours)

该函数接收 3 个参数：cookie 名称，cookie 值，以及在多少小时后过期。这里约定 expireHours 为 0 时不设定过期时间，即当浏览器关闭时 cookie 自动消失。该函数实现如下：

```
<script language="JavaScript" type="text/javascript">
<!--
function addCookie(name,value,expireHours){
	var cookieString=name+"="+escape(value);
	//判断是否设置过期时间
	if(expireHours>0){
		var date=new Date();
		date.setTime(date.getTime+expireHours*3600*1000);
		cookieString=cookieString+"; expire="+date.toGMTString();
	}
	document.cookie=cookieString;
}
//-->
</script>
```

2. 获取指定名称的 cookie 值：getCookie(name)

该函数返回名称为 name 的 cookie 值，如果不存在则返回空，其实现如下：

```
<script language="JavaScript" type="text/javascript">
<!--
function getCookie(name){
	var strCookie=document.cookie;
	var arrCookie=strCookie.split("; ");
	for(var i=0;i<arrCookie.length;i++){
		var arr=arrCookie[i].split("=");
		if(arr[0]==name)return arr[1];
	}
	return "";
}
//-->
</script>
```

3. 删除指定名称的 cookie：deleteCookie(name)

该函数可以删除指定名称的 cookie，其实现如下：

```
<script language="JavaScript" type="text/javascript">
<!--
function deleteCookie(name){
	var date=new Date();
	date.setTime(date.getTime()-10000);
	document.cookie=name+"=v; expire="+date.toGMTString();
}
//-->
</script>
```

7.3 使用正则表达式

7.3.1 正则表达式概述

在前面已经涉及了一些正则表达式的用法，现在将系统地学习正则表达式的语法和用途。正则表达式主要用于进行字符串的模式匹配，例如判断一个字符串是否符合指定格式等。其实读者应该对正则表达式有所接触，例如在 windows 下搜索文件，可以用“*”或者“?”这样的通配符。在正则表达式的语法中，有更多这样的符号用于表示一个字符串的模式，表 7.1 列出了所有的特殊符号，它们也被称为元字符。

表 7.1 正则表达式中的元字符

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，“n”匹配字符“n”。“\n”匹配换行符。序列“\\”匹配“\”，“\(”匹配“(”
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与“\n”或“\r”之后的位置匹配
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与“\n”或“\r”之前的位置匹配
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配“z”和“zoo”。* 等效于 {0,}
+	一次或多次匹配前面的字符或子表达式。例如，“zo+”与“zo”和“zoo”匹配，但与“z”不匹配。+ 等效于 {1,}
?	零次或一次匹配前面的字符或子表达式。例如，“do(es)?”匹配“do”或“does”中的“do”。? 等效于 {0,1}
{n}	n 是非负整数。正好匹配 n 次。例如，“o{2}”与“Bob”中的“o”不匹配，但与“food”中的两个“o”匹配
{n,}	n 是非负整数。至少匹配 n 次。例如，“o{2,}”不匹配“Bob”中的“o”，而匹配“foooood”中的所有 o。'o{1,}' 等效于 'o+'。'o{0,}' 等效于 'o*'
{n,m}	m 和 n 是非负整数，其中 n <= m。至少匹配 n 次，至多匹配 m 次。例如，“o{1,3}”匹配“fooooood”中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间
?	当此字符紧随任何其他限定符（*、+、?、{n}、{n,}、{n,m}）之后时，匹配模式是“非贪心的”。“非贪心的”模式匹配搜索到的、尽可能短的字符串，而默认的“贪心的”模式匹配搜索到的、尽可能长的字符串。例如，在字符串“oooo”中，“o+?”只匹配单个“o”，而“o+”匹配所有“o”
.	匹配除“\n”之外的任何单个字符。若要匹配包括“\n”在内的任意字符，请使用诸如“[\s\S]”之类的模式
(pattern)	匹配 pattern 并捕获该匹配的子表达式。可以使用 $0...$9 属性从结果“匹配”集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用“\(”或者“\)”
(?:pattern)	匹配 pattern 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用“或”字符 (|) 组合模式部件的情况很有用。例如，与“industry|industries”相比，“industr(?:y| ies)”是一个更加经济的表达式
(?=pattern)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 pattern 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，“Windows (?=95| 98| NT| 2000)”与“Windows 2000”中的“Windows”匹配，但不与“Windows 3.1”中的“Windows”匹配。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后
(?!pattern)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 pattern 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，“Windows (?!95| 98| NT| 2000)”与“Windows 3.1”中的“Windows”匹配，但不与“Windows 2000”中的“Windows”匹配。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后
x| y	与 x 或 y 匹配。例如，“z| food”与“z”或“food”匹配。“(z| f)ood”与“zood”或“food”

	匹配
[xyz]	字符集。匹配包含的任一字符。例如，“[abc]”匹配“plain”中的“a”
[^xyz]	反向字符集。匹配未包含的任何字符。例如，“[^abc]”匹配“plain”中的“p”
[a-z]	字符范围。匹配指定范围内的任何字符。例如，“[a-z]”匹配“a”到“z”范围内的任何小写字母
[^a-z]	反向范围字符。匹配不在指定的范围内的任何字符。例如，“[^a-z]”匹配任何不在“a”到“z”范围内的任何字符
\b	匹配一个字边界，即字与空格间的位置。例如，“er\b”匹配“never”中的“er”，但不匹配“verb”中的“er”
\B	非字边界匹配。“er\B”匹配“verb”中的“er”，但不匹配“never”中的“er”
\cx	匹配由 x 指示的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是“c”字符本身
\d	数字字符匹配。等效于 [0-9]
\D	非数字字符匹配。等效于 [^0-9]
\f	换页符匹配。等效于 \x0c 和 \cL
\n	换行符匹配。等效于 \x0a 和 \cJ
\r	匹配一个回车符。等效于 \x0d 和 \cM
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]
\t	制表符匹配。与 \x09 和 \cI 等效
\v	垂直制表符匹配。与 \x0b 和 \cK 等效
\w	匹配任何字类字符，包括下划线。与“[A-Za-z0-9_]”等效。
\W	任何非字字符匹配。与“[^A-Za-z0-9_]”等效
\xn	匹配 n，此处的 n 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，“\x41”匹配“A”。“\x041”与“\x04”&“1”等效。允许在正则表达式中使用 ASCII 代码
\num	匹配 num，此处的 num 是一个正整数。到捕获匹配的反向引用。例如，“(.)\1”匹配两个连续的相同字符
\n	标识一个八进制转义码或反向引用。如果 \n 前面至少有 n 个捕获子表达式，那么 n 是反向引用。否则，如果 n 是八进制数 (0-7)，那么 n 是八进制转义码
\nm	标识一个八进制转义码或反向引用。如果 \nm 前面至少有 nm 个捕获子表达式，那么 nm 是反向引用。如果 \nm 前面至少有 n 个捕获，那么 n 是反向引用，后面跟 m。如果前面的条件均不存在，那么当 n 和 m 是八进制数 (0-7) 时，\nm 匹配八进制转义码 nm
\nml	当 n 是八进制数 (0-3)，m 和 l 是八进制数 (0-7) 时，匹配八进制转义码 nml
\un	匹配 n，其中 n 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)

使用这些元字符，可以表示具有特定模式的字符串，例如：

/^\s*$/：匹配一个空行。

/\d{2}-\d{5}/：匹配由两位数字、一个连字符再加 5 位数字组成的 ID 号。

/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/：匹配 HTML 标记。

像这种以斜杠开始和结尾的字符序列称为正则表达式，在 JavaScript 中可以很方便地使用这些表达式。

7.3.2 使用 RegExp 对象执行字符串模式匹配

RegExp 是 JavaScript 中的正则表达式对象，利用它可以完成字符串匹配的各种操作。获得一个 RegExp 对象可以有两种方式：

```
var objRegExp=/pattern*/[flag]
//或者
var objRegExp=new RegExp("pattern"[,"flag"]);
```

其中 pattern 是要匹配的模式，flag 表示搜索模式，有两个可选参数，分别是 g 和 i。g

表示全局搜索，在后面介绍的 replace 方法中非常有用；i 表示忽略大小写，默认情况下是大小写敏感的。例如：

```
/jack/ig
new RegExp(“jack”,”ig”);
```

都表示全局匹配文本中的“jack”单词，并且忽略大小写。

使用这两种创建方式的效果完全一样，可以直接使用。第一种方式甚至不需要引用变量，而直接把正则表达式当作对象来使用，例如：

```
/jack/ig.test(sourceString);
```

其中 test 就是正则表达式对象的一个方法，表 7.2 列出了正则表达式的所有方法。

表 7.2　正则表达式对象RegExp的方法

方法	描述
compile(pattern,flags)	将正则表达式转换为内部格式，对于批量匹配可以提高匹配效率
exec(str)	按照RegExp对象的匹配模式对str字符串进行匹配查找，当设定了全局搜索模式（g），则匹配查找从RegExp对象lastIndex属性所指定的目标字符串位置开始；若没有设置全局搜索，则从目标字符串第一个字符开始搜索。若没有任何匹配发生，返回null。 该方法将匹配结果放在一个数组内返回，该数组有三个属性 input:包含目标字符串，同RegExp.index index:匹配到的子字符串在目标字符串中的位置，同RegExp.index lastIndex:匹配到的子字符串后面一个字符的位置，同RegExp.lastIndex
test(str)	判断str是否符合指定的模式，返回一个布尔变量，true或者false。需要注意，这个方法不会改变RegExp的属性值

在执行完字符串匹配后，匹配结果会以 RegExp 静态属性的方式提供给脚本程序，每次执行 exec 都会改变这些静态属性，表 7.3 列出了 RegExp 对象的静态属性。

表 7.3　正则表达式对象RegExp的静态属性

静态属性	描述
RegExp.input	保存被搜索的字符串
RegExp.index	保存匹配的首字符的位置
RegExp.lastIndex	保存匹配的字符串下一个字符的位置
RegExp.lastMatch	保存匹配到的字符串
RegExp.lastParen	保存最后一个被匹配的字符串（最后一个括号的内容）
RegExp.leftContext	保存匹配字符串左边的内容
RegExp.rightContext	保存匹配字符串右边的内容
RegExp.$1~$9	保存最开始的9个子匹配（括号中的内容）

由此可见，所有的匹配结果都保存在同一个位置，当执行 exec 方法后，这些静态属性就会改变。因此，必须确保在执行完匹配后立即去使用匹配结果，或将它们保存到另外的变量中，而不再使用 RegExp 的这些属性。

7.3.3　提取子字符串

在匹配模式中，可以用小括号将子模式括起来，以获取子匹配的内容，这些匹配的结果被存储在 RegExp.$1～RegExp.$9 中。例如，对于 xml 片断：

```
<author>jack</author>
```

如果要使用正则表达式获取其中的作者：jack，可以这样实现：

```
<script language="JavaScript" type="text/javascript">
<!--
var strXml="<author>jack</author>";
var regExp=/<author>(\w*)<\/author>/;
```

```
regExp.exec(strXml);
var author=RegExp.$1;
alert(author);
//-->
</script>
```

图 7.3 给出了执行结果。

图 7.3　正则表达式提取子字符串示例

当需要提取多个子模式时，可以使用 RegExp.$1～RegExp.$9 依次获取得到的子字符串。如果需要得到的子模式不只 9 个，也可以使用 exec 返回的数组来获取子字符串。返回数组的长度为子模式的个数加 1，其中数组索引为 0 的元素表示被搜索的字符串，其后的元素依次对应于模式中的括号。例如上面的例子也可以用下面的代码实现：

```
<script language="JavaScript" type="text/javascript">
<!--
var strXml="<author>jack</author>";
var regExp=/<author>(\w*)<\/author>/;
//exec 返回一个数组对象
var arr=regExp.exec(strXml);
var author=arr[1];
alert(author);
//-->
</script>
```

7.3.4　和字符串相关的操作

在前面讲字符串相关的操作时，有 3 个方法的参数需要用到正则表达式，下面给出它们的具体用法。

1. string.search(regularExpression)

其中 string 是要处理的字符串，regularExpression 是匹配模式。该方法在 string 中查找指定的模式，如果找到，则返回它的第一个字符的索引位置，否则返回-1。例如：

```
<script language="JavaScript" type="text/javascript">
<!--
var strXml="<author>jack</author>";
var i=strXml.search(/jack/);
alert(i);
//-->
</script>
```

这段代码的最后结果显示为 8。和 indexOf 方法不同，该方法接收的是一个正则表达式，而 indexOf 只能接收一个字符串。但两者的行为是类似的。

2. string.replace(regularExpression,replaceString)

其中 regularExpression 是要查找的模式，replaceString 是要替换匹配模式的字符串。

regularExpression 也可以用普通字符串，但那样只能替换第一个出现的匹配，之后的匹配则被忽略。使用正则表达式后可以使用全局模式来实现整个替换。替换后原有字符串不发生变化，而是返回一个新的字符串。例如：在实际开发中经常要删除一段文本中的 Html 标记，以获取纯文本，可以使用如下代码实现。

```
<script language="JavaScript" type="text/javascript">
<!--
//定义函数用于删除文本中的 Html 标记
function stripTags(s) {
      return s.replace(/<\/?[^>]+>/gi, '');
}

var str=stripTags("<author>jack</author>");
alert(str);
//-->
</script>
```

最后，将得到去掉 Html 标记后的文本“jack”。

3. string.match(regularExpression)

该方法根据 regularExpression 正则表达式模式查找字符串 string 中的匹配字符项，将结果以数组形式返回。该数组有 3 个属性值，与 exec 方法返回的数组属性相同。若没有任何匹配，返回 null。

> 注意：若 regularExpression 对象未设定全局匹配模式，则数组索引为 0 的元素就是匹配的整体内容，索引为 1~9 的元素则包含了子匹配得到的字符。若设定了全局模式，则数组包含了搜索到的所有整体匹配项。

7.4 使用 window 对象

7.4.1 使用 window.open 方法新建窗口

window 对象表示的是浏览器窗口，它有多种操作，其中一个重要的方法是 open，表示新建一个窗口来打开指定页面。例如在 a.html 中执行以下语句：

```
window.open("b.html");
```

则新建一个窗口打开了 b.html 页面，这和在 a.html 页面中用一条链接打开页面的效果是一样的：

```
<a href="b.html" target="_blank">b</html>
```

但 window.open 对新建窗口的样式可以有更多的控制，例如：窗口大小、是否显示菜单栏、是否显示滚动条、是否显示地址栏等等。其完整的调用语法如下：

```
window.open(url,windowName,"name1=value1[,name2=value2,[…]]");
```

其中：url 是要打开的页面地址；windowName 表示新建窗口的名字，从而可以对其进行控制；最后是一个用字符串表示的参数列表，每一个参数都是名称和值对应的形式，用逗号隔开，其中可以使用的参数包括：

- height：表示新建窗口的高度；
- width：表示新建窗口的宽度；
- left：表示新建窗口到屏幕左边缘的距离；
- top：表示新建窗口到屏幕顶端的距离。

以上属性的单位均为象素，例如对于 800×600 的分辨率，left=400 则表示新窗口的左边缘出于屏幕的正中间。其余的属性主要是布尔型的，用 yes 或者 1 表示开启，用 no 或者 0 表示关闭。如果是开启，则 yes 或者 1 可省略，例如：toolbar=1 等价于 toolbar=yes 等价于 toolbar，下面分别介绍这些属性：

- directories：是否显示链接工具栏；
- location：是否显示地址栏；
- menubar：是否显示菜单栏；
- resizable：是否允许调整窗口大小；
- scrollbars：是否显示滚动条；
- status：是否显示状态栏；
- toolbar：是否显示工具栏。

例如，下面的代码将显示一个无菜单、无工具条、无滚动条的窗口：

```
window.open("test3.html","","height=200,width=300,toolbar=0,menuber=0,scrollbars=0");
```

图 7.4 显示了窗口的效果。

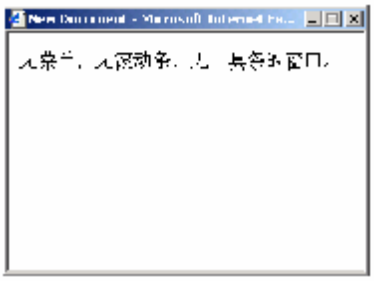

图 7.4　使用 window.open 控制打开窗口的样式

7.4.2　使用定时器实现 JavaScript 的延期执行或重复执行

window 对象提供了两个方法来实现定时器的效果，分别是 window.setTimeout()和 window.setInterval。其中前者可以使一段代码在指定时间后运行；而后者则可以使一段代码每过指定时间就运行一次。它们的原型如下：

```
window.setTimeout(expression,milliseconds);
window.setInterval(expression,milliseconds);
```

其中，expression 可以是用引号括起来的一段代码，也可以是一个函数名，到了指定的时间，系统便会自动调用该函数，当使用函数名作为调用句柄时，不能带有任何参数；而使用字符串时，则可以在其中写入要传递的参数。两个方法的第二个参数是 milliseconds，表示延时或者重复执行的毫秒数。下面分别介绍两种方法。

1. window.setTimeout 方法

该方法可以延时执行一个函数，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
function hello(){
	alert("hello");
}
window.setTimeout(hello,5000);
//-->
</script>
```

这段代码将使得页面打开 5 秒钟后显示对话框“hello”。其中最后一句也可以写为：

```
window.setTimeout(“hello()”,5000);
```

读者可以体会它们的差别，在 window.setInterval 方法中也有这样的性质。

如果在延时期限到达之前取消延时执行，可以使用 window.clearTimeout(timeoutId)方法，该方法接收一个 id，表示一个定时器。这个 id 是由 setTimeout 方法返回的，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
function hello(){
	alert("hello");
}
var id=window.setTimeout(hello,5000);
document.onclick=function(){
	window.clearTimeout(id);
}
//-->
</script>
```

这样，如果要取消显示，只需单击页面任何一部分，就执行了 window.clearTimeout 方法，使得超时操作被取消。

2. window.setInterval 方法

该方法使得一个函数每隔固定时间被调用一次，是一个很常用的方法。如果想要取消定时执行，和 clearTimeout 方法类似，可以调用 window.clearInterval 方法。clearInterval 方法同样接收一个 setInterval 方法返回的值作为参数。例如：

```
//定义一个反复执行的调用
var id=window.setInterval(“somefunction”,10000);
//取消定时执行
window.clearInterval(id);
```

上面的代码毫无意义，仅用于说明怎样取消一个定时执行。实际上在很多场合都需要用到 setInterval 方法，下面将设计一个秒表，来介绍 setInterval 函数的用途：该秒表将包括两个按钮和一个用于显示时间的文本框。当单击开始按钮时开始计时，最小单位为 0.01 秒，此时再次单击按钮则停止计时，文本框显示经过的时间。另外一个按钮用于将当前时间清零。其实现代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title> New Document </title>
</head>
<body>
<form action="somepage.asp">
<input type="text" value="0" name="txt1"/>
<input type="button" value="开始" name="btnStart"/>
<input type="button" value="重置" name="btnReset"/>
</form>
</body>
</html>
<script language="JavaScript" type="text/javascript">
<!--
//获取表单中的表单域
var txt=document.forms[0].elements["txt1"];
var btnStart=document.forms[0].elements["btnStart"];
var btnReset=document.forms[0].elements["btnReset"]
```

```
//定义定时器的 id
var id;
//每 10 毫秒该值增加 1
var seed=0;

btnStart.onclick=function(){
	//根据按钮文本来判断当前操作
	if(this.value=="开始"){
		//使按钮文本变为停止
		this.value="停止";
		//使重置按钮不可用
		btnReset.disabled=true;
		//设置定时器，每 0.01 秒跳一次
		id=window.setInterval(tip,10);
	}else{
		//使按钮文本变为开始
		this.value="开始";
		//使重置按钮可用
		btnReset.disabled=false;
		//取消定时
		window.clearInterval(id);
	}
}

//重置按钮
btnReset.onclick=function(){
	seed=0;
}
//让秒表跳一格
function tip(){
	seed++;
	txt.value=seed/100;
}
//-->
</script>
```

图 7.5 显示了 Firefox 中的运行效果。

图 7.5　秒表运行效果

7.4.3 给定时器调用传递参数

无论是 window.setTimeout 还是 window.setInterval，在使用函数名作为调用句柄时都不能带参数，而在许多场合必需要带参数，这就需要想方法解决。例如对于函数 hello(_name)，它用于针对用户名显示欢迎信息：

```
var userName="jack";
//根据用户名显示欢迎信息
function hello(_name){
    alert("hello,"+_name);
}
```

这时，如果企图使用以下语句来使 hello 函数延迟 3 秒执行是不可行的：

```
window.setTimeout(hello(userName),3000);
```

这将使 hello 函数立即执行，并将返回值作为调用句柄传递给 setTimeout 函数，其结果并不是程序需要的。而使用字符串形式可以达到想要的结果：

```
window.setTimeout(“hello(userName)”,3000);
```

这里的字符串是一段 JavaScript 代码，其中的 userName 表示的是变量。但这种写法不够直观，而且有些场合必须使用函数名，下面用一个小技巧来实现带参数函数的调用：

```
<script language="JavaScript" type="text/javascript">
<!--
var userName="jack";
//根据用户名显示欢迎信息
function hello(_name){
    alert("hello,"+_name);
}
//创建一个函数，用于返回一个无参数函数
function _hello(_name){
    return function(){
        hello(_name);
    }
}
window.setTimeout(_hello(userName),3000);
//-->
</script>
```

这里定义了一个函数_hello，用于接收一个参数，并返回一个不带参数的函数，在这个函数内部使用了外部函数的参数，从而对其调用，不需要使用参数。在 window.setTimeout 函数中，使用_hello(userName)来返回一个不带参数的函数句柄，从而实现了参数传递的功能。

7.4.4 使用 status 和 defaultStatus 属性改变状态栏信息

status 和 defaultStatus 是 window 对象的属性，用于设置状态栏信息，语法为：

```
window.status=”message”;
window.defaultStatus=”message”;
```

图 7.6 显示了状态栏的位置。

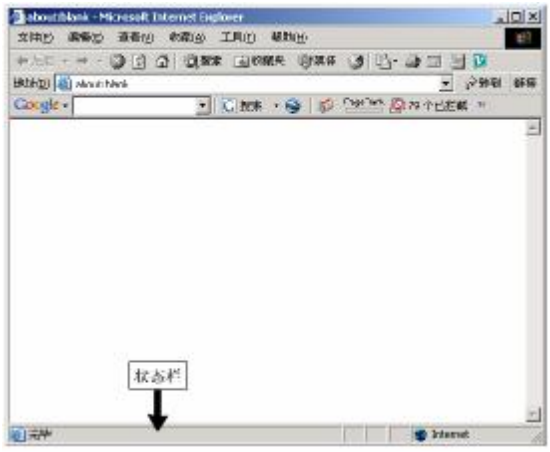

图 7.6　状态栏的位置

其中 status 属性就是用于设置状态栏显示的文本。而 defaultStatus 表示默认的状态栏信息，例如默认情况下 IE 会显示“完毕”，而 Firefox 则显示“完成”。可以通过 defaultStatus 来改变这一信息。

7.4.5　使用 alert、prompt 和 confirm 语句与用户进行交互

这三个语句都是弹出一个对话框，来处理用户输入。它们都是 window 对象的一个方法，在实际使用时，常常省略 window，而直接写成 alert(“hello”)类似的形式。下面分别介绍：

1. alert 语句

该语句的原型是：

```
window.alert(message);
```

alert 接收一个参数，该参数将转换为字符串直接显示在对话框上，例如：

```
alert(“hello,ajax”);
```

图 7.7 显示了执行效果。

图 7.7　alert 语句效果

2. prompt 语句

该语句的原型是：

```
window.prompt(message,defaultValue);
```

prompt 用于让用户输入一个值，其中 message 表示提示信息，defaultValue 表示显示于文本框的初始值；函数返回用户的输入。对话框包括【确定】和【取消】两个按钮，用户单击【确定】按钮则返回文本框中的内容，单击【取消】则返回 null。例如：

```
var userName=window.prompt(“请输入您的姓名：”,””);
alert(“hello,”+userName);
```

其中 prompt 用户让用户输入其姓名，使用 userName 变量获取用户输入，并显示欢迎信

息。图 7.8 显示了 prompt 语句的效果。

图 7.8　prompt 语句效果

3. confirm 语句

该语句的原型是：

```
window.confirm(message);
```

其作用是显示一条信息让用户确认，弹出的对话框包括【确定】和【取消】两个按钮，如果用户点击【确定】，则 confirm 函数返回 true，否则返回 false。例如下面的语句：

```
if(confirm("确定删除该记录吗？")){
    //删除记录的操作
}else{
    //不删除记录
}
```

图 7.9 显示了 confirm 语句的效果。

图 7.9　confirm 语句效果

7.5　异常处理

7.5.1　异常处理概述

在代码的运行过程中，错误是不可避免的，总的来说，错误发生于两种情况：一是程序内部的逻辑或者语法错误；二是运行环境或者用户输入中不可预知的数据造成的错误。对于前者，就称之为错误（error），可以通过调试程序来解决；而后一种则更多的称之为异常（exception），顾名思义，就是超出常规，没有按程序设计的意愿来输入数据。当然，异常还会有许多种类型。

所以说，异常并不等价于错误，相反，有时还会利用异常来解决一些问题。JavaScript 可以捕获一个异常并进行相应的处理，从而避免了浏览器向用户报错。

7.5.2　使用 try-catch-finally 处理异常

用户可以使用该结构处理可能发生异常的代码，如果发生异常，则由 catch 捕获并进行

处理，其语法如下：

```
try{
    //要执行的代码
}
catch(e){
    //处理异常的代码
}
finally{
    //无论异常发生与否，都会执行的代码
}
```

这和 Java 或者 C# 的异常处理的语言是一致的。通过异常处理，可以避免程序停止运行，从而具有了一定的自我修复能力。

在 Ajax 开发中，利用异常处理的一个典型应用就是创建 XMLHttpRequest 对象，不同浏览器创建它的方式是不一样的，为了使代码能够跨浏览器运行，就可以利用异常，一种方法不行，再用另一种方法，直到不发生异常为止，例如：

```
<script language="JavaScript" type="text/javascript">
<!--
var xmlhttp;
try{
    //尝试用 IE 的方式创建 XMLHttpRequest 对象
    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}catch(e){
    try{
        //尝试用非 IE 的方式常见 XMLHttpRequest 对象
        xmlhttp=new XMLHttpRequest();
    }catch(e){}
}
//-->
</script>
```

通过这种方式，就可以跨浏览器创建 XMLHttpRequest 对象。注意，即使不在 catch 块内进行处理，catch 标识及其参数 e 也是必须写的，否则会产生语法错误。而 finnally 则不是必需的。

7.5.3 使用 throw 语句抛出异常

在 JavaScript 中有其内部的异常机制，在遇到非法操作时能自动抛出异常；后面讲到的 DOM 模型标准中的一些方法也能够产生异常。但在实际的开发中，随着程序的复杂，需要能自己实现异常，这可以通过 throw 语句来实现：

```
throw value;
```

其中 value 就是要抛出的异常变量，它可以是 JavaScript 中的任何一种类型。但在 JavaScript 内部的异常中，异常参数（即 catch(e)中的 e）是一个名为 error 的对象，可以通过 new Error(message)来创建这个对象，异常的描述被作为 error 对象的一个属性 message，可以由构造函数传入，也可以之后赋值。通过这个异常描述，可以让程序获取异常的详细信息，从而自动处理。

例如，下面的程序计算两个数据的和，如果参数不是数字，则抛出异常：

```
<script language="JavaScript" type="text/javascript">
<!--
//函数默认要求参数为数字
```

```
function sum(a,b){
        a=parseInt(a);
        b=parseInt(b);
        //如果 a 或 b 不能转换为数字则抛出一个异常对象
        if(isNaN(a) || isNaN(b)){
                throw new Error("arguments are not numbers");
        }
        return a+b;
}

try{
        //错误的调用
        var s=sum("c","d");
}catch(e){
        //显示异常的详细信息
        alert(e.message);
}
//-->
</script>
```

程序中使用字母作为参数传递给 sum 函数，是错误的，所以函数内抛出了一个异常对象，这个对象被 catch 语句获取，并使用 alert 语句显示了其详细信息。图 7.10 显示了异常信息的内容。

图 7.10　异常信息

注意：使用 new Error(message)创建异常对象只是一种默认的习惯，也是内置异常的实现方式。但这并不是必需的，完全可以抛出任意数据类型的异常，例如一个整数，来作为异常的描述。只要在程序中抛出异常和捕获异常能匹配即可。

Error 对象除了 message 属性以外，还有一些其他的属性，但这些属性因浏览器而异，例如：在 IE 中，error 对象具有的属性包括 name、number、description、message；而在 Firefox 中，error 对象具有的属性包括 message、fileName、lineNumber、stack、name。在实际的应用中如果要实现自己的异常，这些属性只要被赋值，都是可用的，其中 Firefox 还会自动对 stack 属性赋值，用于显示异常出现的位置。