第 8 章　JavaScript 的调试与优化

编写 JavaScript 程序时或多或少会遇到各种各样的错误，有语法错误、逻辑错误等。短小的代码可以通过仔细检查来排除问题，但稍微大一点的程序，调试错误便是一件令开发者头疼的事情。另外，即使代码没有问题，对于大网站而言，执行的效率也是十分关键的，这就直接关系到代码的优化。

本章围绕 JavaScript 的错误处理和优化做简要的介绍，包括常见的错误和异常、调试的技巧、调试的工具、优化的细则等。

8.1　常见的错误和异常

错误主要分为语法错误和运行时的错误，前者在代码解析时就会产生，影响程序的进一步执行，后者通常称为异常，影响它所发生的线程。本节将简单地介绍 JavaScript 中的一些常见错误。

8.1.1　拼写错误

任何开发者在编写 JavaScript 程序时都犯过拼写错误，例如将 getElementsByTagName()漏写成 getElementByTagName()，将 getElementById()拼成 getElementByID()，等等。还有一些则是大小写的问题，在打字时顺手误按 Shift 键等，例如：

```
If(photo.href){
    var url = photo.href;
}
```

以上代码将关键字“if”拼成了“If”，导致了语法错误。这种拼写错误通常可以通过编写软件的语法高亮来发现，例如 Dreamweaver 的代码界面就会给关键字加粗、变色，如图 8.1 所示。

而另外一些变量上的拼写错误则稍微麻烦一些，通常需要开发者耐心地检查，如例 8.1 所示。

【例 8.1】拼写错误（光盘文件：第 8 章\8-1.html）

```
var PhotoAlbum = "isaac";
var PhotoAlbumWife = "fresheggs";
alert(  "the photos:\n" +
        PhotoAblum +
        "and" + PhotoAblumWife );
```

以上代码会提示 PhotoAblum 未定义，如图 8.2 所示。因为定义了“PhotoAlbum”，而不是“PhotoAblum”。

图 8.1　Dreamweaver 中的语法高亮

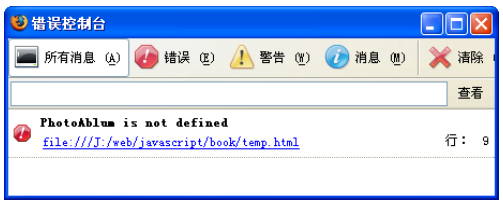


图 8.2　拼写错误

8.1.2 访问不存在的变量

最规范的变量定义通常使用关键字 var，但 JavaScript 允许不使用关键字而直接定义变量，例如下面两行代码都是合法的。

```
var isaac = "husband of fresheggs";
fresheggs = "wife of isaac";
```

这就无形中给错误检查增加了麻烦，像下面这几行代码的错误是比较容易发现的，浏览器也会相应地指出变量不存在的位置。

```
var isaac = "husband of fresheggs";
alert(fresheggs);
fresheggs = "wife of isaac";
```

但是很多时候这类型的错误却十分隐蔽，如例 8.2 所示的代码。

【例 8.2】访问不存在的变量（光盘文件：第 8 章\8-2.html）

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>访问不存在的变量</title>
<style>
<!--
#popup{
    position:absolute; width:202px;
    color:#004a7e; font-size:12px;
    font-family:Arial, Helvetica, sans-serif;
    left:41px; top:25px;
}
#popup.show{border:1px solid #004a7e;}
#popup.hide{border:none;}
li.mouseOver{
    background-color:#004a7e;
    color:#FFFFFF;
}
li.mouseOut{
    background-color:#FFFFFF;
    color:#004a7e;
}
-->
</style>
<script language="javascript">
var oInputField;
var oPopDiv;
var oColorsUl;
var aColors =
["red","green","blue","magenta","yellow","cornfloewrblue"];
function clearColors(){
    for(var i=oColorsUl.childNodes.length-1;i>=0;i--)
        oColorsUl.removeChild(oColorsUl.childNodes[i]);
    oPopDiv.className = "hide";
}
function setColors(the_clors){
    oInputField = document.forms["myForm1"].colors;
    oPopDiv = document.getElementById("popup");
    oColorsUl = document.getElementById("colors_ul");
    clearColors();
    oPopDiv.className = "show";
```

```
    var oLi;
    for(var i=0;i<aColors.length;i++){
        oLi = document.createElement("li");
        oColorsUl.appendChild(oLi);
        oLi.appendChild(document.createTextNode(the_colors[i]));

        oLi.onmouseover = function(){
            this.className = "mouseOver";
        }
        oLi.onmouseout = function(){
            this.className = "mouseOut";
        }
        oLi.onclick = function(){
            oInputField.value = this.firstChild.nodeValue;
            clearColors();
        }
    }
}
</script>
</head>
<body>
<form method="post" name="myForm1">
Color: <input type="text" name="colors" id="colors" onkeyup="setColors(aColors);" />
</form>
<div id="popup">
    <ul id="colors_ul"></ul>
</div>
</body>
</html>
```

浏览器会告知 the_colors 变量没有定义，如图 8.3 所示。

图 8.3 错误提示

但是仔细检测第 45 行却发现不了任何错误，如图 8.4 所示。

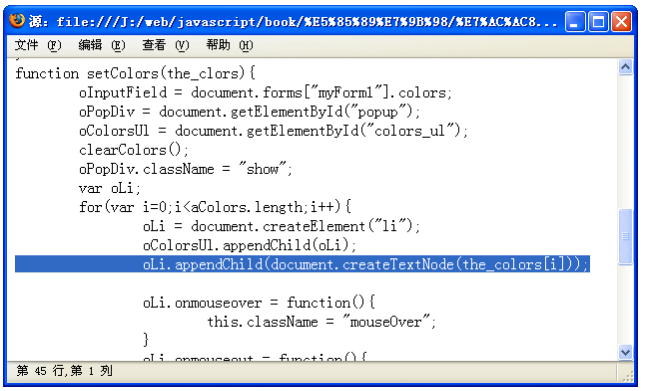


图 8.4 错误提示行无错误

因为真正的错误并不在第 45 行，而在第 35 行函数参数的拼写错误，如图 8.5 所示。像这

样提示与实际错误位置不相吻合的情况大大增加了调试代码的难度。

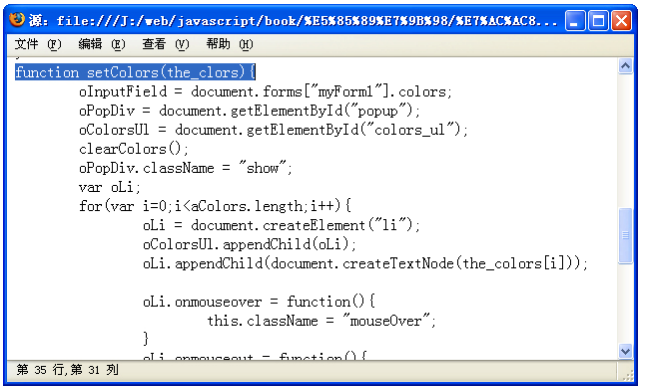


图 8.5 真实错误所在

8.1.3 括号不匹配

编写比较长的逻辑关系代码时常常需要反复使用花括号“{”、“}”和小括号“(”、“)”，很多时候因为删除某些代码，导致前括号和后括号的个数不匹配。这也是编写程序时经常犯的错误，即使是老程序员也不可避免。例如例 8.3 所示。

【例 8.3】括号不匹配（光盘文件：第 8 章\8-3.html）

```
<script language="javascript">
function testPic(x, start, end){
if(x<=end && x>=start){
if(x==start){
alert(x+" is the start of the range");
}
if(x==end){
alert(x+" is the end of the range");
}
if(x!=start && x!=end){
alert(x+" is in the range");
} else{
alert(x+" is not in the range");
}
}
testPic(7,5,8);
</script>
```

错误报告在第 21 行缺少“}”，如图 8.6 所示。这说明代码中缺少“}”，但行数不一定在第 21 行，必须仔细检查代码。

图 8.6 括号不匹配

本例真正漏后括号“}”的地方是第 16 行，正确代码如例 8.4 所示。

【例 8.4】正确的代码格式（光盘文件：第 8 章\8-4.html）

```
<script language="javascript">
function testPic(x, start, end){
    if(x<=end && x>=start){
        if(x==start){
            alert(x+" is the start of the range");
        }
        if(x==end){
            alert(x+" is the end of the range");
        }
        if(x!=start && x!=end){
            alert(x+" is in the range");
        }
    } else{
        alert(x+" is not in the range");
    }
}
testPic(7,5,8);
</script>
```

从正确的代码中可以看到，要避免括号的遗漏最有效的办法便是养成规范的编写习惯，适当地运用 Tab、空行等。

8.1.4 字符串和变量连接错误

在 JavaScript 输出结果时常常需要将字符串和变量相连接，因为加号和引号都比较多，常常容易错漏，如例 8.5 所示。

【例 8.5】连接错误（光盘文件：第 8 章\8-5.html）

```
<script language="javascript">
father = "isaac";
mother = "fresheggs";
child = "none";
family = father+" "+mother+" "child;
</script>
```

以上代码在设定字符串 family 时不小心漏写了一个加号，浏览器报错消息为缺少“;”，如图 8.7 所示。

还有一种典型的错误便是忽略了 JavaScript 的自动类型转换，如例 8.6 所示。

【例 8.6】JavaScript 的自动类型转换错误（光盘文件：第 8 章\8-6.html）

```
<script language="javascript">
a = 10;
b = 20;
c = 30;
alert("a+b+c="+a+b+c);
</script>
```

以上代码看上去很完美，也没有任何的语法错误，但运行结果如图 8.8 所示，这显然不是开发者想要的。

上述问题的解决办法是利用括号，在输出结果时将数值部分和字符串分别处理，最后再加到一块，如例 8.7 所示。

【例 8.7】使用括号避免自动类型转换（光盘文件：第 8 章\8-7.html）

```
<script language="javascript">
a = 10;
```

```
b = 20;
c = 30;
alert("a+b+c="+(a+b+c));
</script>
```

此时输出结果如图 8.9 所示，得到了正确的答案。JavaScript 类似这样自动类型转换的错误还很多，通常都是采用多加括号进行分离的办法来解决。

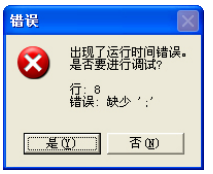


图 8.7 连接错误

图 8.8 连接错误

图 8.9 正确的连接

8.1.5 等号与赋值混淆

在 if 条件判断中，常常会把等号“==”误写成赋值符号“=”，而这样的错误在语法上没有任何问题，JavaScript 会把它处理成赋值是否成功来判断真假，例如：

```
if(isaac = "talking"){
    fresheggs.hear();
}
```

以上代码执行的结果是将变量 isaac 赋值为“talking”，赋值若成功，则执行花括号中的 fresheggs.hear()。但是显然开发者的本意是：

```
if(isaac == "talking"){
    fresheggs.hear();
}
```

即当 isaac 变量的值等于“talking”时运行 fresheggs.hear()。这样的错误在检查程序时是非常难发现的。

8.2 错误处理

了解错误是解决问题的一方面，在 JavaScript 中还提供了一些能够帮助开发者解决部分问题的方法。本节将简单介绍 JavaScript 调试错误的方法。

8.2.1 用 alert()和 document.write()方法监视变量值

alert()与 document.write()方法恐怕是实际调试程序中运用得最多也是最为有效的办法。在程序出错的地方或者代码运行的过程中添加合理的 alert()、document.write()命令来监视变量的值，对解决问题十分奏效。

alert()在弹出对话框显示变量值的同时，会停止代码的继续运行，直到用户单击“确定”按钮，而 document.write()则在输出值之后继续运行代码。这就给了开发者很大的空间来选择这两种方法。例如在如下代码中：

```
for(var i=0;i<aColors.length;i++)
    if(aColors[i].indexOf(oInputField.value) == 0)
        aResult.push(aColors[i]);
```

为了很好地检测加入到数组 aResult 里的值，往往在 if 语句中适当地嵌入 alert()语句。如果加入的值很多，则可以选用 document.write()方法，避免反复单击“确定”按钮，代码如下：

```
for(var i=0;i<aColors.length;i++)
    if(aColors[i].indexOf(oInputField.value) == 0){
        alert(aColors[i]);
        aResult.push(aColors[i]);
    }
```

8.2.2 用 onerror 事件找到错误

当页面出现异常时，error 事件会在 window 对象上触发，它能够在一定程度上告诉开发者出现了错误，并帮助开发者找到错误所在，如例 8.8 所示。

【例 8.8】理解 onerror 事件（光盘文件：第 8 章\8-8.html）

```
<html>
<head>
<title>onerror</title>
<script language="javascript">
window.onerror = function(){
    alert("出错啦！ ");
}
</script>
</head>
<body onload="nonExistent()">
</body>
</html>
```

代码运行时<body>标记的 onload 事件调用了一个不存在的函数 nonExistent()，产生了异常，如图 8.10 所示。

不过，这时浏览器本身的代码调试错误也出现了，如图 8.11 所示。

图 8.10 onerror 事件

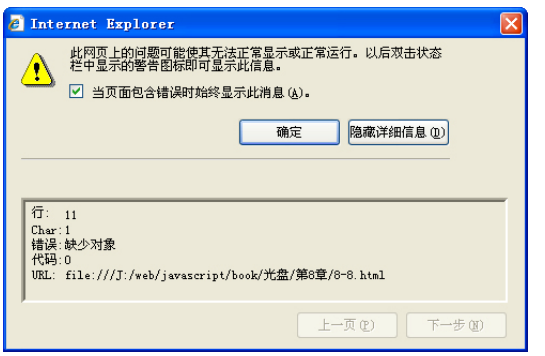


图 8.11 浏览器的错误信息

要避免浏览器自己的错误提示很简单，只需要在 onerror 事件的处理函数最后返回 true 便可，代码如例 8.9 所示。

【例 8.9】用 onerror 事件找错误（光盘文件：第 8 章\8-9.html）

```
<html>
<head>
<title>onerror</title>
<script language="javascript">
window.onerror = function(){
```

```
        alert("出错啦！");
        return true;      //屏蔽系统事件
    }
    </script>
    </head>
    <body onload="nonExistent()">
    </body>
    </html>
```

但是，这样处理之后对于解决错误并没有任何的帮助。其实 onerror 还提供了 3 个参数来确认错误的性质，代码如例 8.10 所示。

【例 8.10】用 onerror 事件的参数确认错误的性质（光盘文件：第 8 章\8-10.html）

```
<html>
<head>
<title>onerror</title>
<script language="javascript">
window.onerror = function(sMessage, sUrl, sLine){
    alert("出错啦:\n" + sMessage + "\nUrl: " + sUrl + "\n 行号: " + sLine);
    return true;      //屏蔽系统事件
}
</script>
</head>
<body onload="nonExistent()">
</body>
</html>
```

以上代码在 IE 7 和 Firefox 中的运行结果分别如图 8.12 和图 8.13 所示。

图 8.12　onerror 在 IE 中的参数

图 8.13　onerror 在 firefox 中的参数

注意：

在 IE 浏览器中发生 error 事件时，正常的代码会继续执行，所有的变量和数据都保存下来，并可通过 onerror 事件处理函数访问。而在 Firefox 中，正常的代码执行都会结束，同时所有错误发生之前的变量和数据都会被销毁。

8.2.3　用 try...catch 语句找到错误

在 ECMAScript 的第 3 版中，借鉴了 Java 语言的特点，引入了 try...catch 语句。该语句首先运行 try 里面的代码，如果发现错误则跳转到运行 catch()里面的代码。如果有 finally，则无论是否出错最后都运行其中的代码。try…catch 语句的语法如下：

```
try{
    //代码
} catch([exception]){
```

```
    //如果 try 中代码有错，则运行
} [finally{
    //最后运行，无论是否出错
}]
```

简单示例如例 8.11 所示。

【例 8.11】用 try...catch 语句找错误（光盘文件：第 8 章\8-11.html）

```
<html>
<head>
<title>try...catch</title>
<script language="javascript">
try{
    alert("this is an example");
    alert(fresheggs);
} catch(exception){
    var sError = "";
    for(var i in exception)
        sError += i + ":" + exception[i] + "\n";
    alert(sError);
}
</script>
</head>
<body>
</body>
</html>
```

以上代码在 try 中特意输出一个从未定义的变量 fresheggs，导致了异常，因此运行了 catch 中的代码段，在 IE 和 Firefox 中的输出结果分别如图 8.14 和图 8.15 所示。

图 8.14 IE 中的 try...catch

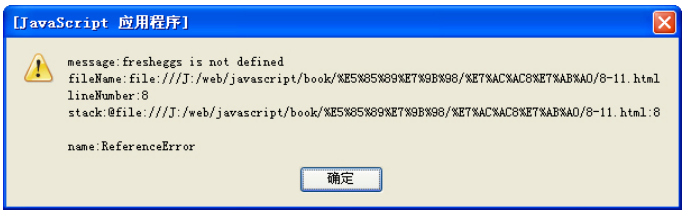


图 8.15 Firefox 中的 try...catch

通过 try...catch 可以很轻松地找到错误的问题，不过很可惜的是，该语句并不能很好地处理语法错误，如例 8.12 所示。

【例 8.12】try...catch 语句的局限性（光盘文件：第 8 章\8-12.html）

```
<html>
<head>
<title>try...catch</title>
<script language="javascript">
try{
    alert("this is an example") );
} catch(exception){
    var sError = "";
    for(var i in exception)
        sError += i + ":" + exception[i] + "\n";
    alert(sError);
}
</script>
</head>
<body>
```

```
</body>
</html>
```

例 8.12 中 try 语句的代码里出现了典型的括号不匹配错误，而整个代码并没有运行 catch 中的模块，而是浏览器弹出了错误提示框，如图 8.16 所示。

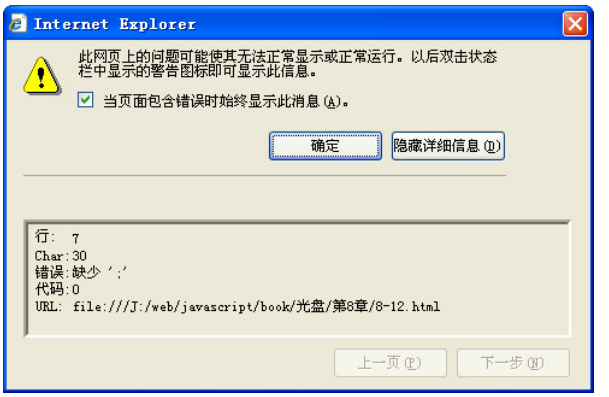


图 8.16　语法错误无能为力

8.3　使用调试器

程序语言的开发大都离不开代码的调试，尽管 JavaScript 本身不具备调试器，但 IE 和 Firefox 都有可以使用的调试器或调试器插件。本节主要介绍几个常用的调试器和调试方法，供读者参考。

8.3.1　用 Firefox 错误控制台调试

Firefox 的错误控制台是该浏览器自带的一个 JavaScript 调试器，而且十分有用。选择菜单栏中的“工具”→“错误控制台”命令便可以打开它，如图 8.17 所示。

所有在浏览器中运行时产生的错误、警告、消息都会传到错误控制台中，如图 8.18 所示。单击每一条记录都会打开相对应的代码，并高亮提示。

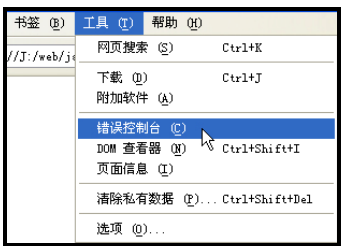


图 8.17　错误控制台

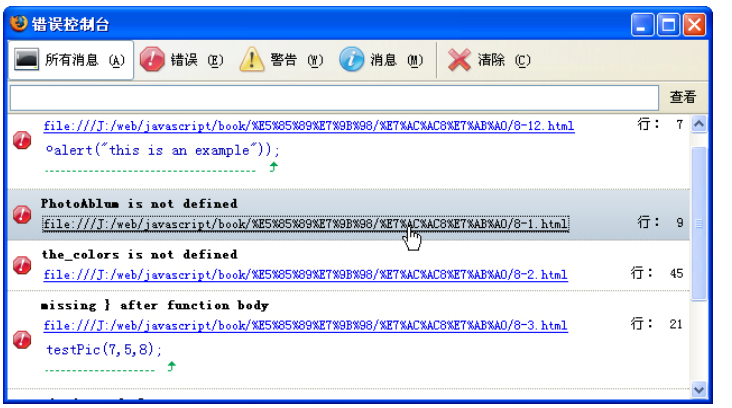


图 8.18　错误记录

Firefox 的错误控制台提供的错误提示相对于 IE 浏览器要全面而且准确得多，在没有特殊

调试需求时是一个非常好的选择。

8.3.2 用 Microsoft Script Debugger 调试

Microsoft Script Debugger 是 Microsoft 公司随 IE 4 一块发布的一个浏览器插件，可以从 Microsoft 的官方网站上免费下载。安装后必须把 IE 浏览器的调试选项打开才能使用，方法是：选择菜单栏中的“工具”→“Internet 选项”命令，打开“Internet 选项”对话框，选择“高级”选项卡，将“禁用脚本调试（Internet Explorer）”复选框中的勾去掉，如图 8.19 所示。

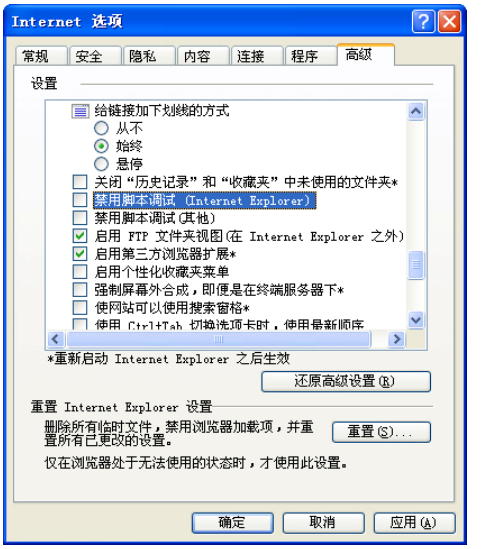


图 8.19 “高级”选项卡

当 IE 浏览器浏览页面时，可以随时运行该程序，以例 7.25 为例，在程序的 Running Document 面板里双击打开，可以看到文件以只读形式（Read only）开启，如图 8.20 所示。

在需要加入断点的地方按“F9”快捷键，或者单击工具栏中的 Toggle Breakpoint 按钮，这时要是再在 IE 浏览器中进行操作，便会在断点处停止，如图 8.21 所示。

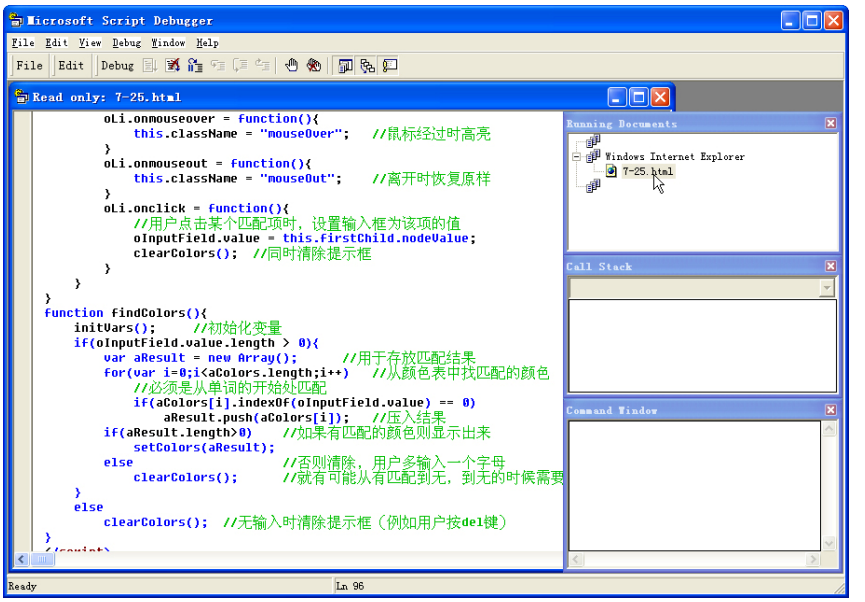


图 8.20 Microsoft Script Debugger

代码在断点处停止后便可以通过工具栏中的 Debug 栏上的按钮，像调试 C 语言一样进行调试，如图 8.22 所示。

另外，一个非常有用的功能是 Command Window，可以在代码断点停止时，在其中输入变量名称，直接按回车键显示出此时变量的值，它甚至可以接受简单的 JavaScript 命令，如图 8.23 所示。

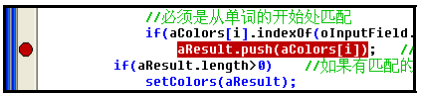


图 8.21 添加断点

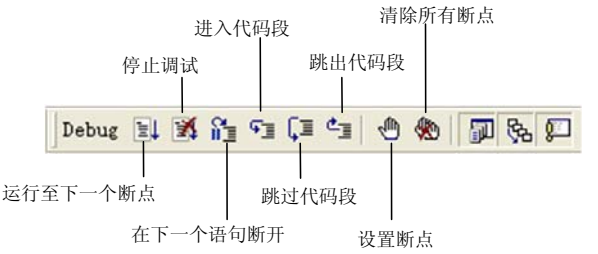


图 8.22 调试工具栏

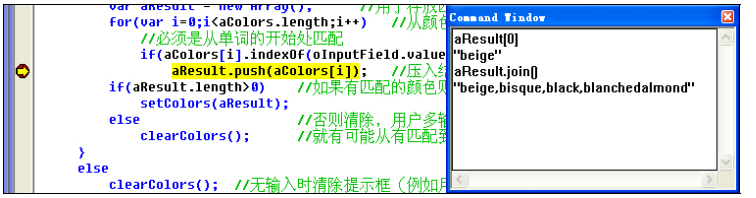


图 8.23 Command Window

当单击“停止调试”按钮或者程序运行完成时便退出整个调试过程。可以看到，该插件是一个针对 IE 浏览器十分有用的工具。

但是很讽刺的是，Microsoft Script Debugger 自身还存在一些问题，有时会导致整个 IE 浏览器死掉。不过只要重启浏览器便又能正常地使用。

8.3.3 用 Venkman 调试

Venkman 是一款针对 Mozilla 的脚本调试器，它的功能比 Microsoft Script Debugger 强大很多，在 Mozilla 的官方网站上可以免费下载。安装后在 Firefox 的菜单栏中的“工具”菜单中会多出“JavaScript Debugger”命令，运行该命令则会展开整个 Venkman 的界面，如图 8.24 所示。

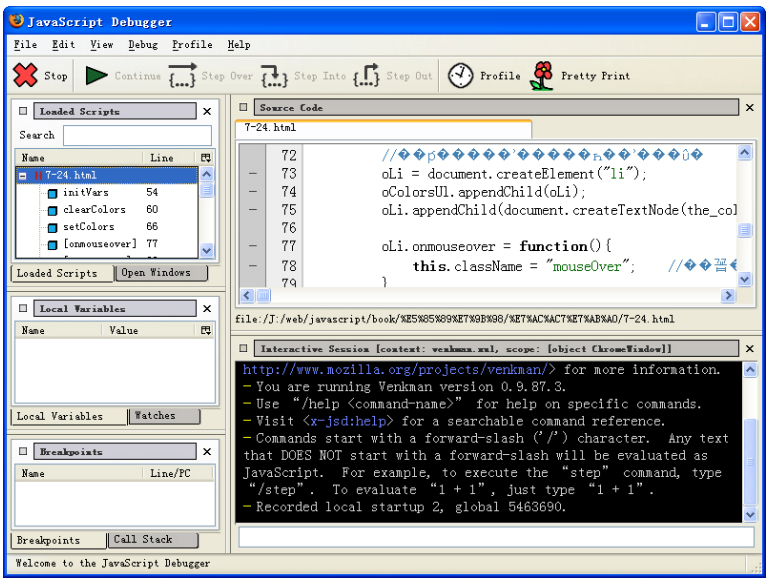


图 8.24 Venkman

Venkman 的面板很多，提供了方方面面的调试。这里仅介绍简单的断点调试，有兴趣的读者可以自行查看帮助文件。

仍然以例 7.25 为调试对象，在 Firefox 中运行该代码，然后打开 Venkman。在 Loaded Scripts 面板中双击 7-25.html 在 Source Code 面板中将其打开，可以看到 Loaded Scripts 面板中罗列出了所有的函数和事件，如图 8.25 所示。

在 Loaded Scripts 面板中，找到需要添加断点的代码，在行号前面单击鼠标便可以添加断点。Venkman 的断点分为两种，即普通的硬断点（hard breakpoint）和未来断点（future breakpoint）。单击一次鼠标添加红色的普通硬断点“B”，这种断点通常用于函数体内。单击两次鼠标则添加桔黄色的未来断点“F”，这种断点则用于函数体外，页面加载时就运行的代码，如图 8.26 所示。

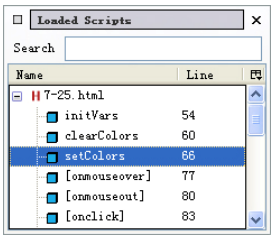


图 8.25 Loaded Scripts 面板

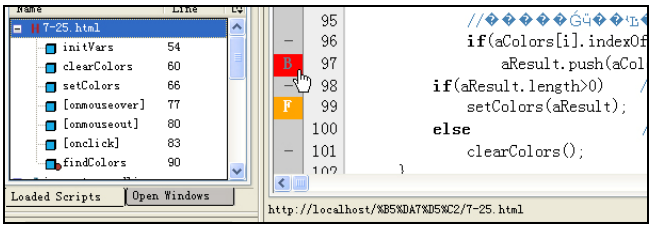


图 8.26 添加断点

在添加了断点之后，再在 Firefox 中执行相关操作，便可以看到 Venkman 的强大。在工具栏中各种按钮可以控制流程，与 Microsoft Script Debugger 差不多，而在 Local Variable 面板中则会自动加载相关的变量，并时时显示变量值，如图 8.27 所示。

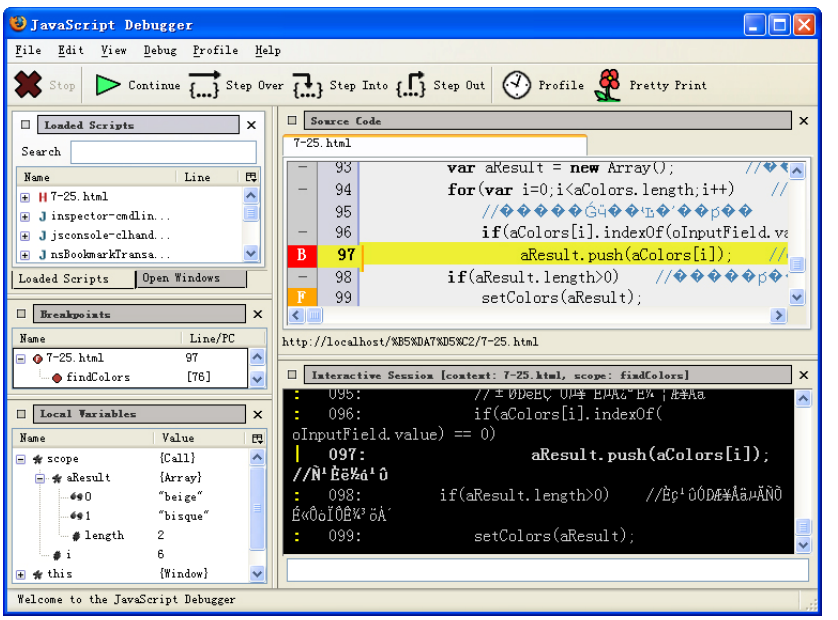


图 8.27 断点调试

更重要的是，Venkman 在运行时效率很高，并且整个插件运行稳定，是一款非常值得推荐的脚本调试器。

8.4 JavaScript 优化

JavaScript 是一门解释性的语言，它不像 C、Java 等程序设计语言，由编译器先进行编译再运行，而是直接下载到用户的客户端进行执行。因此代码本身的优劣直接决定了代码下载的速度以及执行的效率。本节主要介绍 JavaScript 的优化问题，包括代码下载的时间、优化的相关原则等。

8.4.1 减缓代码下载时间

Web 浏览器下载的是 JavaScript 的源码，其中包含的长变量名、注释、空格和换行等多余字符大大减缓了代码下载的时间。这些字符对于团队编写代码时十分有效，但在最后工程完成上传到服务器时，应当将它们全部删除。例如：

```
function showMeTheMoney(money)
{
      if (!money) {
         return false;
      } else {
         ...
      }
}
```

可以优化成：

```
function showMeTheMoney(money){if(!money){reutrn false;}else{...}}
```

这样，优化后节约了 25 个字节，倘若是一个大的 JavaScript 工程，将节省出非常大的空间，不但提高了用户下载的速度，也减轻了服务器的压力。

另外，对于布尔型的值 true 和 false，true 都可以用 1 来替换，而 false 可以用 0 来替换。对于 true 节省了 3 个字节，而 false 则节省了 4 个字节，例如：

```
var bSearch = false;
for(var i=0;i<aChoices.length && !bSearch;i++){
    if(aChoices[i]==vValue)
       bSearch = true;
}
```

替换成：

```
var bSearch = 0;
for(var i=0;i<aChoices.length && !bSearch;i++){
    if(aChoices[i]==vValue)
       bSearch = 1;
}
```

替换了布尔值之后，代码执行的效率、结果都相同，但节省了 7 个字节。

代码中常常会出现检测某个值是否为有效值的语句，而很多条件非的判断就是判断某个变量是否为“undefined”、“null”或者“false”，例如：

```
if(myValue != undefined){
      //...
}

if(myValue != null){
```

```
    //...
}

if(myValue != false){
    //...
}
```

这些虽然都正确，但采用逻辑非操作符“!”也可以有同样的效果，代码如下：

```
if(!myValue){
    //...
}
```

这样的替换也可以节省一部分字节，而且不太影响代码的可读性。类似的代码优化还有将数组定义时的 new Array()直接用“[]”代替，对象定义时的 new Object()用“{}”代替等，例如：

```
var myArray = new Array();
var myArray = [];
var myObject = new Object();
var myObject = {};
```

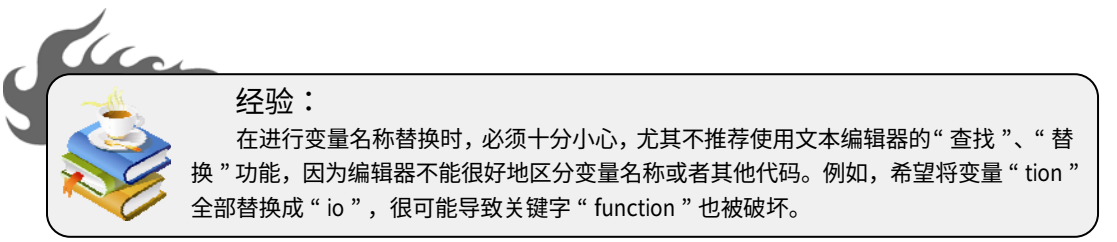
显然，第 2 行和第 4 行的代码较为精简，而且也很容易理解。

另外，在编写代码时往往为了提高可读性，函数名称、变量名称使用了很长的英文单词，同时也大大增加了代码的长度，例如：

```
function AddThreeVarsTogether(firstVar, secondVar, thirdVar){
    return (firstVar + secondVar + thirdVar);
}
```

可以优化成：

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



经验：

在进行变量名称替换时，必须十分小心，尤其不推荐使用文本编辑器的“查找”、“替换”功能，因为编辑器不能很好地区分变量名称或者其他代码。例如，希望将变量“tion”全部替换成“io”，很可能导致关键字“function”也被破坏。

对于上面介绍的这些减少代码体积的方法，有一些很实用的小工具可以自动完成类似的工作，例如 ECMAScript Cruncher、JSMin、Online JavaScript Compressor 等。有兴趣的读者可以自己试验，这里不再详细介绍。

8.4.2 合理声明变量

减少代码的体积仅仅只能使得用户下载的速度变快，但执行程序的速度并没有改变。要提高代码执行的效果，还得在各方面做调整。

在浏览器中，JavaScript 默认的变量范围是 window 对象，也就是全局变量。全局变量只有在浏览器关闭后才释放。而 JavaScript 也有局部变量，通常在 function 中执行完毕就会立即被释放。因此在函数体中要尽可能使用 var 关键字来声明变量，如例 8.13 所示。

【例 8.13】JavaScript 中的变量作用域（光盘文件：第 8 章\8-13.html）

```
<html>
<head>
```

```
<title>使用 var 声明局部变量</title>
<script language="javascript">
function First(){
    a = "字母";    //直接使用变量
}
function Second(){
    alert(a);
}
First();
Second();
</script>
</head>
<body>
</body>
</html>
```

以上代码在函数 First()中直接使用变量 a，而函数 Second()中没有声明，直接调用。运行的结果是函数 Second()顺利地读到了 a 变量的值，如图 8.28 所示。

这说明在函数 First()中，变量 a 被当成了一个全局变量，直到页面关闭时才会被销毁，浪费了不必要的资源。再看例 8.14 的代码。

【例 8.14】使用 var 声明局部变量（光盘文件：第 8 章\8-14.html）

```
<html>
<head>
<title>使用 var 声明局部变量</title>
<script language="javascript">
function First(){
    var a = "字母";    //用 var 声明
}
function Second(){
    alert(a);
}
First();
Second();
</script>
</head>
<body>
</body>
</html>
```

当在函数 First()中用 var 声明变量 a 后，在函数 Second()中便不能够再访问到 a，说明变量 a 被当成了一个局部变量，在执行完 First()后便立即被销毁了。如图 8.29 所示，浏览器提示变量 a 未定义。

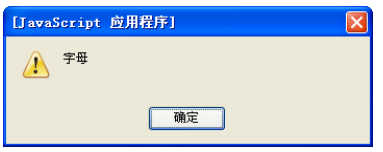


图 8.28　全局变量

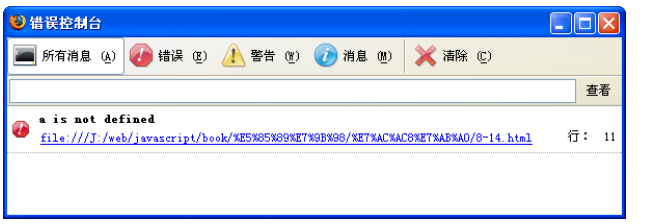


图 8.29　局部变量

因此，在函数体中，如果不是特别需要的全局变量，都应当使用 var 进行声明，从而节省系统资源。

8.4.3　使用内置函数缩短编译时间

只要可能，应当尽量使用 JavaScript 的内置函数。因为这些内置的属性、方法都是用类似 C、

C++之类的语言编译过的，运行起来比实时编译的 JavaScript 快很多。例如计算指数函数，可以自己编写如例 8.15 所示的代码。

【例 8.15】自定义函数（光盘文件：第 8 章\8-15.html）

```
<head>
<title>内置函数</title>
<script language="javascript">
function myPower(iNum, n){
    var iResult = iNum;
    for(var i=1;i<n;i++)
        iResult *= iNum;
    return iResult;
}
document.write(myPower(7,8));
</script>
</head>
```

以上代码自定义了一个 myPower()函数，虽然逻辑完全正确，但效率肯定比内置的函数 Math.pow()要低很多。为了证明这一点这里编写了一个比较程序，如例 8.16 所示。

【例 8.16】内置函数与自定义函数的速度比较（光盘文件：第 8 章\8-16.html）

```
<head>
<title>内置函数</title>
<script language="javascript">
function myPower(iNum, n){
    var iResult = iNum;
    for(var i=1;i<n;i++)
        iResult *= iNum;
    return iResult;
}
var myDate1 = new Date();
for(var i=0;i<150000;i++){
    myPower(7,8);              //自定义方法
}
var myDate2 = new Date();
document.write(myDate2-myDate1);
document.write("<br>");
myDate1 = new Date();
for(var i=0;i<150000;i++){
    Math.pow(7,8);          //采用系统内置方法
}
myDate2 = new Date();
document.write(myDate2-myDate1);
</script>
</head>
```

以上代码将自定义的方法 myPower()和系统内置的方法 Math.pow()各运算了 15 万次，然后分别计算运行时间。运行结果在 IE 和 Firefox 中如图 8.30 所示（不同的计算机运行速度会有差别），可以看到系统内置的方法要快很多。

图 8.30 使用系统内置函数

8.4.4 合理书写 if 语句

if 语句恐怕是所有代码中使用最频繁的，然而很可惜的是它执行的效率并不是很高。在用 if 语句和多个 else 语句时，一定要把最有可能的情况放在第一个，然后是可能性第二的，依次类推。例如预计到某个数值在 0～100 之间出现的概率最大，则可以这样安排代码：

```
if(iNum>0 && iNum<100){
    alert("在 0 和 100 之间");
} else if(iNum>99 && iNum<200){
    alert("在 100 和 200 之间");
} else if(iNum>199 && iNum<300){
    alert("在 200 和 300 之间");
} else{
    alert("小于等于 0 或者大于等于 300");
}
```

总是将出现概率最多的情况放在前面，这样就减少了进行多次测试后才能遇到正确条件的情况。当然也要尽可能减少使用 else if 语句，例如上面的代码还可以进一步优化成如下代码：

```
if(iNum>0){
    if(iNum<100){
       alert("在 0 和 100 之间");
    } else{
        if(iNum<200){
           alert("在 100 和 200 之间");
        } else{
            if(iNum<300){
                alert("在 200 和 300 之间");
            } else{
                alert("大于等于 300");
            }
        }
    }
} else{
    alert("小于等于 0");
}
```

以上代码看上去比较复杂，但因为考虑了很多代码潜在的判断、执行问题，因此执行速度要较前面的代码快。

另外，通常当超过两种情况时，最好能够使用 switch 语句。经常用 switch 语句替代 if 语句，可令执行速度快甚至 10 倍。另外，由于 case 语句可以使用任何类型，也大大方便了 switch 语句的编写。

8.4.5 最小化语句数量

脚本中的语句越少执行的时间就越短，而且代码的体积也会相应减小。例如用 var 语句定义变量时可以一次定义多个，代码如下：

```
var iNum = 365;
var sColor = "yellow";
var aMyNum = [8,7,12,3];
var oMyDate = new Date();
```

上面的多个定义可以用 var 关键字一次性定义，代码如下：

```
var iNum = 365, sColor = "yellow", aMyNum = [8,7,12,3], oMyDate = new Date();
```

同样在很多迭代运算的时候，也应该尽可能减少代码量，如下两行代码：

```
var sCar = aCars[i];
i++;
```

则可以优化成一句代码：

```
var sCar = aCars[i++];
```

8.4.6 节约使用 DOM

JavaScript 对 DOM 的处理可能是最耗费时间的操作之一。每次 JavaScript 对 DOM 的操作都会改变页面的表现，并重新渲染整个页面，从而有明显的时间消耗。比较快捷的方法就是尽可能不在页面中进行 DOM 操作，如下例中为 ul 添加了 10 个条目。

```
var oUl = document.getElementById("ulItem");
for (var i = 0; i < 10; i++) {
   var oLi = document.createElement("li");
   oUl.appendChild(oLi);
   oLi.appendChild(document.createTextNode("Item " + i);
}
```

以上代码在循环中调用 oUl.appendChild(oLi)，每次执行这条语句后，浏览器就会重新渲染页面。其次，给列表添加文本节点 oLi.appendChind(document.createTextNode("Item " + i)，这也会造成页面重新被渲染。因此每次运行都会造成两次重新渲染页面，共 20 次。

通常应当尽量减少 DOM 的操作，将列表项目在添加文本节点之后再添加，并合理地使用 createDocumentFragment()，代码如下：

```
var oUl = document.getElementById("ulItem");
var oTemp = document.createDocumentFragment();
for (var i = 0; i < 10; i++) {
   var oLi = document.createElement("li");
   oLi.appendChild(document.createTextNode("Item " + i);
   oTemp.appendChild(oLi);
}
oUl.appendChild(oTemp);
```