js异常处理机制

# 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。

## 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;  }  } |

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。

## 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encodeURI()或decodeURI()，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。

## 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，所以大可将对该函数的调用放在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。

# 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）。

## 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。

## 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。

## 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;

};

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。

## 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。

## 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。

### 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。

### 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。

### 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，都有可能会产生错误。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。

## 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个部分，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。

## 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，我们推荐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。