encodeURIComponent()编码，会导致这种错误。例如，下面的 URL 格式就不正确：

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

这个 URL 可以通过用 encodeURIComponent()编码"redir="后面的内容来修复，得到的结果如下所示：

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

对于查询字符串，应该都要通过 encodeURIComponent()编码。为此，可以专门定义一个处理查询字符串的函数，比如：

```
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：要添加查询字符串的 URL、参数名和参数值。如果 URL 不包含问号，则要给它加上一个；否则就要使用和号（&），以便拼接更多参数和值，因为这意味着前面已有其他查询参数了。查询字符串的名和值在被编码之后会被添加到 URL 中。可以像下面这样使用这个函数：

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

使用这个函数而不是手动构建 URL 可以保证编码合适，以避免相关错误发生。

在服务器响应非预期值时也会发生通信错误。在动态加载脚本或样式时，请求的资源有可能不可用。有些浏览器在没有返回预期资源时会静默失败，而其他浏览器则会报告错误。不过，在动态加载资源的情况下出错，是不太好做错误处理的。有时候，使用 Ajax 通信可能会提供关于错误条件的更多信息。

21.2.6　区分重大与非重大错误

任何错误处理策略中一个非常重要的方面就是确定某个错误是否为重大错误。具有以下一个或多个特性的错误属于非重大错误：

- ❑ 不会影响用户的主要任务；
- ❑ 只会影响页面中某个部分；
- ❑ 可以恢复；
- ❑ 重复操作可能成功。

本质上，不需要担心非重大错误。例如，Gmail 有一个功能，可以让用户在其界面上发送环聊（Hangouts）消息。如果在某个条件下，环聊功能不工作了，就不能算重大错误，因为这不是应用程序的主要功能。Gmail 主要用于阅读和撰写电子邮件，只要用户可以做到这一点，就没有理由中断用户体验。对于非重大错误，无须明确给用户发送消息。可以将受影响的页面区域替换成一条消息，表示该功能暂时不能使用，但不需要中断用户体验。

另一方面，重大错误具备如下特性：

- 应用程序绝对无法继续运行；
- 错误严重影响了用户的主要目标；
- 会导致其他错误发生。

理解 JavaScript 中何时会发生重大错误极其重要，因为这样才能采取应对措施。当重大错误发生时，应该立即发送消息让用户知晓自己不能再继续使用应用程序了。如果必须刷新页面才能恢复应用程序，那就应该明确告知用户，并提供一个自动刷新页面的按钮。

代码中则不要区分什么是或什么不是重大错误。非重大错误和重大错误的区别主要体现在对用户的影响上。好的代码设计意味着应用程序某个部分的错误不会影响其他部分，实际上根本不应该相关。例如，在个性化的主页上，比如 Gmail，可能包含多个相互独立的功能模块。如果每个模块都通过 JavaScript 调用来初始化，那就可能会在代码中看到以下逻辑：

```
for (let mod of mods){
    mod.init(); // 可能的重大错误
}
```

表面上看，这段代码没什么问题，就是依次调用每个模块的 `init()`方法。问题在于，这里只要有一个模块的 `init()`方法出错，数组中其后的所有模块都不会被初始化。如果错误发生在第一个模块上，页面上就没有模块会被初始化了。逻辑上，这样写代码是不合适的，因为每个模块相互独立，各自功能没有相关性。由此可能导致重大错误的原因是代码的结构。好在可以简单地重写以上代码，让每个模块的错误变成非重大错误：

```
for (let mod of mods){
  try {
    mod.init();
  } catch (ex){
    // 在这里处理错误
  }
}
```

通过在 `for` 循环中加入 `try/catch` 语句，模块初始化过程中的任何错误都不会影响其他模块初始化。如果代码中有错误发生，则可以单独处理，并不会影响用户体验。

21.2.7 把错误记录到服务器中

Web 应用程序开发中的一个常见做法是建立中心化的错误日志存储和跟踪系统。数据库和服务器错误正常写到日志中并按照常用 API 加以分类。对复杂的 Web 应用程序而言，最好也把 JavaScript 错误发送回服务器记录下来。这样做可以把错误记录到与服务器相同的系统，只要把它们归类到前端错误即可。使用相同的系统可以进行相同的分析，而不用考虑错误来源。

要建立 JavaScript 错误日志系统，首先需要在服务器上有页面或入口可以处理错误数据。该页面只要从查询字符串中取得错误数据，然后把它们保存到错误日志中即可。比如，该页面可以使用如下代码：

```
function logError(sev, msg) {
  let img = new Image(),
      encodedSev = encodeURIComponent(sev),
      encodedMsg = encodeURIComponent(msg);
  img.src = 'log.php?sev=${encodedSev}&msg=${encodedMsg}';
}
```

logError()函数接收两个参数：严重程度和错误消息。严重程度可以是数值或字符串，具体取决于使用的日志系统。这里使用 Image 对象发送请求主要是从灵活性方面考虑的。

- 所有浏览器都支持 Image 对象，即使不支持 XMLHttpRequest 对象也一样。
- 不受跨域规则限制。通常，接收错误消息的应该是多个服务器中的一个，而 XMLHttpRequest 此时就比较麻烦。
- 记录错误的过程很少出错。大多数 Ajax 通信借助 JavaScript 库的包装来处理。如果这个库本身出错，而你又要利用它记录错误，那么显然错误消息永远不会发给服务器。

只要是使用 try/catch 语句的地方，都可以把相关错误记录下来。下面是一个例子：

```
for (let mod of mods){
  try {
    mod.init();
  } catch (ex){
    logError("nonfatal", 'Module init failed: ${ex.message}');
  }
}
```

在这个例子中，模块初始化失败就会调用 logError()函数。第一个参数是表示错误严重程度的"nonfatal"，第二个参数在上下文信息后面追加了 JavaScript 错误消息。记录到服务器的错误消息应该包含尽量多的上下文信息，以便找出错误的确切原因。

21.3　调试技术

在 JavaScript 调试器出现以前，开发者必须使用创造性的方法调试代码。结果就出现了各种各样专门为输出调试信息而设计的代码。其中最为常用的调试技术是在相关代码中插入 alert()，这种方式既费事（调试完之后还得清理）又麻烦（如果有漏洞的警告框出现在产品环境中，会给用户造成不便）。已不再推荐将警告框用于调试，因为有其他更好的解决方案。

21.3.1　把消息记录到控制台

所有主流浏览器都有 JavaScript 控制台，该控制台可用于查询 JavaScript 错误。另外，这些浏览器都支持通过 console 对象直接把 JavaScript 消息写入控制台，这个对象包含如下方法。

- error(*message*)：在控制台中记录错误消息。
- info(*message*)：在控制台中记录信息性内容。
- log(*message*)：在控制台记录常规消息。
- warn(*message*)：在控制台中记录警告消息。

记录消息时使用的方法不同，消息显示的样式也不同。错误消息包含一个红叉图标，而警告消息包含一个黄色叹号图标。可以像下面这样使用控制台消息：

```
function sum(num1, num2){
  console.log('Entering sum(), arguments are ${num1},${num2}');
  console.log("Before calculation");
  const result = num1 + num2;
  console.log("After calculation");

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

在调用 sum()函数时，会有一系列消息输出到控制台以辅助调试。

把消息输出到 JavaScript 控制台可以辅助调试代码，但在产品环境下应该删除所有相关代码。这可以在部署时使用代码自动完成清理，也可以手动删除。

> **注意** 相比于使用警告框，打印日志消息是更好的调试方法。这是因为警告框会阻塞代码执行，从而影响对异步操作的计时，进而影响代码的结果。打印日志也可以随意输出任意多个参数并检查对象实例（警告框只能将对象序列化为一个字符串再展示出来，因此经常会看到 Object[Object]。

21.3.2 理解控制台运行时

浏览器控制台是个读取–求值–打印–循环（REPL，read-eval-print-loop），与页面的 JavaScript 运行时并发。这个运行时就像浏览器对新出现在 DOM 中的<script>标签求值一样。在控制台中执行的命令可以像页面级 JavaScript 一样访问全局和各种 API。控制台中可以执行任意数量的代码，与它可能会阻塞的任何页面级代码一样。修改、对象和回调都会保留在 DOM 和运行时中。

JavaScript 运行时会限制不同窗口可以访问哪些内容，因而在所有主流浏览器中都可以选择在哪个窗口中执行 JavaScript 控制台输入。你所执行的代码不会有特权提升，仍会受跨源限制和其他浏览器施加的控制规则约束。

控制台运行时也会集成开发者工具，提供常规 JavaScript 开发中所没有的上下文调试工具。其中一个非常有用的工具是最后点击选择器，所有主流浏览器都会提供。在开发者工具的 Element（元素）标签页内，单击 DOM 树中一个节点，就可以在 Console（控制台）标签页中使用$0 引用该节点的 JavaScript 实例。它就跟普通的 JavaScript 实例一样，因此可以读取属性（如$0.scrollWidth），或者调用成员方法（如$0.remove()）。

21.3.3 使用 JavaScript 调试器

在所有主流浏览器中都可以使用的还有 JavaScript 调试器。ECMAScript 5.1 规范定义了 debugger 关键字，用于调用可能存在的调试功能。如果没有相关的功能，这条语句会被简单地跳过。可以像下面这样使用 debugger 关键字：

```
function pauseExecution(){
  console.log("Will print before breakpoint");
  debugger;
  console.log("Will not print until breakpoint continues");
}
```

在运行时碰到这个关键字时，所有主流浏览器都会打开开发者工具面板，并在指定位置显示断点。然后，可以通过单独的浏览器控制台在断点所在的特定词法作用域中执行代码。此外，还可以执行标准的代码调试器操作（单步进入、单步跳过、继续，等等）。

浏览器也支持在开发者工具的源代码标签页中选择希望设置断点的代码行来手动设置断点（不使用 debugger 关键字）。这样设置的断点与使用 debugger 关键字设置的一样，只是不会在不同浏览器会话之间保持。

21.3.4　在页面中打印消息

另一种常见的打印调试消息的方式是把消息写到页面中指定的区域。这个区域可以是所有页面中都包含的元素，但仅用于调试目的；也可以是在需要时临时创建的元素。例如，可以定义这样 log() 函数：

```
function log(message) {
  // 这个函数的词法作用域会使用这个实例
  // 而不是 window.console
  const 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>';
}
```

在这个 log() 函数中，代码先检测是否已创建了调试用的元素。如果没有，就创建一个新 元素并给它添加一些样式，以便与页面其他部分区分出来。此后，再使用 innerHTML 属性把消息写到这个 中。结果就是在页面的一个小区域内显示日志信息。

> **注意**　与在控制台输出消息一样，在页面中输入消息的代码也需要从生产环境中删除。

21.3.5　补充控制台方法

记住使用哪个日志方法（原生的 console.log() 和自定义的 log() 方法），对开发者来说是一种负担。因为 console 是一个全局对象，所以可以为这个对象添加方法，也可以用自定义的函数重写已有的方法，这样无论在哪里用到的日志打印方法，都会按照自定义的方式行事。

比如，可以这样重新定义 console.log 函数：

```
// 把所有参数拼接为一个字符串，然后打印出结果
console.log = function() {
  // 'arguments'并没有 join 方法，这里先把它转换为数组
  const args = Array.prototype.slice.call(arguments);
  console.log(args.join(', '));
}
```

这样，其他代码调用的将是这个函数，而不是通用的日志方法。这样的修改在页面刷新后会失效，因此只是调试或拦截日志的一个有用而轻量的策略。

21.3.6　抛出错误

如前所述，抛出错误是调试代码的很好方式。如果错误消息足够具体，只要看一眼错误就可以确定原因。好的错误消息包含关于错误原因的确切信息，因此可以减少额外调试的工作量。比如下面的函数：

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

这个简单的函数执行两个数的除法，但如果任何一个参数不是数值，则返回 NaN。当 Web 应用程序意外返回 NaN 时，简单的计算可能就会出问题。此时，可以检查每个参数的类型是不是数值，然后再进行计算。来看下面的例子：

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

这里，任何一个参数不是数值都会抛出错误。错误消息中包含函数名和错误的具体原因。当浏览器报告这个错误消息时，你立即就能根据它包含的信息定位到问题，包括问题的解决方案。相对于没那么具体的浏览器错误消息，这个错误消息显示更有价值。

在大型应用程序中，自定义错误通常使用 assert() 函数抛出错误。这个函数接收一个应该为 true 的条件，并在条件为 false 时抛出错误。下面是一个基本的 assert() 函数：

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

这个 assert() 函数可用于代替多个 if 语句，同时也是记录错误的好地方。下面的代码演示了如何使用它：

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

相比于之前的例子，使用 assert() 函数可以减少抛出自定义错误所需的代码量，并且让代码更好理解。

21.4 旧版 IE 的常见错误

IE 曾是最难调试 JavaScript 错误的浏览器之一。该浏览器的旧版本抛出的错误通常比较短，比较含糊，缺少上下文。接下来几节分别讨论旧版 IE 中可能会出现的常见且难于调试的 JavaScript 错误。因为这些浏览器不支持 ES6，所以代码会考虑向后兼容。

21.4.1 无效字符

JavaScript 文件中的代码必须由特定字符构成。在检测到 JavaScript 文件中存在无效字符时，IE 会抛出"invalid character"错误。所谓无效字符，指的是 JavaScript 语法中没有定义过的字符。例如，一个看起来像减号而实际上并不是减号的字符（Unicode 值为\u2013）。这个字符不能用于代替减号（ASCII 码为 45），因为它不是 JavaScript 语法定义的减号。这个特殊字符经常会被自动插入 Word 文档，因此如果把它从 Word 文档复制到文本编辑器然后在 IE 中运行，IE 就会报告文件中包含非法字符。其他

浏览器也类似，Firefox 抛出"illegal character"错误，Safari 报告语法错误，而 Opera 则报告 ReferenceError（因为把这个字符当成了未定义标识符来解释）。

21.4.2　未找到成员

如前所述，旧版 IE 中所有 DOM 对象都是用 COM 对象实现的，并非原生 JavaScript 对象。在涉及垃圾回收时，这可能会导致很多奇怪的行为。其中，"member not found"错误是 IE 中垃圾回收程序常报告的错误。

这个错误通常会在给一个已被销毁的对象赋值时发生。这个对象必须是 COM 对象才会出现这个消息。最好的一个例子就是 event 对象。IE 的 event 对象是作为 window 的一个属性存在的，会在事件发生时创建，在事件处理程序执行完毕后销毁。因此，如果你想在稍后会执行的闭包中使用 event 对象，尝试给 event 对象赋值就会导致这个错误，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
  }, 1000);
};
```

在这个例子中，文档被添加了单击事件处理程序。事件处理程序把 window.event 对象保存在一个名为 event 的本地变量中。然后，在传递给 setTimeout()的闭包中引用这个事件变量。当 onclick 事件处理程序退出后，event 对象会被销毁，因此闭包中对它的引用也就不存在了，于是就会报告未找到成员错误。之所以给 event.returnValue 赋值会导致"member not found"错误，是因为不能给已将其成员销毁的 COM 对象赋值。

21.4.3　未知运行时错误

使用 innerHTML 或 outerHTML 属性以下面一种方式添加 HTML 时会发生未知运行时错误：比如将块级元素插入行内元素，或者在表格的任何部分（<table>、<tbody>等）访问了其中一个属性。例如，从技术角度来说，<p>标签不能包含另一个块级元素，如 ，因此以下代码会导致未知运行时错误：

```
p.innerHTML = "<div>Hi</div>"; // where p contains a <p> element
```

在将块级元素插入不恰当的位置时，其他浏览器会尝试纠正，这样就不会发生错误，但 IE 在这种情况下要严格得多。

21.4.4　语法错误

通常，当 IE 报告语法错误时，原因是很清楚的。一般来说，可以通过错误消息追踪到少了一个分号或括号错配。不过，有一种情况下报告的语法错误并不清楚。

如果网页中引用的一个外部 JavaScript 文件由于某种原因返回了非 JavaScript 代码，则 IE 会抛出语法错误。例如，错误地把<script>标签的 src 属性设置为指向一个 HTML 文件，就会导致语法错误。通常会报告该语法错误发生在脚本第一行的第一个字符。Opera 和 Safari 此时也会报告语法错误，但它们也会报告是引用文件不当导致的问题。IE 没有这些信息，因此需要仔细检查引用的每个外部 JavaScript 文件。Firefox 会忽略作为 JavaScript 引用的非 JavaScript 文件导致的解析错误。

这种错误通常发生在服务器端动态生成 JavaScript 的情况下。很多服务器端语言会在发生运行时错误时，自动向输出中插入 HTML。这种输出显然会导致 JavaScript 语法错误。如果你碰到了难以排除的语法错误，可以仔细检查所有外部文件，确保没有文件包含服务器由于错误而插入的 HTML。

21.4.5 系统找不到指定资源

还有一个可能最没用的消息："The system cannot locate the resource specified"（系统找不到指定资源）。这个错误会在 JavaScript 向某个 URL 发送请求，而该 URL 长度超过了 IE 允许的最大 URL 长度（2083 个字符）时发生。这个长度限制不仅针对 JavaScript，而且针对 IE 本身。（其他浏览器没有这么严格地限制 URL 长度。）另外，IE 对 URL 路径还有 2048 个字符的限制。下面的代码会导致这个错误：

```
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` 对象尝试向超过 URL 长度限制的地址发送请求。在调用 `open()` 方法时，错误会发生。为避免这种错误，一个办法是缩短请求成功所需的查询字符串，比如缩短参数名或去掉不必要的数据。另一个办法是改为使用 POST 请求，不用查询字符串而通过请求体发送数据。

21.5 小结

对于今天复杂的 Web 应用程序而言，JavaScript 中的错误处理十分重要。未能预测什么时候会发生错误以及如何从错误中恢复，会导致糟糕的用户体验，甚至造成用户流失。大多数浏览器默认不向用户报告 JavaScript 错误，因此在开发和调试时需要自己实现错误报告。不过在生产环境中，不应该以这种方式报告错误。

下列方法可用于阻止浏览器对 JavaScript 错误作出反应。

- 使用 `try/catch` 语句，可以通过更合适的方式对错误做出处理，避免浏览器处理。
- 定义 `window.onerror` 事件处理程序，所有没有通过 `try/catch` 处理的错误都会被该事件处理程序接收到（仅限 IE、Firefox 和 Chrome）。

开发 Web 应用程序时，应该认真考虑可能发生的错误，以及如何处理这些错误。

- 首先，应该分清哪些算重大错误，哪些不算重大错误。
- 然后，要通过分析代码预测很可能发生哪些错误。由于以下因素，JavaScript 中经常出现错误：
  - 类型转换；
  - 数据类型检测不足；
  - 向服务器发送错误数据或从服务器接收到错误数据。

IE、Firefox、Chrome、Opera 和 Safari 都有 JavaScript 调试器，有的内置在浏览器中，有的是作为扩展，需另行下载。所有调试器都能够设置断点、控制代码执行和在运行时检查变量值。

第22章

处理 XML

本章内容

- ❑ 浏览器对 XML DOM 的支持
- ❑ 在 JavaScript 中使用 XPath
- ❑ 使用 XSLT 处理器

XML 曾一度是在互联网上存储和传输结构化数据的标准。XML 的发展反映了 Web 的发展，因为 DOM 标准不仅是为了在浏览器中使用，而且还为了在桌面和服务器应用程序中处理 XML 数据结构。在没有 DOM 标准的时候，很多开发者使用 JavaScript 编写自己的 XML 解析器。自从有了 DOM 标准，所有浏览器都开始原生支持 XML、XML DOM 及很多其他相关技术。

22.1 浏览器对 XML DOM 的支持

因为很多浏览器在正式标准问世之前就开始实现 XML 解析方案，所以不同浏览器对标准的支持不仅有级别上的差异，也有实现上的差异。DOM Level 3 增加了解析和序列化能力。不过，在 DOM Level 3 制定完成时，大多数浏览器也已实现了自己的解析方案。

22.1.1 DOM Level 2 Core

正如第 12 章所述，DOM Level 2 增加了 `document.implementation` 的 `createDocument()` 方法。有读者可能还记得，可以像下面这样创建空 XML 文档：

```
let xmldom = document.implementation.createDocument(namespaceUri, root, doctype);
```

在 JavaScript 中处理 XML 时，`root` 参数通常只会使用一次，因为这个参数定义的是 XML DOM 中 `document` 元素的标签名。`namespaceUri` 参数用得很少，因为在 JavaScript 中很难管理命名空间。`doctype` 参数则更是少用。

要创建一个 `document` 对象标签名为`<root>`的新 XML 文档，可以使用以下代码：

```
let xmldom = document.implementation.createDocument("", "root", null);

console.log(xmldom.documentElement.tagName); // "root"

let child = xmldom.createElement("child");
xmldom.documentElement.appendChild(child);
```

这个例子创建了一个 XML DOM 文档，该文档没有默认的命名空间和文档类型。注意，即使不指定命名空间和文档类型，参数还是要传的。命名空间传入空字符串表示不应用命名空间，文档类型传入 `null` 表示没有文档类型。`xmldom` 变量包含 DOM Level 2 `Document` 类型的实例，包括第 12 章介绍的

所有 DOM 方法和属性。在这个例子中，我们打印了 document 元素的标签名，然后又为它创建并添加了一个新的子元素。

要检查浏览器是否支持 DOM Level 2 XML，可以使用如下代码：

```
let hasXmlDom = document.implementation.hasFeature("XML", "2.0");
```

实践中，很少需要凭空创建 XML 文档，然后使用 DOM 方法来系统创建 XML 数据结构。更多是把 XML 文档解析为 DOM 结构，或者相反。因为 DOM Level 2 并未提供这种功能，所以出现了一些事实标准。

22.1.2 DOMParser 类型

Firefox 专门为把 XML 解析为 DOM 文档新增了 DOMParser 类型，后来所有其他浏览器也实现了该类型。要使用 DOMParser，需要先创建它的一个实例，然后再调用 parseFromString()方法。这个方法接收两个参数：要解析的 XML 字符串和内容类型（始终应该是"text/html"）。返回值是 Document 的实例。来看下面的例子：

```
let parser = new DOMParser();
let xmldom = parser.parseFromString("<root><child/></root>", "text/xml");

console.log(xmldom.documentElement.tagName); // "root"
console.log(xmldom.documentElement.firstChild.tagName); // "child"

let anotherChild = xmldom.createElement("child");
xmldom.documentElement.appendChild(anotherChild);

let children = xmldom.getElementsByTagName("child");
console.log(children.length); // 2
```

这个例子把简单的 XML 字符串解析为 DOM 文档。得到的 DOM 结构中<root>是 document 元素，它有个子元素<child>。然后就可以使用 DOM 方法与返回的文档进行交互。

DOMParser 只能解析格式良好的 XML，因此不能把 HTML 解析为 HTML 文档。在发生解析错误时，不同浏览器的行为也不一样。Firefox、Opera、Safari 和 Chrome 在发生解析错误时，parseFromString()方法仍会返回一个 Document 对象，只不过其 document 元素是<parsererror>，该元素的内容为解析错误的描述。下面是一个解析错误的示例：

```
<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">XML
Parsing Error: no element found Location: file:// /I:/My%20Writing/My%20Books/
Professional%20JavaScript/Second%20Edition/Examples/Ch15/DOMParserExample2.js Line
Number 1, Column 7:<sourcetext>&lt;root&gt; ------^</sourcetext></parsererror>
```

Firefox 和 Opera 都会返回这种格式的文档。Safari 和 Chrome 返回的文档会把<parsererror>元素嵌入在发生解析错误的位置。早期 IE 版本会在调用 parseFromString()的地方抛出解析错误。由于这些差异，最好使用 try/catch 来判断是否发生了解析错误，如果没有错误，则通过 getElements-ByTagName()方法查找文档中是否包含<parsererror>元素，如下所示：

```
let parser = new DOMParser(),
  xmldom,
  errors;
try {
  xmldom = parser.parseFromString("<root>", "text/xml");
  errors = xmldom.getElementsByTagName("parsererror");
  if (errors.length > 0) {
```