

第 17 章

错误处理与调试

本章内容

- 理解浏览器报告的错误
- 处理错误
- 调试 JavaScript 代码

由于 JavaScript 本身是动态语言，而且多年来一直没有固定的开发工具，因此人们普遍认为它是一种最难于调试的编程语言。脚本出错时，浏览器通常会给出类似于“object expected”（缺少对象）这样的消息，没有上下文信息，让人摸不着头脑。ECMAScript 第 3 版致力于解决这个问题，专门引入了 try-catch 和 throw 语句以及一些错误类型，意在让开发人员能够适当地处理错误。几年之后，Web 浏览器中也出现了一些 JavaScript 调试程序和工具。2008 年以来，大多数 Web 浏览器都已经具备了一些调试 JavaScript 代码的能力。

在有了语言特性和工具支持之后，现在的开发人员已经能够适当地实现错误处理，并且能够找到错误的根源。

17.1 浏览器报告的错误

IE、Firefox、Safari、Chrome 和 Opera 等主流浏览器，都具有某种向用户报告 JavaScript 错误的机制。默认情况下，所有浏览器都会隐藏此类信息，毕竟除了开发人员之外，很少有人关心这些内容。因此，在基于浏览器编写 JavaScript 脚本时，别忘了启用浏览器的 JavaScript 报告功能，以便及时收到错误通知。

17.1.1 IE

IE 是唯一一个在浏览器的界面窗体(chrome)中显示 JavaScript 错误信息的浏览器。在发生 JavaScript 错误时，浏览器左下角会出现一个黄色的图标，图标旁边则显示着“Error on page”（页面中有错误）。假如不是存心去看的话，你很可能不会注意这个图标。双击这个图标，就会看到一个包含错误消息的对话框，其中还包含诸如行号、字符数、错误代码及文件名（其实就是你在查看的页面的 URL）等相关信息。图 17-1 展示了 IE 的错误消息对话框。

这些信息对于一般用户还算说得过去，但对 Web 开发来说就远远不够了。可以通过设置让错误对话框一发生错误就显示出来。为此，要打开“Tools”（工具）菜单中的“Internet Options”（Internet 选项）对话框，切换到“Advanced”（高级）选项卡，选中“Display a notification about every script error”（显示每个脚本错误的通知）复选框（参见图 17-2）。单击“OK”（确定）按钮保存设置。

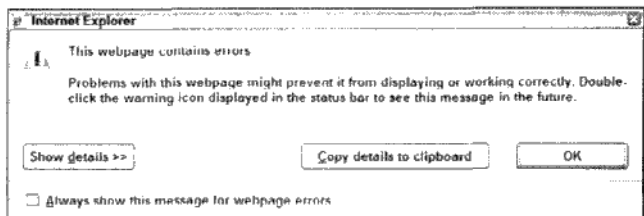


图 17-1

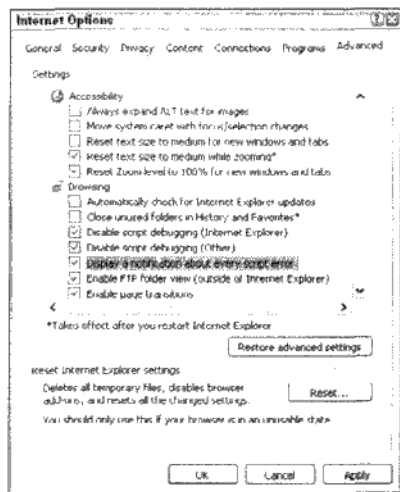


图 17-2

保存了设置之后，通常要双击黄色图标才会显示的对话框，就会变成一有错误发生随即自动显示出来。

另外，如果启用了脚本调试功能的话（默认是禁用的），那么在发生错误时，你不仅会显示错误通知，而且还会看到另一个对话框，询问是否想要调试错误（参见图 17-3）。

要启用脚本调试功能，必须要在 IE 中安装某种脚本调试器。（IE8 和 IE9 自带调试器。）本章后面会单独讨论调试器。

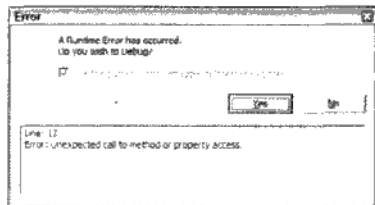


图 17-3



在 IE7 及更早版本中，如果错误发生在位于外部文件的脚本中，行号通常会与错误所在的行号差 1。如果是嵌入在页面中的脚本发生错误，则行号就是错误所在的行号。

17.1.2 Firefox

默认情况下，Firefox 在 JavaScript 发生错误时不会通过浏览器界面给出提示。但它会在后台将错误

记录到错误控制台中。单击“Tools”（工具）菜单中的“Error Console”（错误控制台）可以显示错误控制台（见图 17-4）。你会发现，错误控制台中实际上还包含与 JavaScript、CSS 和 HTML 相关的警告和信息，可以通过筛选找到错误。

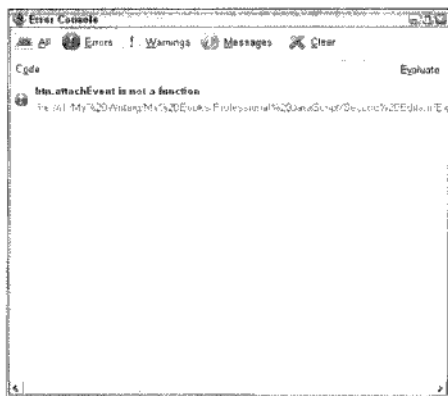


图 17-4

在发生 JavaScript 错误时, Firefox 会将其记录为一个错误, 包括错误消息、引发错误的 URL 和错误所在的行号等信息。单击文件名即可以只读方式打开发生错误的脚本, 发生错误的代码行会突出显示。

目前，最流行的 Firefox 插件 Firebug，已经成为开发人员必备的 JavaScript 纠错工具。这个可以从 www.getfirebug.com 下载到的插件，会在 Firefox 状态栏的右下角区域添加一个图标。默认情况下，右下角区域显示的是一个绿色对勾图标。在有 JavaScript 错误发生时，图标会变成红叉，同时旁边显示错误的数量。单击这个红叉会打开 Firebug 控制台，其中显示有错误消息、错误所在的代码行（不包含上下文）、错误所在的 URL 以及行号（参见图 17-5）。

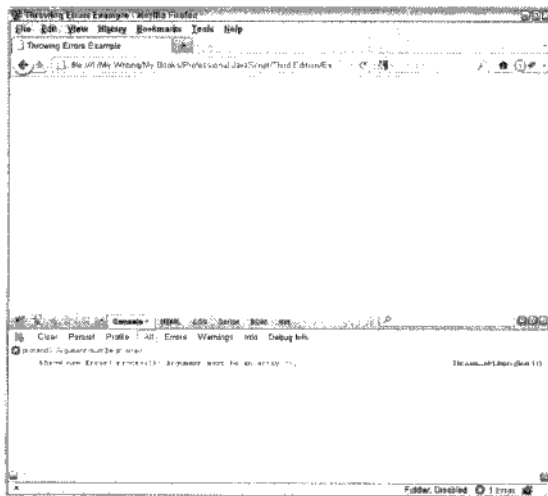


图 17-5

在 Firebug 中单击导致错误的代码行，将在一个新 Firebug 视图中打开整个脚本，该代码行在其中突出显示。



除了显示错误之外，Firebug 还有更多的用处。实际上，它还是针对 Firefox 的成熟的调试环境，为调试 JavaScript、CSS、DOM 和网络连接错误提供了诸多功能。

17.1.3 Safari

Windows 和 Mac OS 平台的 Safari 在默认情况下都会隐藏全部 JavaScript 错误。为了访问到这些信息，必须启用“Develop”（开发）菜单。为此，需要单击“Edit”（编辑）菜单中的“Preferences”（偏好设置），然后在“Advanced”（高级）选项卡中，选中“Show develop menu in menubar”（在菜单栏中显示“开发”菜单）。启用此项设置之后，就会在 Safari 的菜单栏中看到一个“Develop”菜单（参见图 17-6）。

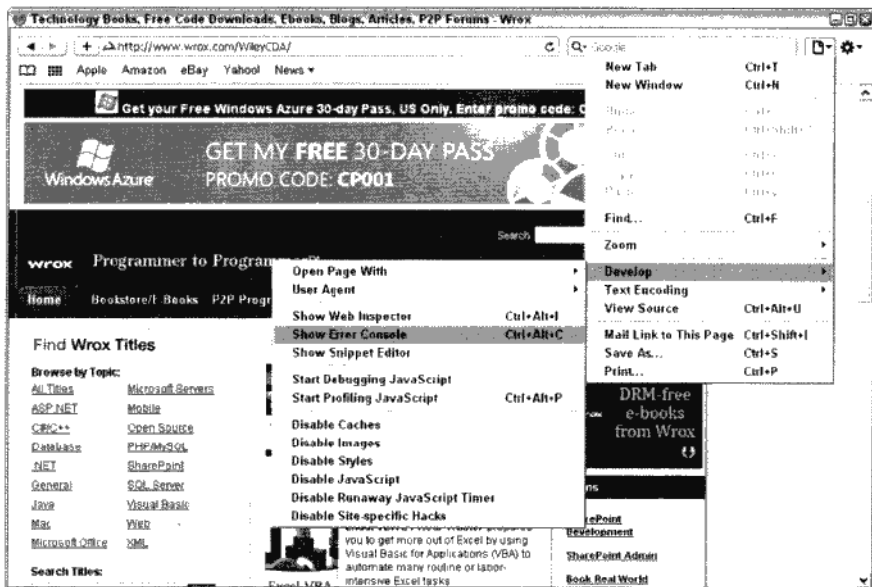


图 17-6

“Develop”菜单中提供了一些与调试有关的选项，还有一些选项可以影响当前加载的页面。单击“Show Error Console”（显示错误控制台）选项，将会看到一组 JavaScript 及其他错误。控制台中显示着错误消息、错误的 URL 及错误的行号（参见图 17-7）。

单击控制台中的错误消息，就可以打开导致错误的源代码。除了被输出到控制台之外，JavaScript 错误不会影响 Safari 窗口的外观。

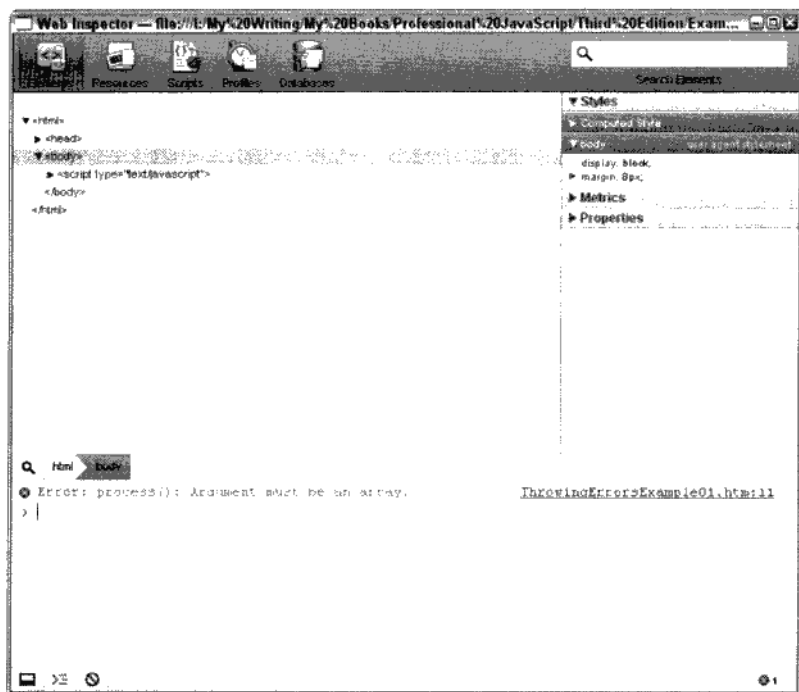


图 17-7

17.1.4 Opera

Opera 在默认情况下也会隐藏 JavaScript 错误，所有错误都会被记录到错误控制台中。要打开错误控制台，需要单击“Tools”（工具）菜单，在“Advanced”（高级）子菜单项下面再单击“Error Console”（错误控制台）。与 Firefox 一样，Opera 的错误控制台中也包含了除 JavaScript 错误之外的很多来源（如 HTML、CSS、XML、XSLT 等）的错误和警告信息。要分类查看不同来源的消息，可以使用左下角的下拉选择框（参见图 17-8）。

错误消息中显示着导致错误的 URL 和错误所在的线程。有时候，还会有栈跟踪信息。除了错误控制台中显示的信息之外，没有其他途径可以获得更多信息。

也可以让 Opera 一发生错误就弹出错误控制台。为此，要在“Tools”（工具）菜单中单击“Preferences”（首选项），再单击“Advanced”（高级）选项卡，然后从左侧菜单中选择“Content”（内容）。单击“JavaScript Options”（JavaScript 选项）按钮，显示选项对话框（如图 17-9 所示）。

在这个选项对话框中，选中“Open console on error”（出错时打开控制台），单击“OK”（确定）按钮。这样，每当发生 JavaScript 错误时，就会弹出错误控制台。另外，还可以针对特定的站点来作此设置，方法是单击“Tools”（工具）、“Quick Preferences”（快速参数）、“Edit Site Preferences”（编辑站点首选项），选择“Scripting”（脚本）选项卡，最后选中“Open console on error”（出错时打开控制台）。

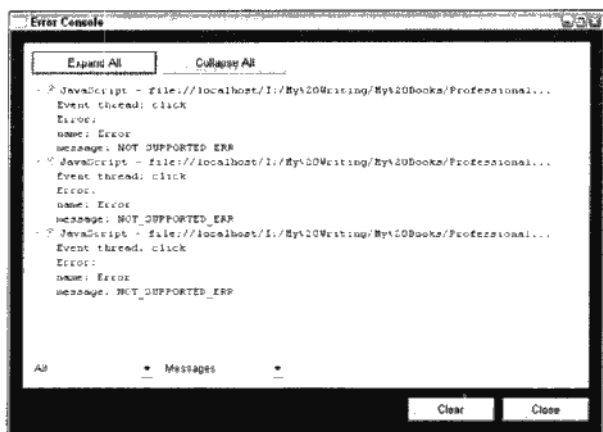


图 17-8

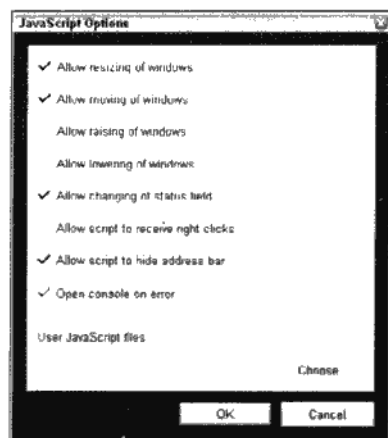


图 17-9

17.1.5 Chrome

与 Safari 和 Opera 一样，Chrome 在默认情况下也会隐藏 JavaScript 错误。所有错误都将被记录到 Web Inspector 控制台中。要查看错误消息，必须打开 Web Inspector。为此，要单击位于地址栏右侧的“Control this page”（控制当前页）按钮，选择“Developer”（开发人员）、“JavaScript console”（JavaScript 控制台），参见图 17-10。

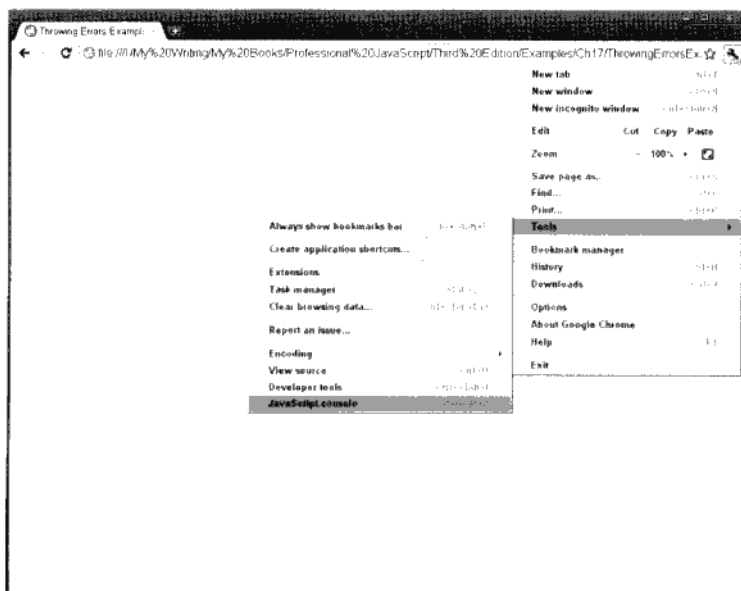


图 17-10

打开的 Web Inspector 中包含着有关页面的信息和 JavaScript 控制台。控制台中显示着错误消息、错误的 URL 和错误的行号（参见图 17-11）。

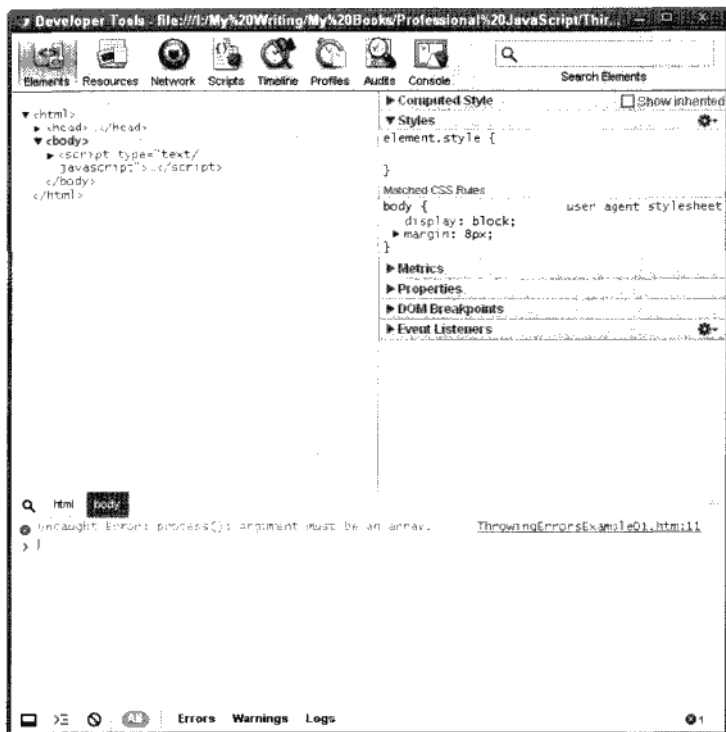


图 17-11

单击 JavaScript 控制台中的错误，就可以定位到导致错误的源代码行。

17.2 错误处理

错误处理在程序设计中的重要性是毋庸置疑的。任何有影响力的 Web 应用程序都需要一套完善的错误处理机制，当然，大多数佼佼者确实做到了这一点，但通常只有服务器端应用程序才能做到如此。实际上，服务器端团队往往会在错误处理机制上投入较大的精力，通常要考虑按照类型、频率，或者其他重要的标准对错误进行分类。这样一来，开发人员就能够理解用户在使用简单数据库查询或者报告生成脚本时，应用程序可能会出现的问题。

虽然客户端应用程序的错误处理也同样重要，但真正受到重视，还是最近几年的事。实际上，我们要面对这样一个不争的事实：使用 Web 的绝大多数人都不是技术高手，其中甚至有很多人根本就不明白浏览器到底是什么，更不用说让他们说喜欢哪一个了。本章前面讨论过，每个浏览器在发生 JavaScript 错误时的行为都或多或少有一些差异。有的会显示小图标，有的则什么动静也没有，浏览器对 JavaScript 错误的这些默认行为对最终用户而言，毫无规律可循。最理想的情况下，用户遇到错误搞不清为什么，

他们会再试着重做一次；最糟糕的情况下，用户会恼羞成怒，一去不复返了。良好的错误处理机制可以让用户及时得到提醒，知道到底发生了什么事，因而不会惊惶失措。为此，作为开发人员，我们必须理解在处理 JavaScript 错误的时候，都有哪些手段和工具可以利用。

17.2.1 try-catch 语句

ECMA-262 第 3 版引入了 try-catch 语句，作为 JavaScript 中处理异常的一种标准方式。基本的语法如下所示，显而易见，这与 Java 中的 try-catch 语句是完全相同的。

```
try{
    // 可能会导致错误的代码
} catch(error){
    // 在错误发生时怎么处理
}
```

也就是说，我们应该把所有可能会抛出错误的代码都放在 try 语句块中，而把那些用于错误处理的代码放在 catch 块中。例如：

```
try {
    window.someNonexistentFunction();
} catch (error){
    alert("An error happened!");
}
```

如果 try 块中的任何代码发生了错误，就会立即退出代码执行过程，然后接着执行 catch 块。此时，catch 块会接收到一个包含错误信息的对象。与在其他语言中不同的是，即使你不想使用这个错误对象，也要给它起个名字。这个对象中包含的实际信息会因浏览器而异，但共同的是有一个保存着错误消息的 message 属性。ECMA-262 还规定了一个保存错误类型的 name 属性；当前所有浏览器都支持这个属性（Opera 9 之前的版本不支持这个属性）。因此，在发生错误时，就可以像下面这样实事求是地显示浏览器给出的消息。

```
try {
    window.someNonexistentFunction();
} catch (error){
    alert(error.message);
}
```


TryCatchExample01.htm

这个例子在向用户显示错误消息时，使用了错误对象的 message 属性。这个 message 属性是唯一一个能够保证所有浏览器都支持的属性，除此之外，IE、Firefox、Safari、Chrome 以及 Opera 都为事件对象添加了其他相关信息。IE 添加了与 message 属性完全相同的 description 属性，还添加了保存着内部错误数量的 number 属性。Firefox 添加了 fileName、lineNumber 和 stack（包含栈跟踪信息）属性。Safari 添加了 line（表示行号）、sourceId（表示内部错误代码）和 sourceURL 属性。当然，在跨浏览器编程时，最好还是只使用 message 属性。

1. finally 子句

虽然在 try-catch 语句中是可选的，但 finally 子句一经使用，其代码无论如何都会执行。换句话说，try 语句块中的代码全部正常执行，finally 子句会执行；如果因为出错而执行了 catch 语句

块, finally 子句照样还会执行。只要代码中包含 finally 子句, 则无论 try 或 catch 语句块中包含什么代码——甚至 return 语句, 都不会阻止 finally 子句的执行。来看下面这个函数。




```
function testFinally(){
    try {
        return 2;
    } catch (error){
        return 1;
    } finally {
        return 0;
    }
}
```

[TryCatchExample02.htm](#)

这个函数在 try-catch 语句的每一部分都放了一条 return 语句。表面上看, 调用这个函数会返回 2, 因为返回 2 的 return 语句位于 try 语句块中, 而执行该语句又不会出错。可是, 由于最后还有一个 finally 子句, 结果就会导致该 return 语句被忽略; 也就是说, 调用这个函数只能返回 0。如果把 finally 子句拿掉, 这个函数将返回 2。

如果提供 finally 子句, 则 catch 子句就成了可选的 (catch 或 finally 有一个即可)。IE7 及更早版本中有一个 bug: 除非有 catch 子句, 否则 finally 中的代码永远不会执行。如果你仍然要考虑 IE 的早期版本, 那就只好提供一个 catch 子句, 哪怕里面什么都不写。IE8 修复了这个 bug。



请读者务必要记住, 只要代码中包含 finally 子句, 那么无论 try 还是 catch 语句块中的 return 语句都将被忽略。因此, 在使用 finally 子句之前, 一定要非常清楚你想让代码怎么样。

2. 错误类型

执行代码期间可能会发生的错误有多种类型。每种错误都有对应的错误类型, 而当错误发生时, 就会抛出相应类型的错误对象。ECMA-262 定义了下列 7 种错误类型:

- ☐ Error
- ☐ EvalError
- ☐ RangeError
- ☐ ReferenceError
- ☐ SyntaxError
- ☐ TypeError
- ☐ URIError

其中, Error 是基类型, 其他错误类型都继承自该类型。因此, 所有错误类型共享了一组相同的属性 (错误对象中的方法全是默认的对象方法)。Error 类型的错误很少见, 如果有也是浏览器抛出的; 这个基类型的主要目的是供开发人员抛出自定义错误。

EvalError 类型的错误会在使用 eval() 函数而发生异常时被抛出。ECMA-262 中对这个错误有如下描述: “如果以非直接调用的方式使用 eval 属性的值 (换句话说, 没有明确地将其名称作为作一个 Identifier, 即用作 CallExpression 中的 MemberExpression), 或者为 eval 属性赋值。” 简单地说, 如果没有把 eval() 当成函数调用, 就会抛出错误, 例如:

```
new eval();    //抛出 EvalError
eval = foo;    //抛出 EvalError
```

在实践中，浏览器不一定会在应该抛出错误时就抛出 `EvalError`。例如，Firefox 4+和 IE8 对第一种情况会抛出 `TypeError`，而第二种情况会成功执行，不发生错误。有鉴于此，加上在实际开发中极少会这样使用 `eval()`，所以遇到这种错误类型的可能性极小。

`RangeError` 类型的错误会在数值超出相应范围时触发。例如，在定义数组时，如果指定了数组不支持的项数（如 -20 或 `Number.MAX_VALUE`），就会触发这种错误。下面是具体的例子。

```
var items1 = new Array(-20);           //抛出 RangeError
var items2 = new Array(Number.MAX_VALUE); //抛出 RangeError
```

JavaScript 中经常会出现这种范围错误。

在找不到对象的情况下，会发生 `ReferenceError`（这种情况下，会直接导致人所共知的“object expected”浏览器错误）。通常，在访问不存在的变量时，就会发生这种错误，例如：

```
var obj = x;           //在 x 并未声明的情况下抛出 ReferenceError
```

至于 `SyntaxError`，当我们把语法错误的 JavaScript 字符串传入 `eval()` 函数时，就会导致此类错误。例如：

```
eval("a ++ b");        //抛出 SyntaxError
```

如果语法错误的代码出现在 `eval()` 函数之外，则不太可能使用 `SyntaxError`，因为此时的语法错误会导致 JavaScript 代码立即停止执行。

`TypeError` 类型在 JavaScript 中会经常用到，在变量中保存着意外的类型时，或者在访问不存在的方法时，都会导致这种错误。错误的原因虽然多种多样，但归根结底还是由于在执行特定于类型的操作时，变量的类型并不符合要求所致。下面来看几个例子。

```
var o = new 10;           //抛出 TypeError
alert("name" in true);    //抛出 TypeError
Function.prototype.toString.call("name"); //抛出 TypeError
```

最常发生类型错误的情况，就是传递给函数的参数事先未经检查，结果传入类型与预期类型不相符。

在使用 `encodeURIComponent()` 或 `decodeURIComponent()`，而 URI 格式不正确时，就会导致 `URIError` 错误。这种错误也很少见，因为前面说的这两个函数的容错性非常高。

利用不同的错误类型，可以获悉更多有关异常的信息，从而有助于对错误作出恰当的处理。要想知道错误的类型，可以像下面这样在 `try-catch` 语句的 `catch` 语句中使用 `instanceof` 操作符。

```
try {
    someFunction();
} catch (error) {
    if (error instanceof TypeError) {
        //处理类型错误
    } else if (error instanceof ReferenceError) {
        //处理引用错误
    } else {
        //处理其他类型的错误
    }
}
```

在跨浏览器编程中，检查错误类型是确定处理方式的最简便途径；包含在 message 属性中的错误消息会因浏览器而异。

3. 合理使用 try-catch

当 try-catch 语句中发生错误时，浏览器会认为错误已经被处理了，因而不会通过本章前面讨论的机制记录或报告错误。对于那些不要求用户懂技术，也不需要用户理解错误的 Web 应用程序，这应该说是个理想的结果。不过，try-catch 能够让我们实现自己的错误处理机制。

使用 try-catch 最适合处理那些我们无法控制的错误。假设你在使用一个大型 JavaScript 库中的函数，该函数可能会有意无意地抛出一些错误。由于我们不能修改这个库的源代码，所以大可将该函数的调用放在 try-catch 语句当中，万一有什么错误发生，也好恰当地处理它们。

在明明白白地知道自己的代码会发生错误时，再使用 try-catch 语句就不太合适了。例如，如果传递给函数的参数是字符串而非数值，就会造成函数出错，那么就应该先检查参数的类型，然后再决定如何去做。在这种情况下，不应用使用 try-catch 语句。

17.2.2 抛出错误

与 try-catch 语句相配的还有一个 throw 操作符，用于随时抛出自定义错误。抛出错误时，必须要给 throw 操作符指定一个值，这个值是什么类型，没有要求。下列代码都是有效的。

```
throw 12345;
throw "Hello world!";
throw true;
throw { name: "JavaScript"};
```

在遇到 throw 操作符时，代码会立即停止执行。仅当有 try-catch 语句捕获到被抛出的值时，代码才会继续执行。

通过使用某种内置错误类型，可以更真实地模拟浏览器错误。每种错误类型的构造函数接收一个参数，即实际的错误消息。下面是一个例子。

```
throw new Error("Something bad happened.");
```

这行代码抛出了一个通用错误，带有一条自定义错误消息。浏览器会像处理自己生成的错误一样，来处理这行代码抛出的错误。换句话说，浏览器会以常规方式报告这一错误，并且会显示这里的自定义错误消息。像下面使用其他错误类型，也可以模拟出类似的浏览器错误。

```
throw new SyntaxError("I don't like your syntax.");
throw new TypeError("What type of variable do you take me for?");
throw new RangeError("Sorry, you just don't have the range.");
throw new EvalError("That doesn't evaluate.");
throw new URIError("Uri, is that you?");
throw new ReferenceError("You didn't cite your references properly.");
```

在创建自定义错误消息时最常用的错误类型是 Error、RangeError、ReferenceError 和 TypeError。

另外，利用原型链还可以通过继承 Error 来创建自定义错误类型（原型链在第 6 章中介绍）。此时，需要为新创建的错误类型指定 name 和 message 属性。来看一个例子。

```
function CustomError(message){
    this.name = "CustomError";
    this.message = message;
}
```



```
CustomError.prototype = new Error();  
throw new CustomError("My message");
```

ThrowingErrorsExample01.htm

浏览器对待继承自 `Error` 的自定义错误类型，就像对待其他错误类型一样。如果要捕获自己抛出的错误并且把它与浏览器错误区别对待的话，创建自定义错误是很有用的。



IE 只有在抛出 `Error` 对象的时候才会显示自定义错误消息。对于其他类型，它都无一例外地显示“**exception thrown and not caught**”（抛出了异常，且未被捕获）。

1. 抛出错误的时机

要针对函数为什么会执行失败给出更多信息，抛出自定义错误是一种很方便的方式。应该在出现某种特定的已知错误条件，导致函数无法正常执行时抛出错误。换句话说，浏览器会在某种特定的条件下执行函数时抛出错误。例如，下面的函数会在参数不是数组的情况下失败。



```
function process(values){  
    values.sort();  
  
    for (var i=0, len=values.length; i < len; i++){  
        if (values[i] > 100){  
            return values[i];  
        }  
    }  
  
    return -1;  
}
```

ThrowingErrorsExample02.htm

如果执行这个函数时传给它一个字符串参数，那么对 `sort()` 的调用就会失败。对此，不同浏览器会给出不同的错误消息，但都不是特别明确，如下所示。

- IE: 属性或方法不存在。
- Firefox: `values.sort()` 不是函数。
- Safari: 值 `undefined` (表达式 `values.sort` 的结果) 不是对象。
- Chrome: 对象名没有方法 `'sort'`。
- Opera: 类型不匹配 (通常是在需要对象的地方使用了非对象值)。

尽管 Firefox、Chrome 和 Safari 都明确指出了代码中导致错误的部分，但错误消息并没有清楚地告诉我们到底出了什么问题，该怎么修复问题。在处理类似前面例子中的那个函数时，通过调试处理这些错误消息没有什么困难。但是，在面对包含数千行 JavaScript 代码的复杂的 Web 应用程序时，要想查找错误来源就没有那么容易了。这种情况下，带有适当信息的自定义错误能够显著提升代码的可维护性。来看下面的例子。



```
function process(values){
```

```
    if (!(values instanceof Array)){
        throw new Error("process(): Argument must be an array.");
    }

    values.sort();

    for (var i=0, len=values.length; i < len; i++){
        if (values[i] > 100){
            return values[i];
        }
    }

    return -1;
}
```

ThrowingErrorsExample02.htm

在重写后的这个函数中，如果 `values` 参数不是数组，就会抛出一个错误。错误消息中包含了函数的名称，以及为什么会发生错误的明确描述。如果一个复杂的 Web 应用程序发生了这个错误，那么查找问题的根源也就容易多了。

建议读者在开发 JavaScript 代码的过程中，重点关注函数和可能导致函数执行失败的因素。良好的错误处理机制应该可以确保代码中只发生你自己抛出的错误。



在多框架环境下使用 `instanceof` 来检测数组有一些问题。详细内容请参考 22.1.1 节。

2. 抛出错误与使用 `try-catch`

关于何时该抛出错误，而何时该使用 `try-catch` 来捕获它们，是一个老生常谈的问题。一般来说，应用程序架构的较低层次中经常会抛出错误，但这个层次并不会影响当前执行的代码，因而错误通常得不到真正的处理。如果你打算编写一个要在很多应用程序中使用的 JavaScript 库，甚至只编写一个可能会在应用程序内部多个地方使用的辅助函数，我都强烈建议你在抛出错误时提供详尽的信息。然后，即可在应用程序中捕获并适当地处理这些错误。

说到抛出错误与捕获错误，我们认为只应该捕获那些你确切地知道该如何处理的错误。捕获错误的目的在于避免浏览器以默认方式处理它们；而抛出错误的目的在于提供错误发生具体原因的消息。

17.2.3 错误 (error) 事件

任何没有通过 `try-catch` 处理的错误都会触发 `window` 对象的 `error` 事件。这个事件是 Web 浏览器最早支持的事件之一，IE、Firefox 和 Chrome 为保持向后兼容，并没有对这个事件作任何修改（Opera 和 Safari 不支持 `error` 事件）。在任何 Web 浏览器中，`onerror` 事件处理程序都不会创建 `event` 对象，但它可以接收三个参数：错误消息、错误所在的 URL 和行号。多数情况下，只有错误消息有用，因为 URL 只是给出了文档的位置，而行号所指的代码行既可能出自嵌入的 JavaScript 代码，也可能出自外部的文件。要指定 `onerror` 事件处理程序，必须使用如下所示的 DOM0 级技术，它没有遵循“DOM2 级事件”的标准格式。

```
window.onerror = function(message, url, line){
    alert(message);
};
```

只要发生错误，无论是不是浏览器生成的，都会触发 `error` 事件，并执行这个事件处理程序。然后，浏览器默认的机制发挥作用，像往常一样显示出错误消息。像下面这样在事件处理程序中返回 `false`，可以阻止浏览器报告错误的默认行为。

```
window.onerror = function(message, url, line){
    alert(message);
    return false;
};
```

OnErrorExample01.htm

通过返回 `false`，这个函数实际上就充当了整个文档中的 `try-catch` 语句，可以捕获所有无代码处理的运行时错误。这个事件处理程序是避免浏览器报告错误的最后一道防线，理想情况下，只要可能就不应该使用它。只要能够适当地使用 `try-catch` 语句，就不会有错误交给浏览器，也就不会触发 `error` 事件。



浏览器在使用这个事件处理错误时的方式有明显不同。在 IE 中，即使发生 `error` 事件，代码仍然会正常执行；所有变量和数据都将得到保留，因此能在 `onerror` 事件处理程序中访问它们。但在 Firefox 中，常规代码会停止执行，事件发生之前的所有变量和数据都将被销毁，因此几乎就无法判断错误了。

图像也支持 `error` 事件。只要图像的 `src` 特性中的 URL 不能返回可以被识别的图像格式，就会触发 `error` 事件。此时的 `error` 事件遵循 DOM 格式，会返回一个以图像为目标的 `event` 对象。下面是一个例子。

```
var image = new Image();
EventUtil.addHandler(image, "load", function(event){
    alert("Image loaded!");
});
EventUtil.addHandler(image, "error", function(event){
    alert("Image not loaded!");
});
image.src = "smilex.gif"; //指定不存在的文件
```

OnErrorExample02.htm

在这个例子中，当加载图像失败时就会显示一个警告框。需要注意的是，发生 `error` 事件时，图像下载过程已经结束，也就是说不能再重新下载了。

17.2.4 处理错误的策略

过去，所谓 Web 应用程序的错误处理策略仅限于服务器端。在谈到错误与错误处理时，通常要考虑很多方面，涉及一些工具，例如记录和监控系统。这些工具的用途在于分析错误模式，追查错误原因，同时帮助确定错误会影响到多少用户。

在 Web 应用程序的 JavaScript 这一端，错误处理策略也同样重要。由于任何 JavaScript 错误都可能导致网页无法使用，因此搞清楚何时以及为什么发生错误至关重要。绝大多数 Web 应用程序的用户都不懂技术，遇到错误时很容易心烦意乱。有时候，他们可能会刷新页面以期解决问题，而有时候则会放弃努力。作为开发人员，必须要知道代码何时可能出错，会出什么错，同时还要有一个跟踪此类问题的系统。

17.2.5 常见的错误类型

错误处理的核心，是首先要知道代码里会发生什么错误。由于 JavaScript 是松散类型的，而且也不会验证函数的参数，因此错误只会在代码运行期间出现。一般来说，需要关注三种错误：

- 类型转换错误
- 数据类型错误
- 通信错误

以上错误分别会在特定的模式下或者没有对值进行足够的检查的情况下发生。

1. 类型转换错误

类型转换错误发生在使用某个操作符，或者使用其他可能会自动转换值的数据类型的语言结构时。在使用相等（==）和不相等（!=）操作符，或者在 if、for 及 while 等流控制语句中使用非布尔值时，最常发生类型转换错误。

第 3 章讨论的相等和不相等操作符在执行比较之前会先转换不同类型的值。由于在非动态语言中，开发人员都使用相同的符号执行直观的比较，因此在 JavaScript 中往往也会以相同方式错误地使用它们。多数情况下，我们建议使用全等（===）和不全等（!==）操作符，以避免类型转换。来看一个例子。

```
alert(5 == "5");      //true
alert(5 === "5");     //false
alert(1 == true);     //true
alert(1 === true);    //false
```

这里使用了相等和全等操作符比较了数值 5 和字符串"5"。相等操作符首先会将数值 5 转换成字符串"5"，然后再将其与另一个字符串"5"进行比较，结果是 true。全等操作符知道要比较的是两种不同的数据类型，因而直接返回 false。对于 1 和 true 也是如此：相等操作符认为它们相等，而全等操作符认为它们不相等。使用全等和非全等操作符，可以避免发生因为使用相等和不相等操作符引发的类型转换错误，因此我们强烈推荐使用。

容易发生类型转换错误的另一个地方，就是流控制语句。像 if 之类的语句在确定下一步操作之前，会自动把任何值转换成布尔值。尤其是 if 语句，如果使用不当，最容易出错。来看下面的例子。

```
function concat(str1, str2, str3){
    var result = str1 + str2;
    if (str3){      //绝对不要这样!!!
        result += str3;
    }
    return result;
}
```

这个函数的用意是拼接两或三个字符串，然后返回结果。其中，第三个字符串是可选的，因此必须要检查。第 3 章曾经介绍过，未使用过的命名变量会自动被赋予 undefined 值。而 undefined 值可以被转换成布尔值 false，因此这个函数中的 if 语句实际上只适用于提供了第三个参数的情况。问题在

于，并不是只有 `undefined` 才会被转换成 `false`，也不是只有字符串值才可以转换为 `true`。例如，假设第三个参数是数值 0，那么 `if` 语句的测试就会失败，而对数值 1 的测试则会通过。

在流控制语句中使用非布尔值，是极为常见的一个错误来源。为避免此类错误，就要做到在条件比较时切实传入布尔值。实际上，执行某种形式的比较就可以达到这个目的。例如，我们可以将前面的函数重写如下。

```
function concat(str1, str2, str3){
    var result = str1 + str2;
    if (typeof str3 == "string"){ //恰当的比较
        result += str3;
    }
    return result;
}
```

在这个重写后的函数中，`if` 语句的条件会基于比较返回一个布尔值。这个函数相对可靠得多，不容易受非正常值的影响。

2. 数据类型错误

JavaScript 是松散类型的，也就是说，在使用变量和函数参数之前，不会对它们进行比较以确保它们的数据类型正确。为了保证不会发生数据类型错误，只能依靠开发人员编写适当的数据类型检测代码。在将预料之外的值传递给函数的情况下，最容易发生数据类型错误。

在前面的例子中，通过检测第三个参数可以确保它是一个字符串，但是并没有检测另外两个参数。如果该函数必须要返回一个字符串，那么只要给它传入两个数值，忽略第三个参数，就可以轻易地导致它的执行结果错误。类似的情况也存在于下面这个函数中。

```
//不安全的函数，任何非字符串值都会导致错误
function getQueryString(url){
    var pos = url.indexOf("?");
    if (pos > -1){
        return url.substring(pos + 1);
    }
    return "";
}
```

这个函数的用意是返回给定 URL 中的查询字符串。为此，它首先使用 `indexOf()` 寻找字符串中的问号。如果找到了，利用 `substring()` 方法返回问号后面的所有字符串。这个例子中的两个函数只能操作字符串，因此只要传入其他数据类型的值就会导致错误。而添加一条简单的类型检测语句，就可以确保函数不那么容易出错。

```
function getQueryString(url){
    if (typeof url == "string"){ //通过检查类型确保安全
        var pos = url.indexOf("?");
        if (pos > -1){
            return url.substring(pos + 1);
        }
    }
    return "";
}
```

重写后的这个函数首先检查了传入的值是不是字符串。这样，就确保了函数不会因为接收到非字符串值而导致错误。

前一节提到过，在流控制语句中使用非布尔值作为条件很容易导致类型转换错误。同样，这样做也经常会导致数据类型错误。来看下面的例子。

```
//不安全的函数，任何非数组值都会导致错误
function reverseSort(values){
    if (values){           //绝对不要这样!!!
        values.sort();
        values.reverse();
    }
}
```

这个 reverseSort() 函数可以将数组反向排序，其中用到了 sort() 和 reverse() 方法。对于 if 语句中的控制条件而言，任何会转换为 true 的非数组值都会导致错误。另一个常见的错误就是将参数与 null 值进行比较，如下所示。

```
//不安全的函数，任何非数组值都会导致错误
function reverseSort(values){
    if (values != null){    //绝对不要这样!!!
        values.sort();
        values.reverse();
    }
}
```

与 null 进行比较只能确保相应的值不是 null 和 undefined（这就相当于使用相等和不相等操作）。要确保传入的值有效，仅检测 null 值是不够的；因此，不应该使用这种技术。同样，我们也不推荐将某个值与 undefined 作比较。

另一种错误的做法，就是只针对要使用的某一个特性执行特性检测。来看下面的例子。

```
//还是不安全，任何非数组值都会导致错误
function reverseSort(values){
    if (typeof values.sort == "function"){           //绝对不要这样!!!
        values.sort();
        values.reverse();
    }
}
```

在这个例子中，代码首先检测了参数中是否存在 sort() 方法。这样，如果传入一个包含 sort() 方法的对象（而不是数组）当然也会通过检测，但在调用 reverse() 函数时可能就会出错了。在确切知道应该传入什么类型的情况下，最好是使用 instanceof 来检测其数据类型，如下所示。

```
//安全，非数组值将被忽略
function reverseSort(values){
    if (values instanceof Array){           //问题解决了
        values.sort();
        values.reverse();
    }
}
```

最后一个 reverseSort() 函数是安全的：它检测了 values，以确保这个参数是 Array 类型的实例。这样一来，就可以保证函数忽略任何非数组值。

大体上来说，基本类型的值应该使用 typeof 来检测，而对象的值则应该使用 instanceof 来检测。根据使用函数的方式，有时候并不需要逐个检测所有参数的数据类型。但是，面向公众的 API 则必须无条件地执行类型检查，以确保函数始终能够正常地执行。

3. 通信错误

随着 Ajax 编程的兴起（第 21 章讨论 Ajax），Web 应用程序在其生命周期内动态加载信息或功能，已经成为一件司空见惯的事。不过，JavaScript 与服务器之间的任何一次通信，都有可能产生错误。

第一种通信错误与格式不正确的 URL 或发送的数据有关。最常见的问题是在将数据发送给服务器之前，没有使用 `encodeURIComponent()` 对数据进行编码。例如，下面这个 URL 的格式就是不正确的：

```
http://www.yourdomain.com/?redir=http://www.someotherdomain.com?a=b&c=d
```

针对“redir=”后面的所有字符串调用 `encodeURIComponent()` 就可以解决这个问题，结果将产生如下字符串：

```
http://www.yourdomain.com/?redir=http%3A%2F%2Fwww.someotherdomain.com%3Fa%3Db%26c%3Dd
```

对于查询字符串，应该记住必须要使用 `encodeURIComponent()` 方法。为了确保这一点，有时候可以定义一个处理查询字符串的函数，例如：

```
function addQueryStringArg(url, name, value){
    if (url.indexOf("?") == -1){
        url += "?";
    } else {
        url += "&";
    }

    url += encodeURIComponent(name) + "=" + encodeURIComponent(value);
    return url;
}
```

这个函数接收三个参数：要追加查询字符串的 URL、参数名和参数值。如果传入的 URL 不包含问号，还要给它添加问号；否则，就要添加一个和号，因为有问号就意味着有其他查询字符串。然后，再将经过编码的查询字符串的名和值添加到 URL 后面。可以像下面这样使用这个函数：

```
var url = "http://www.somedomain.com";
var newUrl = addQueryStringArg(url, "redir",
                                "http://www.someotherdomain.com?a=b&c=d");
alert(newUrl);
```

使用这个函数而不是手工构建 URL，可以确保编码正确并避免相关错误。

另外，在服务器响应的数据不正确时，也会发生通信错误。第 10 章曾经讨论过动态加载脚本和动态加载样式，运用这两种技术都有可能遇到资源不可用的情况。在没有返回相应资源的情况下，Firefox、Chrome 和 Safari 会默默地失败，IE 和 Opera 则都会报错。然而，对于使用这两种技术产生的错误，很难判断和处理。在某些情况下，使用 Ajax 通信可以提供有关错误状态的更多信息。



在使用 Ajax 通信的情况下，也可能发生通信错误。相关的问题和错误将在第 21 章讨论。

17.2.6 区分致命错误和非致命错误

任何错误处理策略中最重要的一部分，就是确定错误是否致命。对于非致命错误，可以根据下列一或多个条件来确定：

- ❑ 不影响用户的主要任务；
- ❑ 只影响页面的一部分；
- ❑ 可以恢复；
- ❑ 重复相同操作可以消除错误。

本质上，非致命错误并不是需要关注的问题。例如，Yahoo! Mail (<http://mail.yahoo.com>) 有一项功能，允许用户在其界面上发送手机短信。如果由于某种原因，发不了手机短信了，那也不算是致命错误，因为并不是应用程序的主要功能有问题。用户使用 Yahoo! Mail 主要是为了查收和撰写电子邮件。只在这个主要功能正常，就没有理由打断用户。没有必要因为发生了非致命错误而对用户给出提示——可以把页面中受到影响的区域替换掉，比如替换成说明相应功能无法使用的消息。但是，如果因此打断用户，那确实没有必要。

致命错误，可以通过以下一或多个条件来确定：

- ❑ 应用程序根本无法继续运行；
- ❑ 错误明显影响到了用户的主要操作；
- ❑ 会导致其他连带错误。

要想采取适当的措施，必须要知道 JavaScript 在什么情况下会发生致命错误。在发生致命错误时，应该立即给用户发送一条消息，告诉他们无法再继续手头的事情了。假如必须刷新页面才能让应用程序正常运行，就必须通知用户，同时给用户提供一个点击即可刷新页面的按钮。

区分非致命错误和致命错误的主要依据，就是看它们对用户的影响。设计良好的代码，可以做到应用程序某一部分发生错误不会不必要地影响另一个实际上毫不相干的部分。例如，My Yahoo! (<http://my.yahoo.com>) 的个性化主页上包含了很多互不依赖的模块。如果每个模块都需要通过 JavaScript 调用来初始化，那么你可能会看到类似下面这样的代码：

```
for (var i=0, len=mods.length; i < len; i++){
    mods[i].init(); //可能会导致致命错误
}
```

表面上看，这些代码没什么问题：依次对每个模块调用 `init()` 方法。问题在于，任何模块的 `init()` 方法如果出错，都会导致数组中后续的所有模块无法再进行初始化。从逻辑上说，这样编写代码没有什么意义。毕竟，每个模块相互之间没有依赖关系，各自实现不同功能。可能会导致致命错误的原因是代码的结构。不过，经过下面这样修改，就可以把所有模块的错误变成非致命的：

```
for (var i=0, len=mods.length; i < len; i++){
    try {
        mods[i].init();
    } catch (ex) {
        //在这里处理错误
    }
}
```

通过在 `for` 循环中添加 `try-catch` 语句，任何模块初始化时出错，都不会影响其他模块的初始化。在以上重写的代码中，如果有错误发生，相应的错误将会得到独立的处理，并不会影响到用户的体验。

17.2.7 把错误记录到服务器

开发 Web 应用程序过程中的一种常见的做法，就是集中保存错误日志，以便查找重要错误的原因。

例如数据库和服务器错误都会定期写入日志，而且会按照常用 API 进行分类。在复杂的 Web 应用程序中，我们同样推荐你把 JavaScript 错误也回写到服务器。换句话说，也要将这些错误写入到保存服务器端错误的地方，只不过要标明它们来自前端。把前后端的错误集中起来，能够极大地方便对数据的分析。

要建立这样一种 JavaScript 错误记录系统，首先需要在服务器上创建一个页面（或者一个服务器入口点），用于处理错误数据。这个页面的作用无非就是从查询字符串中取得数据，然后再将数据写入错误日志中。这个页面可能会使用如下所示的函数：

```
function logError(sev, msg){
    var img = new Image();
    img.src = "log.php?sev=" + encodeURIComponent(sev) + "&msg=" +
        encodeURIComponent(msg);
}
```

这个 `logError()` 函数接收两个参数：表示严重程度的数值或字符串（视所用系统而异）及错误消息。其中，使用了 `Image` 对象来发送请求，这样做非常灵活，主要表现如下几方面。

- 所有浏览器都支持 `Image` 对象，包括那些不支持 `XMLHttpRequest` 对象的浏览器。
- 可以避免跨域限制。通常都是一台服务器要负责处理多台服务器的错误，而这种情况下使用 `XMLHttpRequest` 是不行的。
- 在记录错误的过程中出问题的概率比较低。大多数 Ajax 通信都是由 JavaScript 库提供的包装函数来处理的，如果库代码本身有问题，而你还在依赖该库记录错误，可想而知，错误消息是不可能得到记录的。

只要是使用 `try-catch` 语句，就应该把相应错误记录到日志中。来看下面的例子。

```
for (var i=0, len=mods.length; i < len; i++){
    try {
        mods[i].init();
    } catch (ex){
        logError("nonfatal", "Module init failed: " + ex.message);
    }
}
```

在这里，一旦模块初始化失败，就会调用 `logError()`。第一个参数是“nonfatal”（非致命），表示错误的严重程度。第二个参数是上下文信息加上真正的 JavaScript 错误消息。记录到服务器中的错误消息应该尽可能多地带有上下文信息，以便鉴别导致错误的真正原因。

17.3 调试技术

在不那么容易找到 JavaScript 调试程序的年代，开发人员不得不发挥自己的创造力，通过各种方法来调试自己的代码。结果，就出现了以这样或那样的方式置入代码，从而输出调试信息的做法。其中，最常见的做法就是在要调试的代码中随处插入 `alert()` 函数。但这种做法一方面比较麻烦（调试之后还需要清理），另一方面还可能引入新问题（想象一下把某个 `alert()` 函数遗留在产品代码中的结果）。如今，已经有了很多更好的调试工具，因此我们也不再建议在调试中使用 `alert()` 了。

17.3.1 将消息记录到控制台

IE8、Firefox、Opera、Chrome 和 Safari 都有 JavaScript 控制台，可以用来查看 JavaScript 错误。而

且，在这些浏览器中，都可以通过代码向控制台输出消息。对 Firefox 而言，需要安装 Firebug (www.getfirebug.com)，因为 Firefox 要使用 Firebug 的控制台。对 IE8、Firefox、Chrome 和 Safari 来说，则可以通过 console 对象向 JavaScript 控制台中写入消息，这个对象具有下列方法。

- ❑ `error(message)`：将错误消息记录到控制台
- ❑ `info(message)`：将信息性消息记录到控制台
- ❑ `log(message)`：将一般消息记录到控制台
- ❑ `warn(message)`：将警告消息记录到控制台

在 IE8、Firebug、Chrome 和 Safari 中，用来记录消息的方法不同，控制台中显示的错误消息也不一样。错误消息带有红色图标，而警告消息带有黄色图标。以下函数展示了使用控制台输出消息的一个示例。

```
function sum(num1, num2){
    console.log("Entering sum(), arguments are " + num1 + "," + num2);

    console.log("Before calculation");
    var result = num1 + num2;
    console.log("After calculation");

    console.log("Exiting sum()");
    return result;
}
```

在调用这个 `sum()` 函数时，控制台中会出现一些消息，可以用来辅助调试。在 Safari 中，通过“Develop”（开发）菜单可以打开其 JavaScript 控制台（前面讨论过）；在 Chrome 中，单击“Control this page”（控制当前页）按钮并选择“Developer”（开发人员）和“JavaScript console”（JavaScript 控制台）即可；而在 Firefox 中，要打开控制台需要单击 Firefox 状态栏右下角的图标。IE8 的控制台是其 Developer Tools（开发人员工具）扩展的一部分，通过“Tools”（工具）菜单可以找到，其控制台在“Script”（脚本）选项卡中。

Opera 10.5 之前的版本中，JavaScript 控制台可以通过 `opera.postError()` 方法来访问。这个方法接受一个参数，即要写入到控制台中的参数，其用法如下。

```
function sum(num1, num2){
    opera.postError("Entering sum(), arguments are " + num1 + "," + num2);

    opera.postError("Before calculation");
    var result = num1 + num2;
    opera.postError("After calculation");

    opera.postError("Exiting sum()");
    return result;
}
```

别看 `opera.postError()` 方法的名字好像是只能输出错误，但实际上能通过它向 JavaScript 控制台中写入任何信息。

还有一种方案是使用 LiveConnect，也就是在 JavaScript 中运行 Java 代码。Firefox、Safari 和 Opera 都支持 LiveConnect，因此可以操作 Java 控制台。例如，通过下列代码就可以在 JavaScript 中把消息写入到 Java 控制台。

```
java.lang.System.out.println("Your message");
```

可以用这行代码替代 `console.log()` 或 `opera.postError()`，如下所示。


```
function sum(num1, num2){
    java.lang.System.out.println("Entering sum(), arguments are " + num1 + "," + num2);

    java.lang.System.out.println("Before calculation");
    var result = num1 + num2;
    java.lang.System.out.println("After calculation");

    java.lang.System.out.println("Exiting sum()");
    return result;
}
```

如果系统设置恰当，可以在调用 LiveConnect 时就立即显示 Java 控制台。在 Firefox 中，通过“Tools”（工具）菜单可以打开 Java 控制台；在 Opera 中，要打开 Java 控制台，可以选择菜单“Tools”（工具）及“Advanced”（高级）。Safari 没有内置对 Java 控制台的支持，必须单独运行。

不存在一种跨浏览器向 JavaScript 控制台写入消息的机制，但下面的函数倒可以作为统一的接口。



```
function log(message){
    if (typeof console == "object"){
        console.log(message);
    } else if (typeof opera == "object"){
        opera.postError(message);
    } else if (typeof java == "object" && typeof java.lang == "object"){
        java.lang.System.out.println(message);
    }
}
```

[ConsoleLoggingExample01.htm](#)

这个 `log()` 函数检测了哪个 JavaScript 控制台接口可用，然后使用相应的接口。可以在任何浏览器中安全地使用这个函数，不会导致任何错误，例如：

```
function sum(num1, num2){
    log("Entering sum(), arguments are " + num1 + "," + num2);

    log("Before calculation");
    var result = num1 + num2;
    log("After calculation");

    log("Exiting sum()");
    return result;
}
```

[ConsoleLoggingExample01.htm](#)


向 JavaScript 控制台中写入消息可以辅助调试代码，但在发布应用程序时，还必须要移除所有消息。在部署应用程序时，可以通过手工或通过特定的代码处理步骤来自动完成清理工作。



记录消息要比使用 `alert()` 函数更可取，因为警告框会阻断程序的执行，而在测定异步处理对时间的影响时，使用警告框会影响结果。

17.3.2 将消息记录到当前页面

另一种输出调试消息的方式，就是在页面中开辟一小块区域，用以显示消息。这个区域通常是一个元素，而该元素可以总是出现在页面中，但仅用于调试目的；也可以是一个根据需要动态创建的元素。例如，可以将 `log()` 函数修改为如下所示：



```
function log(message){
    var console = document.getElementById("debuginfo");
    if (console === null){
        console = document.createElement("div");
        console.id = "debuginfo";
        console.style.background = "#dedede";
        console.style.border = "1px solid silver";
        console.style.padding = "5px";
        console.style.width = "400px";
        console.style.position = "absolute";
        console.style.right = "0px";
        console.style.top = "0px";
        document.body.appendChild(console);
    }
    console.innerHTML += "<p>" + message + "</p>";
}
```

PageLoggingExample01.htm

这个修改后的 `log()` 函数首先检测是否已经存在调试元素，如果没有则会新创建一个 `<div>` 元素，并为该元素应用一些样式，以便与页面中的其他元素区别开。然后，又使用 `innerHTML` 将消息写入到这个 `<div>` 元素中。结果就是页面中会有一小块区域显示错误消息。这种技术在不支持 JavaScript 控制台的 IE7 及更早版本或其他浏览器中十分有用。



与把错误消息记录到控制台相似，把错误消息输出到页面的代码也要在发布前删除。

17.3.3 抛出错误

如前所述，抛出错误也是一种调试代码的好办法。如果错误消息很具体，基本上就可以把它当作确定错误来源的依据。但这种错误消息必须能够明确给出导致错误的原因，才能省去其他调试操作。来看下面的函数：


```
function divide(num1, num2){
    return num1 / num2;
}
```

这个简单的函数计算两个数的除法，但如果有一个参数不是数值，它会返回 `NaN`。类似这样简单的计算如果返回 `NaN`，就会在 Web 应用程序中导致问题。对此，可以在计算之前，先检测每个参数是否都是数值。例如：

```
function divide(num1, num2){
    if (typeof num1 != "number" || typeof num2 != "number"){
        throw new Error("divide(): Both arguments must be numbers.");
    }
    return num1 / num2;
}
```

在此，如果有一个参数不是数值，就会抛出错误。错误消息中包含了函数的名字，以及导致错误的真正原因。浏览器只要报告了这个错误消息，我们就可以立即知道错误来源及问题的性质。相对来说，这种具体的错误消息要比那些泛泛的浏览器错误消息更有用。

对于大型应用程序来说，自定义的错误通常都使用 `assert()` 函数抛出。这个函数接受两个参数，一个是求值结果应该为 `true` 的条件，另一个是条件为 `false` 时要抛出的错误。以下就是一个非常基本的 `assert()` 函数。



```
function assert(condition, message){
    if (!condition){
        throw new Error(message);
    }
}
```

AssertExample01.htm

可以用这个 `assert()` 函数代替某些函数中需要调试的 `if` 语句，以便输出错误消息。下面是使用这个函数的例子。

```
function divide(num1, num2){
    assert(typeof num1 == "number" && typeof num2 == "number",
        "divide(): Both arguments must be numbers.");
    return num1 / num2;
}
```

AssertExample01.htm

可见，使用 `assert()` 函数可以减少抛出错误所需的代码量，而且也比前面的代码更容易看懂。

17.4 常见的 IE 错误

多年以来，IE 一直都是最难于调试 JavaScript 错误的浏览器。IE 给出的错误消息一般很短又语焉不详，而且上下文信息也很少，有时甚至一点都没有。但作为用户最多的浏览器，如何看懂 IE 给出的错误也是最受关注的。下面几小节将分别探讨一些在 IE 中难于调试的 JavaScript 错误。

17.4.1 操作终止

在 IE8 之前的版本中，存在一个相对于其他浏览器而言，最令人迷惑、讨厌，也最难于调试的错误：操作终止（operation aborted）。在修改尚未加载完成的页面时，就会发生操作终止错误。发生错误时，会出现一个模态对话框，告诉你“操作终止。”单击确定（OK）按钮，则卸载整个页面，继而显示一张空白屏幕；此时要进行调试非常困难。下面的示例将会导致操作终止错误。

```
<!DOCTYPE html>
<html>
<head>
    <title>Operation Aborted Example</title>
</head>
<body>
    <p>The following code should cause an Operation Aborted error in IE versions
    prior to 8.</p>
```



```


<div>
  <script type="text/javascript">
    document.body.appendChild(document.createElement("div"));
  </script>
</div>
</body>
</html>

```

OperationAbortedExample01.htm

这个例子中存在的问题是：JavaScript 代码在页面尚未加载完毕时就要修改 document.body，而且 <script> 元素还不是 <body> 元素的直接子元素。准确一点说，当 <script> 节点被包含在某个元素中，而且 JavaScript 代码又要使用 appendChild()、innerHTML 或其他 DOM 方法修改该元素的父元素或祖先元素时，将会发生操作终止错误（因为只能修改已经加载完毕的元素）。

要避免这个问题，可以等到目标元素加载完毕后再对它进行操作，或者使用其他操作方法。例如，为 document.body 添加一个绝对定位在页面上的覆盖层，就是一种非常常见的操作。通常，开发人员都是使用 appendChild() 方法来添加这个元素的，但换成使用 insertBefore() 方法也很容易。因此，只要修改前面例子中的一行代码，就可以避免操作终止错误。



```


<!DOCTYPE html>
<html>
<head>
  <title>Operation Aborted Example</title>
</head>
<body>
  <p>The following code should not cause an Operation Aborted error in IE
  versions prior to 8.</p>
  <div>
    <script type="text/javascript">
      document.body.insertBefore(document.createElement("div"),
      document.body.firstChild);
    </script>
  </div>
</body>
</html>

```

OperationAbortedExample02.htm

在这个例子中，新的 <div> 元素被添加到 document.body 的开头部分而不是末尾。因为完成这一操作所需的所有信息在脚本运行时都是已知的，所以这不会引发错误。

除了改变方法之外，还可以把 <script> 元素从包含元素中移出来，直接作为 <body> 的子元素。例如：



```

<!DOCTYPE html>
<html>
<head>
  <title>Operation Aborted Example</title>
</head>
<body>
  <p>The following code should not cause an Operation Aborted error in IE
  versions prior to 8.</p>
  <div>
  </div>
  <script type="text/javascript">
    document.body.appendChild(document.createElement("div"));
  </script>

```

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

这一次也不会发生错误，因为脚本修改的是它的直接父元素，而不再是间接的祖先元素。在同样的情况下，IE8不再抛出操作终止错误，而是抛出常规的 JavaScript 错误，带有如下错误消息：

```
HTML Parsing Error: Unable to modify the parent container element before the child
element is closed (KB927917).
```

不过，虽然浏览器抛出的错误不同，但解决方案仍然是一样的。

17.4.2 无效字符

根据语法，JavaScript 文件必须只包含特定的字符。在 JavaScript 文件中存在无效字符时，IE 会抛出无效字符（invalid character）错误。所谓无效字符，就是 JavaScript 语法中未定义的字符。例如，有一个很像减号但却由 Unicode 值 8211 表示的字符（\u2013），就不能用作常规的减号（ASCII 编码为 45），因为 JavaScript 语法中没有定义该字符。这个字符通常是在 Word 文档中自动插入的。如果你的代码是从 Word 文档中复制到文本编辑器中，然后又在 IE 中运行的，那么就可能会遇到无效字符错误。其他浏览器对无效字符做出的反应与 IE 类似，Firefox 会抛出非法字符（illegal character）错误，Safari 会报告发生了语法错误，而 Opera 则会报告发生了 ReferenceError（引用错误），因为它会将无效字符解释为未定义的标识符。

17.4.3 未找到成员

如前所述，IE 中的所有 DOM 对象都是以 COM 对象，而非原生 JavaScript 对象的形式实现的。这会导致一些与垃圾收集相关的非常奇怪的行为。IE 中的未找到成员（Member not found）错误，就是由于垃圾收集例程配合错误所直接导致的。

具体来说，如果在对象被销毁之后，又给该对象赋值，就会导致未找到成员错误。而导致这个错误的，一定是 COM 对象。发生这个错误的最常见情形是使用 event 对象的时候。IE 中的 event 对象是 window 的属性，该对象在事件发生时创建，在最后一个事件处理程序执行完毕后销毁。假设你在一个闭包中使用了 event 对象，而该闭包不会立即执行，那么在将来调用它并给 event 的属性赋值时，就会导致未找到成员错误，如下面的例子所示。

```
document.onclick = function(){
    var event = window.event;
    setTimeout(function(){
        event.returnValue = false;           //未找到成员错误
    }, 1000);
};
```

在这段代码中，我们将一个单击事件处理程序指定给了文档。在事件处理程序中，window.event 被保存在 event 变量中。然后，传入 setTimeout() 中的闭包里又包含了 event 变量。当单击事件处理程序执行完毕后，event 对象就会被销毁，因而闭包中引用对象的成员就成了不存在的了。换句话说，由于不能在 COM 对象被销毁之后再给其成员赋值，在闭包中给 returnValue 赋值就会导致未找到成员错误。

17.4.4 未知运行时错误

当使用 `innerHTML` 或 `outerHTML` 以下列方式指定 HTML 时,就会发生未知运行时错误(`Unknown runtime error`):一是把块元素插入到行内元素时,二是访问表格任意部分(`<table>`、`<tbody>` 等)的任意属性时。例如,从技术角度说, `` 标签不能包含 `<div>` 之类的块级元素,因此下面的代码就会导致未知运行时错误:

```
span.innerHTML = "<div>Hi</div>";           //这里,span 包含了<div>元素
```

在遇到把块级元素插入到不恰当位置的情况时,其他浏览器会尝试纠正并隐藏错误,而 IE 在这一点上反倒很较真儿。

17.4.5 语法错误


通常,只要 IE 一报告发生了语法错误(`syntax error`),都可以很快找到错误的原因。这时候,原因可能是代码中少了一个分号,或者花括号前后不对应。然而,还有一种原因不十分明显的情况需要格外注意。

如果你引用了外部的 JavaScript 文件,而该文件最终并没有返回 JavaScript 代码,IE 也会抛出语法错误。例如, `<script>` 元素的 `src` 特性指向了一个 HTML 文件,就会导致语法错误。报告语法错误的位置时,通常都会说该错误位于脚本第一行的第一个字符处。Opera 和 Safari 也会报告语法错误,但它们会给出导致问题的外部文件的信息;IE 就不会给出这个信息,因此就需要我们自己重复检查一遍引用的外部 JavaScript 文件。但 Firefox 会忽略那些被当作 JavaScript 内容嵌入到文档中的非 JavaScript 文件中的解析错误。

在服务器端组件动态生成 JavaScript 的情况下,比较容易出现这种错误。很多服务器端语言都会发生在运行时错误时,向输出中插入 HTML 代码,而这种包含 HTML 的输出很容易就会违反 JavaScript 语法。如果在追查语法错误时遇到了麻烦,我们建议你再仔细检查一遍引用的外部文件,确保这些文件中没有包含服务器因错误而插入到其中的 HTML。

17.4.6 系统无法找到指定资源

系统无法找到指定资源(`The system cannot locate the resource specified`)这种说法,恐怕要算是 IE 给出的最有价值的错误消息了。在使用 JavaScript 请求某个资源 URL,而该 URL 的长度超过了 IE 对 URL 最长不能超过 2083 个字符的限制时,就会发生这个错误。IE 不仅限制 JavaScript 中使用的 URL 的长度,而且也限制用户在浏览器自身中使用的 URL 长度(其他浏览器对 URL 的限制没有这么严格)。IE 对 URL 路径还有一个不能超过 2048 个字符的限制。下面的代码将会导致错误。



```
function createLongUrl(url){
    var s = "?";
    for (var i=0, len=2500; i < len; i++){
        s += "a";
    }

    return url + s;
}

var x = new XMLHttpRequest();
```

```
x.open("get", createLongUrl("http://www.somedomain.com/"), true);  
x.send(null);
```

在这个例子中, XMLHttpRequest 对象试图向一个超出最大长度限制的 URL 发送请求。在调用 open() 方法时, 就会发生错误。避免这个问题的办法, 无非就是通过给查询字符串参数起更短的名字, 或者减少不必要的参数, 来缩短查询字符串的长度。另外, 还可以把请求方法改为 POST, 通过请求体而不是查询字符串来发送数据。有关 Ajax (或者说 XMLHttpRequest 对象) 的详细内容, 将在第 21 章全面讨论。

17.5 小结

错误处理对于今天复杂的 Web 应用程序开发而言至关重要。不能提前预测到可能发生的错误, 不能提前采取恢复策略, 可能导致较差的用户体验, 最终引发用户不满。多数浏览器在默认情况下都不会向用户报告错误, 因此在开发和调试期间需要启用浏览器的错误报告功能。然而, 在投入运行的产品代码中, 则不应该再有诸如此类的错误报告出现。

下面是几种避免浏览器响应 JavaScript 错误的方法。

- 在可能发生错误的地方使用 try-catch 语句, 这样你还有机会以适当的方式对错误给出响应, 而不必沿用浏览器处理错误的机制。
- 使用 window.onerror 事件处理程序, 这种方式可以接受 try-catch 不能处理的所有错误(仅限于 IE、Firefox 和 Chrome)。

另外, 对任何 Web 应用程序都应该分析可能的错误来源, 并制定处理错误的方案。

- 首先, 必须要明确什么是致命错误, 什么是非致命错误。
- 其次, 再分析代码, 以判断最可能发生的错误。JavaScript 中发生错误的主要原因如下。
 - 类型转换
 - 未充分检测数据类型
 - 发送给服务器或从服务器接收到的数据有错误

IE、Firefox、Chrome、Opera 和 Safari 都有 JavaScript 调试器, 有的是内置的, 有的是以需要下载的扩展形式存在的。这些调试器都支持设置断点、控制代码执行及在运行时检测变量的值。