JavaScript：名字和版本

JavaScript是由Web发展初期的网景（Netscape）公司创建，“JavaScript”是Sun Microsystem公司（现在的Oracle）的注册商标，用来特指网景（现在的Mozilla）对这门语言的实现。网景将这门语言作为标准提交给了ECMA——欧洲计算机制造协会——由于商标上的冲突，这门语言的标准版本改了一个丑陋的名字“ECMAScript”。同样由于商标的冲突，微软对这门语言的实现版本取了一个广为人知的名字“Jscript”。实际上，几乎所有人都将这门语言叫做“JavaScript”。本书也仅仅使用“ECMAScript”来指代语言标准。

在最近10年间，所有的Web浏览器都实现了第3版ECMAScript标准，我们也已经不必再去考虑版本号了：语言标准已经很稳定了，并且被几乎所有浏览器完整地实现了。最近，ECMAScript第5版定义了新的语言标准，在撰写本书时，浏览器已经开始实现它了。除了ECMAScript 3长期保留下来的特性，本书还涵盖了所有ECMAScript 5的新特性。正如我们经常将JavaScript简写成JS一样，ECMAScript 3和ECMAScript 5有时也会简写成ES3和ES5。

当我们提到这门语言本身时，通常所指的语言版本是ECMAScript 3和ECMAScript 5（ECMAScript 4已经开发了数年，但由于太过庞大，从未发布过正式版本）。有时会看到JavaScript的版本号（比如JavaScript 1.5或JavaScript 1.8）。这些是Mozilla的版本号：版本1.5基本上就是ECMAScript 3，后续版本包含了非标准的语言扩展（参照第11章）。最后，JavaScript解释器或者“引擎”（engine）也有版本号，比如，Google将它的JavaScript解释器叫做V8，在撰写本书时V8引擎最新版本是3.0。

为了有用起见，通常每一种编程语言都有各自的开发平台、标准库或API函数，用来提供诸如基本输入输出的功能。JavaScript语言核心针对文本、数组、日期和正则表达式的操作定义了很少的API，但是这些API不包括输入输出功能。输入和输出功能（类似网络、存储和图形相关的复杂特性）是由JavaScript所属的“宿主环境”（host enviroment）提供的。这里所说的宿主环境通常是Web浏览器（第12章会介绍两个不基于Web浏览器的JavaScript宿主环境），本书第一部分涵盖了语言本身的特性以及少量的内置API。第二部分讲解了JavaScript如何在Web浏览器中工作，并涵盖基于浏览器的API——这部分也称做“客户端JavaScript”。

本书第三部分是核心API的参考手册。比如，在这部分，可以查找“Array”以获得JavaScript数组操作API的详细信息。第四部分是客户端JavaScript参考手册，比如，你可以在这部分查找“canvas”来获得HTML 5 <canvas>元素定义的图形编程API。

本书首先讲解初级的基础部分，然后讲解更高级和更抽象的内容。本书的章节也是如此安排以便你能循序阅读。但学习一门新的编程语言绝非易事，当然要完整描述这门语言就没办法苛求知识点的循次渐进，每一种语言特性都和其他特性相互关联，本书的知识点也是前后参照的，有时参照的知识点在后续章节，有时在已经阅读的章节。本章快速介绍了语言核心部分和客户端API及其关键特性，以便你能更方便更深入地阅读理解后续章节。

JavaScript 初探

当学习一门新的编程语言的时候，应当对照书中的示例边学边动手做，反复演练以加深自己的理解。因此，你需要一个JavaScript解释器。幸运的是，每一个Web浏览器都包含一个JavaScript解释器，当你阅读本书时，你可能已经在电脑上安装了不止一个Web浏览器了。

可以通过在HTML文件里写一个<script>标签来嵌入JavaScript代码，当浏览器加载HTML文件的时候，它会自动执行这段代码，随后会有提到。幸运的是，如果运行的是一小段JavaScript代码，则不必每次都这样做。我们可以利用Firefox的一个强大的革命性的插件Firebug（见图1-1，可以从*http://getfirebug.com/*下载Firebug）来运行这些小段代码，而且如今的Web浏览器带有很多开发工具，可以用来调试、试验和学习。通常在浏览器的“工具”菜单中可以看到类似“开发者工具”或者“Web控制台”的选项（Firefox 4内置了“Web控制台”，不过更推荐使用Firebug）。可以通过按F12键或者Ctrl+Shift+J快捷键来唤醒控制台[译注1]。控制台工具通常会在浏览器窗口的顶部或底部，有时候也可以单独打开一个窗口（见图1-1），这样会更加方便。

通常“开发者工具面板”窗口包含了很多选项卡，可以查看HTML文档结构、CSS样式、网络请求等。其中第一个选项卡是“JavaScript控制台”，可以直接输入JavaScript代码并运行出结果。用这种方式来调试JavaScript既简单又实用，这里强烈推荐读者使用这种方式来辅助你阅读本书。

一些现代浏览器有可能实现了一个简单的控制台API。可以通过使用函数console.log()来向控制台输出消息，使用console.log()来做简单的输出演示，通过这种方式可以非常方便地调试本书的示例代码。同样，也可以通过给alert()函数传入一段文本来弹出一个对话框，但这种输出调试信息的方法更具侵入性。

译注1： F12用来唤醒/关闭Firebug操作面板，Ctrl+Shift+J 用来唤醒错误控制台(Error Console)。

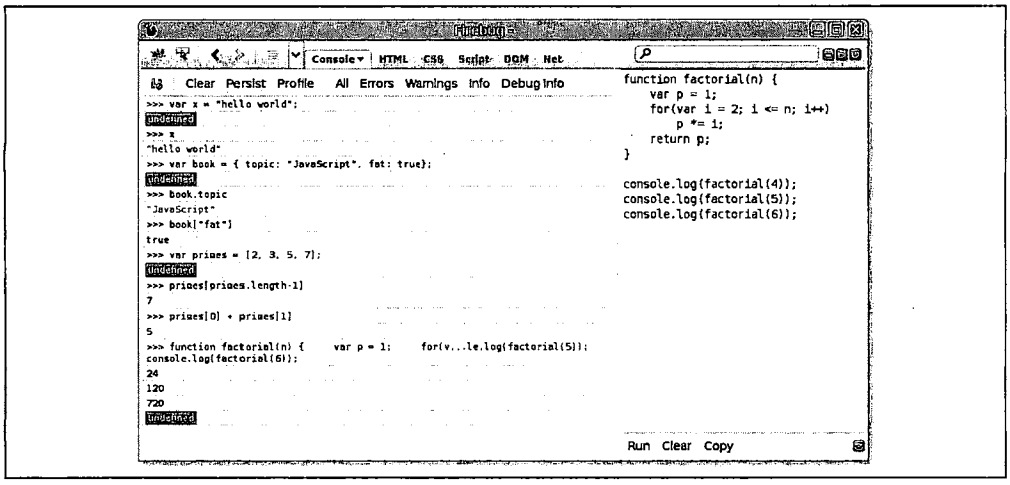

图1-1：Firebug控制台

1.1 JavaScript语言核心

本节是JavaScript语言的一个快速概览，也是本书第一部分的快速概览。在本章之后，我们将着重关注JavaScript的基础知识：第2章讲解JavaScript注释、分号和Unicode字符集；第3章会更加有意思，主要讲解JavaScript变量和赋值。这里有一些示例代码来说明这两章的重点内容：

```
// 所有在双斜线之后的内容都属于注释
// 仔细阅读这里的注释：它们对JavaScript代码做了解释

// 变量是表示值的一个符号名字
// 变量是通过var关键字声明的
var x;                  // 声明一个变量x

// 值可以通过等号赋值给变量
x = 0;                  // 现在变量x的值为0
x                       // => 0:通过变量获取其值

// JavaScript支持多种数据类型
x = 1;                  // 数字
x = 0.01;               // 整数和实数共用一种数据类型
x = "hello world";      // 由双引号内的文本构成的字符串
x = 'JavaScript';       // 单引号内的文本同样构成字符串
x = true;               // 布尔值
x = false;              // 另一个布尔值
x = null;               // null是一个特殊的值，意思是"空"
x = undefined;          // undefined和null非常类似
```

c++/C#: 双引号
Delphi: 单引号
Python/JS: 均可

JavaScript中两个非常重要的数据类型是对象和数组。第6章介绍对象，第7章介绍数组，对象和数组在JavaScript中是如此之重要，以至于你在本书中处处都能看到它们的身影。

```
//JavaScript中的最重要的类型就是对象
//对象是名/值对的集合，或字符串到值映射的集合
var book = {                    // 对象是由花括号括起来的
    topic: "JavaScript",        // 属性"topic"的值是"JavaScript"
    fat: true                   // 属性"fat"的值是true
};                              // 右花括号标记了对象的结束

// 通过"."或" []"来访问对象属性
book.topic                      // => "JavaScript"
book["fat"]                     // => true:另外一种获取属性的方式
book.author = "Flanagan";       // 通过赋值创建一个新属性
book.contents = {};             // {} 是一个空对象，它没有属性

// JavaScript同样支持数组（以数字为索引的列表）
var primes = [2, 3, 5, 7];      // 拥有4个值的数组，由"["和"]"划定边界
primes[0]                       // => 2: 数组中的第一个元素（索引为0）
primes.length                   // => 4: 数组中的元素个数
primes[primes.length - 1]       // => 7:数组的最后一个元素
primes[4] = 9;                  // 通过赋值来添加新元素
primes[4] = 11;                 // 或通过赋值来改变已有的元素
var empty = [];                 // [] 是空数组，它具有0个元素
empty.length                    // => 0

// 数组和对象中都可以包含另一个数组或对象:
var points = [                  // 具有两个元素的数组
    {x: 0, y: 0},               // 每个元素都是一个对象
    {x: 1, y: 1}
];
var data = {                    // 一个包含两个属性的对象
    trial1: [[1, 2],[3, 4]],    // 每一个属性都是数组
    trial2: [[2, 3],[4, 5]]     // 数组的元素也是数组
};
```

与JSON一样, {}表示对象, []表示数组

格式类似：
1. JSON
2. Python Dictionary

上段代码中通过方括号定义数组元素和通过花括号定义对象属性名和属性值之间的映射关系的语法称为初始化表达式（initializer expression），第4章有专门的介绍。表达式是JavaScript中的一个短语，这个短语可以通过运算得出一个值。通过“.”和“[]”来引用对象属性或数组元素的值就构成一个表达式。比如，请看一下上述代码中独占一行的表达式，其后的注释中箭头（=>）后的值就是表达式的运算结果。这种写法是本书中的一种约定表述方式。

JavaScript中最常见的表达式写法是像下面代码这样使用运算符（operator）：

```
// 运算符作用于操作数，生成一个新的值
// 最常见的是算术运算符
3 + 2                           // => 5: 加法
3 - 2                           // => 1: 减法
3 * 2                           // => 6: 乘法
3 / 2                           // => 1.5: 除法
points[1].x - points[0].x       // => 1: 更复杂的操作数也能照常工作
"3" + "2"                       // => "32": + 可以完成加法运算也可以作字符串连接

// JavaScript定义了一些算术运算符的简写形式
```

```
var count = 0;                  // 定义一个变量
count++;                        // 自增1
count--;                        // 自减1
count += 2;                     // 自增2：和"count = count + 2;"写法一样
count *= 3;                     // 自乘3：和"count = count * 3;"写法一样
count                           // => 6: 变量名本身也是一个表达式

// 相等关系运算符用来判断两值是否相等
// 不等、大于、小于运算符的运算结果是true或false
var x = 2, y = 3;               // 这里的 = 等号是赋值的意思，不是比较相等
x == y                          // => false: 相等
x != y                          // => true: 不等
x < y                           // => true: 小于
x <= y                          // => true: 小于等于
x > y                           // => false: 大于等于
x >= y                          // => false: 大于等于
"two" == "three"                // => false: 两个字符串不相等
"two" > "three"                 // => true: "tw"在字母表中的索引大于"th"
false == (x > y)                // => true: false和false相等

// 逻辑运算符是对布尔值的合并或求反
(x == 2) && (y == 3)            // => true: 两个比较都是true，&&表示"与"
(x > 3) || (y < 3)              // => false: 两个比较不都是true，||表示"或"
!(x == y)                       // => true: ! 求反
```

如果JavaScript中的“短语”是表达式的话，那么整个句子就称做语句（statement），第5章会详细讲解。在上述代码中，以分号结束的行均是一条语句（下面的代码中，会看到省略分号的多行语句）。实际上，语句和表达式之间有很多共同之处，粗略地讲，表达式仅仅计算出一个值但并不作任何操作，它并不改变程序的运行状态。而语句并不包含一个值（或者说它包含的值我们并不关心），但它们改变程序的运行状态。在上文中已经见过变量声明语句和赋值语句。另一类语句是“控制结构”（control structure），比如条件判断和循环。在介绍完函数之后，我们给出相关的示例代码。

函数是带有名称（named）[译注2]和参数的JavaScript代码段，可以一次定义多次调用。第8章会正式详细地讲解函数。与对象和数组一样，在本书的很多地方都会提到函数。这里是一些简单的示例代码：

```
// 函数是一段带有参数的JavaScript代码端，可以多次调用
function plus1(x) {             // 定义了名为plus1的一个函数，带有参数x
    return x+1;                 // 返回一个比传入的参数大的值
}                               // 函数的代码块是由花括号包裹起来的部分
plus1(y)                                    // => 4: y为3，调用函数的结果为 3+1

var square = function(x) {                  // 函数是一种值，可以赋值给变量
    return x*x;                             // 计算函数的值
};                                          // 分号标识了赋值语句的结束
```

译注2：这里“名称”含义是指函数具有固定标识，并不是指函数变量名称。

```
square(plus1(y))                    // => 16: 在一个表达式中调用两个函数
```

当将函数和对象合写在一起时，函数就变成了“方法”（method）：

```
// 当函数赋值给对象的属性，我们称为
//  "方法"，所有的JavaScript对象都含有方法
var a = [];                         // 创建一个空数组
a.push(1, 2, 3);                    // push()方法向数组中添加元素
a.reverse();                        // 另一个方法：将数组元素的次序反转

// 我们也可以定义自己的方法，"this"关键字是对定义方法
// 的对象的引用：这里的例子是上文中提到的包含两个点位置信息的数组
points.dist = function() {          // 定义一个方法用来计算两点之间的距离
    var p1 = this[0];               // 通过this获得对当前数组的引用
    var p2 = this[1];               // 并取得调用的数组前两个元素
    var a = p2.x - p1.x;            // X坐标轴上的距离
    var b = p2.y - p1.y;            // Y坐标轴上的距离
    return Math.sqrt(a * a +        // 勾股定理
    我们称为b * b);                 // 用Math.sqrt()来计算平方根
};
points.dist()                       // => 1.414: 求得两个点之间的距离
```

现在，给出一些控制语句的例子，这里的示例函数体内包含了最常见的JavaScript控制语句：

```
// 这些JavaScript语句使用该语法包含条件判断和循环
// 使用了类似C、C++、Java和其他语言的语法
function abs(x) {                   // 求绝对值的函数
    if (x >= 0) {                   // if语句...
        return x;                   // 如果比较结果为true则执行这里的代码.
    }                               // 子句的结束.
    else {                          // 当if条件不满足时执行else子句
        return - x;
    }                               // 如果分支中只有一条语句，花括号是可以省略的
}                                   // 注意if/else中嵌套的return语句

function factorial(n) {             // 计算阶乘的函数
    var product = 1;                // 给product赋值为1
    while (n > 1) {                 // 当()内的表达式为true时循环执行{}内的代码
        product *= n;               // "product = product * n; "的简写形式
        n--;                        //  "n = n - 1; "的简写形式
    }                               // 循环结束
    return product;                 // 返回product
}

factorial(4)                        // => 24: 1*4*3*2
function factorial2(n) {            // 实现循环的另一种写法
    var i, product = 1;             // 给product赋值为1
    for (i = 2; i <= n; i++)        // 将i从2自增至n
         product *= i;              // 循环体，当循环体中只有一句代码，可以省略{}
    return product;                 // 返回计算好的阶乘
}
factorial2(5)                       // => 120: 1*2*3*4*5
```

JavaScript是一种面向对象的编程语言，但和传统的面向对象又有很大区别。第9章将详细讲解JavaScript中的面向对象编程，这一章有大量的示例代码，是本书中最长的一章。这里有一个简单的示例，这段代码展示了如何在JavaScript中定义一个类来表示2D平面几何中的点。 这个类实例化的对象拥有一个名为r()的方法，用来计算该点到原点的距离：

```
// 定义一个构造函数以初始化一个新的Point对象
function Point(x, y) {                  // 按照惯例，构造函数均以大写字母开始
    this.x = x;                         // 关键字this指代初始化的实例
    this.y = y;                         // 将函数参数存储为对象的属性
}                                       // 不需要return

// 使用new关键字和构造函数来创建一个实例
var p = new Point(1, 1); // 平面几何中的点 (1,1)

// 通过给构造函数的prototye对象赋值
// 来给Point对象定义方法
Point.prototype.r = function() {
    return Math.sqrt(                   // 返回 x² + y² 的平方根
    this.x * this.x +                   // this指代调用这个方法的对象
    this.y * this.y);
};

// Point的实例对象p（以及所有的Point实例对象）继承了方法 r()
p.r()                                   // => 1.414...
```

第9章是第一部分的精华所在，后续的各章做了一些零星的延伸，将我们对JavaScript语言核心的探索带向尾声。第10章主要讲解了正则表达式的语法，并演示了如何使用这些“正则表达式”进行文本的模式匹配。第11章介绍JavaScript语言核心的子集和超集。最后，在进入客户端 JavaScript的内容之前，第12章介绍两种在Web浏览器之外的两种JavaScirpt运行环境。

1.2 客户端JavaScript

JavaScript语言核心部分的内容中的知识点交叉引用比较多，且知识点的层次感并不分明。而在客户端JavaScript部分的内容编排方式有了较大改变。依照本书给定的知识点顺序进行学习，完全可以学会如何在Web浏览器中使用JavaScript。但如果你想通过阅读本书来学习客户端 JavaScript的话，不能只将眼光落在第二部分，所以本节会对于客户端编程技术做一个快速概览，随后会给出一个有深度的示例。

第13章是第二部分的第一章，该章介绍如何让JavaScript在Web浏览器中运行起来。从该章学到的最重要的内容是，JavaScript代码可以通过<scirpt>标签来嵌入到HTML文件中：

```
<html>
<head>
<script src="library.js"></script> <!-- 引入一个JavaScript库-->
</head>
<body>
<p>This is a paragraph of HTML</p>
<script>
// 在这里编写嵌入到HTML文件中的JavaScript代码
</script>
<p>Here is more HTML.</P>
</body>
</html>
```

第14章讲解Web浏览器端脚本技术，并涵盖客户端JavaScript中的一些重要全局函数，例如：

```
<script>
function moveon() {
    // 通过弹出一个对话框来询问用户一个问题
    var answer = confirm("准备好了吗?");
    // 单击"确定"按钮，浏览器会加载一个新页面
    if (answer) window.location = "http://taobao.com";
}
// 在1分钟（6万毫秒）后执行定义的这个函数
setTimeout(moveon, 60000);
</script>
```

我们注意到，本节展示的客户端示例代码要比前面的示例代码要长很多。这里的示例代码并不是用来在Firebug（或者其他调试工具）控制台窗口中直接输入的，而是作为一个单独的HTML文件，并在Web浏览器中直接打开运行的。比如，上述代码段就是一个HTML文件的完整内容。

第15章的内容更加务实——通过脚本来操纵HTML文档内容。它将展示如何选取特定的HTML元素、如何给HTML元素设置属性、如何修改元素内容，以及如何给文档添加新节点。这里的示例函数展示了如何查找和修改基本文档的内容：

```
// 在document中的一个指定的区域输出调试消息
// 如果document不存在这样一个区域，则创建一个
function debug(msg) {
    // 通过查看HTML元素id属性来查找文档的调试部分
    var log = document.getElementById("debuglog");

    // 如果这个元素不存在，则创建一个
    if (!log) {
        log = document.createElement("div");    // 创建一个新的<div>元素
        log.id = "debuglog";                    // 给这个元素的HTML id赋值
        log.innerHTML = "<h1>Debug Log</h1>";   // 定义初始内容
        document.body.appendChild(log);         // 将其添加到文档的末尾
    }
```

```
    // 将消息包装在<pre>中，并添加至log中
    var pre = document.createElement("pre");    // 创建<pre>标签
    var text = document.createTextNode(msg);    // 将msg包装在一个文本节点中
    pre.appendChild(text);                      // 将文本添加至<pre>
    log.appendChild(pre);                       // 将<pre>添加至log
}
```

第15章讲述JavaScript如何操纵HTML中定义Web内容的元素。第16章讲述如何使用JavaScript来进行CSS样式操作，CSS样式定义了内容的展示方式。这通常会使用到HTML元素的style和class属性：

```
function hide(e, reflow) {                  // 通过JavaScript操纵样式来隐藏元素e
    if (reflow) {                           // 如果第二个参数是true
        e.style.display = "none"            // 隐藏这个元素，其所占的空间也随之消失
    }
    else {                                  // 否则
        e.style.visibility = "hidden"; // 将e隐藏，但是保留其所占的空间
    }
}
function highlight(e) {                     // 通过设置CSS类来高亮显示e
    // 简单地定义或追加HTML类属性
    // 这里假设CSS样式表中已经有"hilite"类的定义
    if (!e.className) e.className = "hilite";
    else e.className += " hilite";
}
```

可以通过JavaScript来操控Web浏览器中的HTML内容和文档的CSS样式，同样，也可以通过事件处理程序（event handler）来定义文档的行为。事件处理程序是一个在浏览器中注册的JavaScript函数，当特定类型的事件发生时浏览器便调用这个函数。通常我们关心的事件类型是鼠标点击事件和键盘按键事件（在智能手机中则是各种触碰事件）。或者说，当浏览器完成了文档的加载，当用户改变窗口大小或当用户向 HTML表单元素中输入数据时便会触发一个事件。第17章详细描述如何定义、注册事件处理程序，以及在事件发生时浏览器是如何调用它们的。

定义事件处理程序最简单的方法是，给HTML的以“on”为前缀的属性绑定一个回调。当写一些简单的测试程序时，最实用的方法就是给“onclick”处理程序绑定回调。假定已经将上文中的debug()和hide()两个函数保存至名为*debug.js*和*hide.js*的文件中，那么就可以写一个简单的HTML测试文件，来给<button>元素的onclick属性指定一个事件处理程序：

```
<script src="debug.js"></script>
<script src="hide.js"></script>
Hello
<button onclick="hide(this,true); debug('hide button 1');">Hide1</button>
<button onclick="hide(this); debug('hide button 2');">Hide2</button>
World
```

下面这些客户端JavaScript代码用到了事件，它给一个很重要的事件——“load”事件注册了一个事件处理程序。同时，也展示了注册“click”事件处理函数更高级的一种方法：

```
// "load"事件只有在文档加载完成后才会触发
//通常需要等待load事件发生后才开始执行JavaScript代码
window.onload = function() { // 当文档加载完成时执行这里的代码
    // 找到文档中所有的<img>标签
    var images = document.getElementsByTagName("img");

    // 遍历 images，给每个节点的"click"事件添加事件处理程序
    // 在点击图片的时候将图片隐藏
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
        if (image.addEventListener)    // 注册事件处理程序的另一种方法
            image.addEventListener("click", hide, false);
        else                           // 兼容IE8及以前的版本
            image.attachEvent("onclick", hide);
    }

    // 这便是上面注册的事件处理函数
    function hide(event) { event.target.style.visibility = "hidden"; }
};
```

第15~17章讲述了如何使用JavaScript来操控网页的内容（HTML）、样式（CSS）以及行为（事件处理）。这些章所讨论的API多少有些复杂，且至今仍具有糟糕的浏览器兼容性。也正是由于这个原因，很多客户端JavaScript程序员选择使用“库”或“框架”来简化他们的编码工作。最流行的库非jQuery莫属。第19章将会详细介绍jQuery库。jQuery定义了一套灵巧易用的 API，用来操控文档内容、样式和行为。jQuery经过了完整的测试，在所有现代主流浏览器，甚至在IE6这种早期浏览器中都可以照常运行。

jQuery代码非常易于识别，因为它充分利用了一个名为$()的函数。这里用jQuery重写了上文中提到的debug()函数：

```
function debug(msg) {
    var log = $("#debuglog");          // 找到要显示msg的元素.
    if (log.length == 0) {             // 如果不存在则创建之
        log = $("<div id='debuglog'><h1>Debug Log</h1></div>");
        log.appendTo(document.body);   // 并将其追加到body里
    }
    log.append($("<pre/>").text(msg)); // 将msg包装在<pre>中，再追加到log里
}
```

目前我们所提到的第二部分的4章都是围绕网页展开讨论的。后续的4章将着眼点转向Web应用。这几章的内容并不是讨论如何通过编写操控内容、样式和行为的脚本使用Web浏览器来渲染文档；而是讲解如何将Web浏览器当做应用平台，并描述了用以支持更复杂精细的客户端Web应用的现代浏览器API。第18章讲解如何使用JavaScript来发起HTTP请求。第20章描述数据存储的机制以及客户端应用中的会话状态的保持。第21章

涵盖基于HTML的<canvas>标签的客户端API，用来进行任意形状图形的绘制。最后，第22章讲解HTML5所提供的新一代Web应用API。网络、存储、图形：这些都是Web浏览器提供的操作系统级的服务，它们定义了全新的跨平台的应用环境。如果你正在进行基于那些支持这些新API的浏览器的开发，这将是你作为客户端JavaScript程序员最激动人心的时刻。最后4章并没有太多示例代码，但下面的例子使用了这些新的API。

示例：一个JavaScript贷款计算器

本章最后展示一个例子，这个例子集中使用了诸多技术，展示了真实环境下的客户端JavaScript（包括HTML和CSS）编程。例1-1给出了一个简单的贷款计算器应用的代码，如图1-2所示。

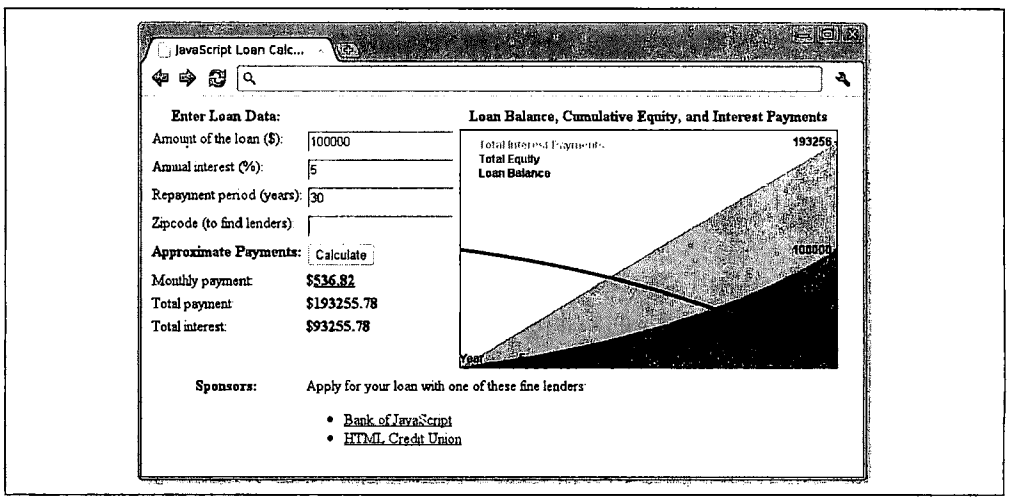

图1-2：一个贷款计算器Web应用

在看代码（例1-1）之前应当先仔细阅读本段文字。你不需要理解所有内容，代码中有着完整的注释，至少你应该能正确运行这段代码得到如图1-2所示的界面。这里的例子展示了诸多JavaScript语言核心特性，同样展示了重要的客户端JavaScript技术：

- 如何在文档中查找元素
- 如何通过表单input元素来获取用户的输入数据
- 如何通过文档元素来设置HTML内容
- 如何将数据存储在浏览器中
- 如何使用脚本发起HTTP请求
- 如何利用<canvas>元素绘图

例1-1：基于JavaScript实现的贷款计算器

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Loan Calculator</title>
<style> /* 这是一个CSS样式表：定义了程序输出的样式 */
.output    { font-weight: bold; }          /* 计算结果定义为粗体 */
#payment { text-decoration:underline; }    /* 定义 id="payment" 的元素样式 */
#graph { border: solid black 1px; }        /* 图表有一个1像素的边框 */
th, td { vertical-align: top;}             /* 表格单元格对其方式为顶端对齐 */
</style>
</head>
<body>
<!--
   这是一个HTML表格，其中包含<input>元素可以用来输入数据。
   程序将在<span>元素中显示计算结果，这些元素都具有类似"interset"和"years"的id
   这些id将在表格下面的JavaScript代码中用到。我们注意到，有一些
   input元素定义了"onchange"或"onclick"的事件处理程序，以便用户在输入数据或者点击inputs时
   执行指定的JavaScript代码段
-->
<table>
  <tr><th>Enter Loan Data:</th>
      <td></td>
      <th>Loan Balance, Cumulative Equity, and Interest Payments</th></tr>
  <tr><td>Amount of the loan ($):</td>
      <td><input id="amount" onchange="calculate();"></td>
      <td rowspan=8>
         <canvas id="graph" width="400" height="250"></canvas></td></tr>
  <tr><td>Annual interest (%):</td>
      <td><input id="apr" onchange="calculate();"></td></tr>
  <tr><td>Repayment period (years):</td>
      <td><input id="years" onchange="calculate();"></td>
  <tr><td>Zipcode (to find lenders):</td>
      <td><input id="zipcode" onchange="calculate();"></td>
  <tr><th>Approximate Payments:</th>
      <td><button onclick="calculate();">Calculate</button></td></tr>
  <tr><td>Monthly payment:</td>
      <td>$<span class="output" id="payment"></span></td></tr>
  <tr><td>Total payment:</td>
      <td>$<span class="output" id="total"></span></td></tr>
  <tr><td>Total interest:</td>
      <td>$<span class="output" id="totalinterest"></span></td></tr>
  <tr><th>Sponsors:</th><td colspan=2>
      Apply for your loan with one of these fine lenders:
      <div id="lenders"></div></td></tr>
</table>

<!-- 随后是JavaScirpt代码，这些代码内嵌在了一个<script>标签里 -->
<!-- 通常情况下，这些脚本代码应当放在<head>标签中 -->
<!-- 将JavaScript代码放在HTML代码之后仅仅是为了便于理解-->
<script>
"use strict"; // 如果浏览器支持的话，则开启ECMAScript 5的严格模式

/*
```

```
 * 这里的脚本定义了caculate()函数，在HTML代码中绑定事件处理程序时会调用它
 * 这个函数从<input>元素中读取数据，计算贷款赔付信息，并将结果显示在<span>元素中
 * 同样，这里还保存了用户数据、展示了放贷人链接并绘制出了图表
*/
function calculate() {
    //查找文档中用于输入输出的元素
    var amount = document.getElementById("amount");
    var apr = document.getElementById("apr");
    var years = document.getElementById("years");
    var zipcode = document.getElementById("zipcode");
    var payment = document.getElementById("payment");
    var total = document.getElementById("total");
    var totalinterest = document.getElementById("totalinterest");

    // 假设所有的输入都是合法的，将从input元素中获取输入数据
    //将百分比格式转换为小数格式，并从年利率转换为月利率
    //将年度赔付转换为月度赔付
    var principal = parseFloat(amount.value);
    var interest = parseFloat(apr.value) / 100 / 12;
    var payments = parseFloat(years.value) * 12;

    // 现在计算月度赔付的数据
    var x = Math.pow(1 + interest, payments); // Math.pow() 进行幂次运算
    var monthly = (principal * x * interest) / (x - 1);

    // 如果结果没有超过JavaScript能表示的数字范围，且用户的输入也正确
    // 这里所展示的结果就是合法的
    if (isFinite(monthly)) {
        // 将数据填充至输出字段的位置，四舍五入到小数点后两位数字
        payment.innerHTML = monthly.toFixed(2);
        total.innerHTML = (monthly * payments).toFixed(2);
        totalinterest.innerHTML = ((monthly * payments) - principal).toFixed(2);

        // 将用户的输入数据保存下来，这样在下次访问时也能取到数据
        save(amount.value, apr.value, years.value, zipcode.value);

        // 找到并展示本地放贷人，但忽略网络错误
        try { // 捕获这段代码抛出的所有异常
            getLenders(amount.value, apr.value, years.value, zipcode.value);
        }
        catch(e) { /* 忽略这些异常 */}

        // 最后，用图表展示贷款余额、利息和资产收益
        chart(principal, interest, monthly, payments);
    }
    else {
        // 计算结果不是数字或者是无穷大，意味着输入数据是非法或不完整的
        // 清空之前的输出数据
        payment.innerHTML = "";             // 清空元素的文本内容
        total.innerHTML = ""
        totalinterest.innerHTML = "";
        chart();                            // 不传参数的话就是清除图表
    }
}
```

```
// 将用户的输入保存至localStorage对象的属性中
// 这些属性在再次访问时还会继续保持在原位置
// 如果你在浏览器中按照file://URL的方式直接打开本地文件，
// 则无法在某些浏览器中使用存储功能（比如FireFox）
// 而通过HTTP打开文件是可行的
function save(amount, apr, years, zipcode) {
    if (window.localStorage) { // 只有在浏览器支持的时候才运行这里的代码
        localStorage.loan_amount = amount;
        localStorage.loan_apr = apr;
        localStorage.loan_years = years;
        localStorage.loan_zipcode = zipcode;
    }
}

// 在文档首次加载时，将会尝试还原输入字段
window.onload = function() {
    // 如果浏览器支持本地存储并且上次保存的值是存在的
    if (window.localStorage && localStorage.loan_amount) {
        document.getElementById("amount").value = localStorage.loan_amount;
        document.getElementById("apr").value = localStorage.loan_apr;
        document.getElementById("years").value = localStorage.loan_years;
        document.getElementById("zipcode").value = localStorage.loan_zipcode;
    }
};

// 将用户的输入发送至服务器端脚本（理论上）将
// 返回一个本地放贷人的链接列表，在这个例子中并没有实现这种查找放贷人的服务
// 但如果该服务存在，该函数会使用它
function getLenders(amount, apr, years, zipcode) {
    // 如果浏览器不支持XMLHttpRequest对象，则退出
    if (!window.XMLHttpRequest) return;

    // 找到要显示放贷人列表的元素
    var ad = document.getElementById("lenders");
    if (!ad) return;                          // 如果返回为空，则退出

    // 将用户的输入数据进行URL编码，并作为查询参数附加在URL里
    var url = "getLenders.php" +              // 处理数据的URL地址
    "?amt=" + encodeURIComponent(amount) + // 使用查询串中的数据
        "&apr=" + encodeURIComponent(apr) +
        "&yrs=" + encodeURIComponent(years) +
        "&zip=" + encodeURIComponent(zipcode);

    // 通过XMLHttpRequest对象来提取返回数据
    var req = new XMLHttpRequest();                  // 发起一个新的请求
    req.open("GET", url);                            // 通过URL发起一个HTTP GET请求
    req.send(null);                                  // 不带任何正文发送这个请求

    // 在返回数据之前，注册了一个事件处理函数，这个处理函数
    // 将会在服务器的响应返回至客户端的时候调用
    // 这种异步编程模型在客户端JavaScript中是非常常见的
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.status == 200) {
            // 如果代码运行到这里，说明我们得到了一个合法且完整的HTTP响应
            var response = req.responseText;       // HTTP响应是以字符串的形式呈现的
```

```
            var lenders = JSON.parse(response);      // 将其解析为JS数组

            // 将数组中的放贷人对象转换为HTML字符串形式
            var list = "";
            for (var i = 0; i < lenders.length; i++) {
                list += "<li><a href='" + lenders[i].url + "'>" +
                        lenders[i].name + "</a>";
            }

            // 将数据在HTML元素中呈现出来
            ad.innerHTML = "<ul>" + list + "</ul>";
        }
    }
}

// 在HTML<canvas>元素中用图表展示月度贷款余额、利息和资产收益
// 如果不传入参数的话，则清空之前的图表数据
function chart(principal, interest, monthly, payments) {
    var graph = document.getElementById("graph"); // 得到 <canvas> 标签
    graph.width = graph.width; // 用一种巧妙的手法清除并重置画布

    // 如果不传入参数，或者浏览器不支持画布，则直接返回
    if (arguments.length == 0 || !graph.getContext) return;

    // 获得画布元素的"context"对象，这个对象定义了一组绘画API
    var g = graph.getContext("2d");                  // 所有的绘画操作都将基于这个对象
    var width = graph.width,
    height = graph.height;                           // 获得画布大小

    // 这里的函数作用是将付款数字和美元数据转换为像素
    function paymentToX(n) {
        return n * width / payments;}
    function amountToY(a) {
        return height - (a * height / (monthly * payments * 1.05));}

    // 付款数据是一条从(0,0)到(payments, monthly*payments) 的直线
    g.moveTo(paymentToX(0), amountToY(0));           // 从左下方开始
    g.lineTo(paymentToX(payments),                   // 绘至右上方
    amountToY(monthly * payments));
    g.lineTo(paymentToX(payments), amountToY(0));    // 再至右下方
    g.closePath();                                   // 将结尾连接至开头
    g.fillStyle = "#f88";                            // 亮红色
    g.fill();                                        // 填充矩形
    g.font = "bold 12px sans-serif";                 // 定义一种字体
    g.fillText("Total Interest Payments", 20, 20);   // 将文字绘制到图例中

    // 很多资产数据并不是线性的，很难将其反映至图表中
    var equity = 0;
    g.beginPath();                                   // 开始绘制新图形
    g.moveTo(paymentToX(0), amountToY(0));           // 从左下方开始
    for (var p = 1; p <= payments; p++) {
        // 计算出每一笔赔付的利息
        var thisMonthsInterest = (principal - equity) * interest;
        equity += (monthly - thisMonthsInterest);    // 得到资产额
        g.lineTo(paymentToX(p), amountToY(equity)); // 将数据绘制到画布上
```

```
    }
    g.lineTo(paymentToX(payments), amountToY(0));   // 将数据线绘制至x轴
    g.closePath();                                  // 将线条结尾连接至线条开头
    g.fillStyle = "green";                          // 使用绿色绘制图形
    g.fill();                                       // 曲线之下的部分均填充
    g.fillText("Total Equity", 20, 35);             // 文本颜色设置为绿色

    // 再次循环，余额数据显示为黑色粗线条
    var bal = principal;
    g.beginPath();
    g.moveTo(paymentToX(0), amountToY(bal));
    for (var p = 1; p <= payments; p++) {
        var thisMonthsInterest = bal * interest;
        bal -= (monthly - thisMonthsInterest);      // 得到资产额
        g.lineTo(paymentToX(p), amountToY(bal));    // 将直线连接至某点
    }
    g.lineWidth = 3;                                // 将直线宽度加粗
    g.stroke();                                     // 绘制余额的曲线
    g.fillStyle = "black";                          // 使用黑色字体
    g.fillText("Loan Balance", 20, 50);             // 图例文字

    // 将年度数据在X轴做标记
    g.textAlign = "center";                                 // 文字居中对齐
    var y = amountToY(0);                                   // Y坐标设为0
    for (var year = 1; year * 12 <= payments; year++) {     // 遍历每年
        var x = paymentToX(year * 12);                      // 计算标记位置
        g.fillRect(x - 0.5, y - 3, 1, 3);                   // 开始绘制标记
        if (year == 1) g.fillText("Year", x, y - 5);        // 在坐标轴做标记
        if (year % 5 == 0 && year * 12 !== payments)        // 每5年的数据
           g.fillText(String(year), x, y - 5);
    }

    // 将赔付数额标记在右边界
    g.textAlign = "right";                          // 文字右对齐
    g.textBaseline = "middle";                      // 文字垂直居中
    var ticks = [monthly * payments, principal];    // 我们将要用到的两个点
    var rightEdge = paymentToX(payments);           // 设置X坐标
    for (var i = 0; i < ticks.length; i++) {        // 对每两个点做循环
        var y = amountToY(ticks[i]);                // 计算每个标记的Y坐标
        g.fillRect(rightEdge - 3, y - 0.5, 3, 1);           // 绘制标记
        g.fillText(String(ticks[i].toFixed(0)),             // 绘制文本
        rightEdge - 5, y);
    }
}
</script>
</body>
</html>
```