### 21.1.2　移动控制台

移动浏览器不会直接在设备上提供控制台界面。不过，还是有一些途径可以在移动设备中检查错误。

Chrome 移动版和 Safari 的 iOS 版内置了实用工具，支持将设备连接到宿主操作系统中相同的浏览器。然后，就可以在对应的桌面浏览器中查看错误了。这涉及设备之间的硬件连接，且要遵循不同的操作步骤，比如 Chrome 的操作步骤参见 Google Developers 网站的文章《Android 设备的远程调试入门》，Safari 的操作步骤参见 Apple Developer 网站的文章“Safari Web Inspector Guide”。

此外也可以使用第三方工具直接在移动设备上调试。Firefox 常用的调试工具是 Firebug Lite，这需要通过 JavaScript 的书签小工具向当前页面中加入 Firebug 脚本才可以。脚本运行后，就可以直接在移动浏览器上打开调试界面。Firebug Lite 也有面向其他浏览器（如 Chrome）的版本。

## 21.2　错误处理

错误处理在编程中的重要性毋庸置疑。所有主流 Web 应用程序都需要定义完善的错误处理协议，大多数优秀的应用程序有自己的错误处理策略，尽管主要逻辑是放在服务器端的。事实上，服务器端团队通常会花很多精力根据错误类型、频率和其他重要指标来定义规范的错误日志机制。最终实现通过简单的数据库查询或报告生成脚本就可以了解应用程序的运行状态。

错误处理在应用程序的浏览器端进展较慢，尽管其重要性一点也不低。这里有一个重要的事实：大多数上网的人没有技术背景，甚至连什么是浏览器都不十分清楚，而且有的人不知道自己使用的是什么浏览器。如前所述，当网页中的 JavaScript 脚本发生错误时，不同浏览器的处理方式不同。不过浏览器处理 JavaScript 报告错误的默认方式对用户并不友好。最好的情况是用户自己不知道发生了什么，然后再重试；最坏的情况是用户感觉特别厌烦，于是永远不回来了。有一个良好的错误处理策略可以让用户知道到底发生了什么。为此，必须理解各种捕获和处理 JavaScript 错误的方式。

### 21.2.1　try/catch 语句

ECMA-262 第 3 版新增了 try/catch 语句，作为在 JavaScript 中处理异常的一种方式。基本的语法如下所示，跟 Java 中的 try/catch 语句一样：

```
try {
  // 可能出错的代码
} catch (error) {
  // 出错时要做什么
}
```

任何可能出错的代码都应该放到 try 块中，而处理错误的代码则放在 catch 块中，如下所示：

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

如果 try 块中有代码发生错误，代码会立即退出执行，并跳到 catch 块中。catch 块此时接收到一个对象，该对象包含发生错误的相关信息。与其他语言不同，即使在 catch 块中不使用错误对象，也必须为它定义名称。错误对象中暴露的实际信息因浏览器而异，但至少包含保存错误消息的 message 属性。ECMA-262 也指定了定义错误类型的 name 属性，目前所有浏览器中都有这个属性。因此，可以

像下面的代码这样在必要时显示错误消息：

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

这个例子使用 message 属性向用户显示错误消息。message 属性是唯一一个在 IE、Firefox、Safari、Chrome 和 Opera 中都有的属性，尽管每个浏览器添加了其他属性。IE 添加了 description 属性（其值始终等于 message）和 number 属性（它包含内部错误号）。Firefox 添加了 fileName、lineNumber 和 stack（包含栈跟踪信息）属性。Safari 添加了 line（行号）、sourceId（内部错误号）和 sourceURL 属性。同样，为保证跨浏览器兼容，最好只依赖 message 属性。

#### 1. finally 子句

try/catch 语句中可选的 finally 子句始终运行。如果 try 块中的代码运行完，则接着执行 finally 块中的代码。如果出错并执行 catch 块中的代码，则 finally 块中的代码仍执行。try 或 catch 块无法阻止 finally 块执行，包括 return 语句。比如：

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

这个函数在 try/catch 语句的各个部分都只放了一个 return 语句。看起来该函数应该返回 2，因为它在 try 块中，不会导致错误。但是，finally 块的存在导致 try 块中的 return 语句被忽略。因此，无论什么情况下调用该函数都会返回 0。如果去掉 finally 子句，该函数会返回 2。如果写出 finally 子句，catch 块就成了可选的（它们两者中只有一个是必需的）。

> **注意** 只要代码中包含了 finally 子句，try 块或 catch 块中的 return 语句就会被忽略，理解这一点很重要。在使用 finally 时一定要仔细确认代码的行为。

#### 2. 错误类型

代码执行过程中会发生各种类型的错误。每种类型都会对应一个错误发生时抛出的错误对象。ECMA-262 定义了以下 8 种错误类型：

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

Error 是基类型，其他错误类型继承该类型。因此，所有错误类型都共享相同的属性（所有错误对

象上的方法都是这个默认类型定义的方法）。浏览器很少会抛出 Error 类型的错误，该类型主要用于开发者抛出自定义错误。

InternalError 类型的错误会在底层 JavaScript 引擎抛出异常时由浏览器抛出。例如，递归过多导致了栈溢出。这个类型并不是代码中通常要处理的错误，如果真发生了这种错误，很可能代码哪里弄错了或者有危险了。

EvalError 类型的错误会在使用 eval()函数发生异常时抛出。ECMA-262 规定，“如果 eval 属性没有被直接调用（即没有将其名称作为一个 Identifier，也就是 CallExpression 中的 MemberExpression），或者如果 eval 属性被赋值”，就会抛出该错误。基本上，只要不把 eval()当成函数调用就会报告该错误：

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

实践中，浏览器不会总抛出 EvalError。Firefox 和 IE 在上面第一种情况下抛出 TypeError，在第二种情况下抛出 EvalError。为此，再加上代码中不大可能这样使用 eval()，因此几乎遇不到这种错误。

RangeError 错误会在数值越界时抛出。例如，定义数组时如果设置了并不支持的长度，如-20 或 Number.MAX_VALUE，就会报告该错误：

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

RangeError 在 JavaScript 中发生得不多。

ReferenceError 会在找不到对象时发生。（这就是著名的"object expected"浏览器错误的原因。）这种错误经常是由访问不存在的变量而导致的，比如：

```
let obj = x; // 在 x 没有声明时会抛出 ReferenceError
```

SyntaxError 经常在给 eval()传入的字符串包含 JavaScript 语法错误时发生，比如：

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

在 eval()外部，很少会用到 SyntaxError。这是因为 JavaScript 代码中的语法错误会导致代码无法执行。

TypeError 在 JavaScript 中很常见，主要发生在变量不是预期类型，或者访问不存在的方法时。很多原因可能导致这种错误，尤其是在使用类型特定的操作而变量类型不对时。下面是几个例子：

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

在给函数传参数之前没有验证其类型的情况下，类型错误频繁发生。

最后一种错误类型是 URIError，只会在使用 encodeURI()或 decodeURI()但传入了格式错误的 URI 时发生。这个错误恐怕是 JavaScript 中难得一见的错误了，因为上面这两个函数非常稳健。

不同的错误类型可用于为异常提供更多信息，以便实现适当的错误处理逻辑。在 try/catch 语句的 catch 块中，可以使用 instanceof 操作符确定错误的类型，比如：

```
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 {
    // 处理所有其他类型的错误
  }
}
```

检查错误类型是以跨浏览器方式确定适当操作过程的最简单方法，因为 message 属性中包含的错误消息因浏览器而异。

#### 3. try/catch 的用法

当 try/catch 中发生错误时，浏览器会认为错误被处理了，因此就不会再使用本章前面提到的机制报告错误。如果应用程序的用户不懂技术，那么他们即使看到错误也看不懂，这是一个理想的结果。使用 try/catch 可以针对特定错误类型实现自定义的错误处理。

try/catch 语句最好用在自己无法控制的错误上。例如，假设你的代码中使用了一个大型 JavaScript 库的某个函数，而该函数可能会有意或由于出错而抛出错误。因为不能修改这个库的代码，所以为防止这个函数报告错误，就有必要通过 try/catch 语句把该函数调用包装起来，对可能的错误进行处理。

如果你明确知道自己的代码会发生某种错误，那么就不适合使用 try/catch 语句。例如，如果给函数传入字符串而不是数值时就会失败，就应该检查该函数的参数类型并采取相应的操作。这种情况下，没有必要使用 try/catch 语句。



### 21.2.2 抛出错误

与 try/catch 语句对应的一个机制是 throw 操作符，用于在任何时候抛出自定义错误。throw 操作符必须有一个值，但值的类型不限。下面这些代码都是有效的：

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

使用 throw 操作符时，代码立即停止执行，除非 try/catch 语句捕获了抛出的值。

可以通过内置的错误类型来模拟浏览器错误。每种错误类型的构造函数都只接收一个参数，就是错误消息。下面看一个例子：

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

以上代码使用一个自定义的错误消息生成了一个通用错误。浏览器会像处理自己生成的错误一样来处理这个自定义错误。换句话说，浏览器会像通常一样报告这个错误，最终显示这个自定义错误。当然，使用特定的错误类型也是一样的，如以下代码所示：

```
throw new SyntaxError("I don't like your syntax.");
throw new InternalError("I can't do that, Dave.");
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.");
```

自定义错误常用的错误类型是 Error、RangeError、ReferenceError 和 TypeError。

此外，通过继承 Error（第 6 章介绍过继承）也可以创建自定义的错误类型。创建自定义错误类型时，需要提供 name 属性和 message 属性，比如：

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
    this.message = message;
  }
}

throw new CustomError("My message");
```

继承 Error 的自定义错误类型会被浏览器当成其他内置错误类型。自定义错误类型有助于在捕获错误时更准确地区分错误。

**1. 何时抛出错误**

抛出自定义错误是解释函数为什么失败的有效方式。在出现已知函数无法正确执行的情况时就应该抛出错误。换句话说，浏览器会在给定条件下执行该函数时抛出错误。例如，下面的函数会在参数不是数组时抛出错误：

```
function process(values){
  values.sort();

  for (let value of values){
    if (value > 100){
      return value;
    }
  }

  return -1;
}
```

如果给这个函数传入字符串，调用 sort()函数就会失败。每种浏览器对此都会给出一个模棱两可的错误消息，如下所示。

- IE：属性或方法不存在。
- Firefox：values.sort()不是函数。
- Safari：值 undefined（对表达式 values.sort 求值的结果）不是一个对象。
- Chrome：对象名没有方法'sort'。
- Opera：类型不匹配（通常是在需要对象时使用了非对象值）。

虽然 Firefox、Chrome 和 Safari 至少给出了导致错误的相关代码，但并没有哪个错误消息特别明确地指出发生了什么，或者怎么修复。对于上面的一个函数来说，通过这样的错误消息调试还是很容易的。但是，如果是一个复杂的 Web 应用程序，有几千行 JavaScript 代码，想要找到错误的原因就会很难。

这时候，使用适当的信息创建自定义错误可以有效提高代码的可维护性。比如下面的例子：

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

  values.sort();
```

```
    for (let value of values){
      if (value > 100){
        return value;
      }
    }

    return -1;
}
```

在这个重写后的函数中，如果 values 参数不是数组就会抛出错误。错误消息包含函数名以及对错误原因非常清晰的描述。即使在复杂的应用程序中出现这个错误，也可以很容易理解问题所在。

实际编写 JavaScript 代码时，应该仔细评估每个函数，以及可能导致它们失败的情形。良好的错误处理协议可以保证只会发生你自己抛出的错误。

#### 2. 抛出错误与 try/catch

一个常见的问题是何时抛出错误，何时使用 try/catch 捕获错误。一般来说，错误要在应用程序架构的底层抛出，在这个层面上，人们对正在进行的流程知之甚少，因此无法真正地处理错误。如果你在编写一个可能用于很多应用程序的 JavaScript 库，或者一个会在应用程序的很多地方用到的实用函数，那么应该认真考虑抛出带有详细信息的错误。然后捕获和处理错误交给应用程序就行了。

至于抛出错误与捕获错误的区别，可以这样想：应该只在确切知道接下来该做什么的时候捕获错误。捕获错误的目的是阻止浏览器以其默认方式响应；抛出错误的目的是为错误提供有关其发生原因的说明。



### 21.2.3 error 事件

任何没有被 try/catch 语句处理的错误都会在 window 对象上触发 error 事件。该事件是浏览器早期支持的事件，为保持向后兼容，很多浏览器保持了其格式不变。在 onerror 事件处理程序中，任何浏览器都不会传入 event 对象。相反，会传入 3 个参数：错误消息、发生错误的 URL 和行号。大多数情况下，只有错误消息有用，因为 URL 就是当前文档的地址，而行号可能指嵌入 JavaScript 或外部文件中的代码。另外，onerror 事件处理程序需要使用 DOM Level 0 技术来指定，因为它不遵循 DOM Level 2 Events 标准格式：

```
window.onerror = (message, url, line) => {
  console.log(message);
};
```

在任何错误发生时，无论是否是浏览器生成的，都会触发 error 事件并执行这个事件处理程序。然后，浏览器的默认行为就会生效，像往常一样显示这条错误消息。可以返回 false 来阻止浏览器默认报告错误的行为，如下所示：

```
window.onerror = (message, url, line) => {
  console.log(message);
  return false;
};
```

通过返回 false，这个函数实际上就变成了整个文档的 try/catch 语句，可以捕获所有未处理的运行时错误。这个事件处理程序应该是处理浏览器报告错误的最后一道防线。理想情况下，最好永远不要用到。适当使用 try/catch 语句意味着不会有错误到达浏览器这个层次，因此也就不会触发 error 事件。

> **注意**　浏览器在使用这个事件处理错误时存在明显差异。在 IE 中发生 error 事件时，正常代码会继续执行，所有变量和数据会保持，且可以在 onerror 事件处理程序中访问。然而在 Firefox 中，正常代码会执行会终止，错误发生之前的所有变量和数据会被销毁，导致很难真正分析处理错误。

图片也支持 error 事件。任何时候，如果图片 src 属性中的 URL 没有返回可识别的图片格式，就会触发 error 事件。这个事件遵循 DOM 格式，返回一个以图片为目标的 event 对象。下面是个例子：

```
const image = new Image();

image.addEventListener("load", (event) => {
  console.log("Image loaded!");
});
image.addEventListener("error", (event) => {
  console.log("Image not loaded!");
});

image.src = "doesnotexist.gif"; // 不存在，资源会加载失败
```

在这个例子中，图片加载失败后会显示一个 alert 警告框。这里的关键在于，当 error 事件发生时，图片下载过程已结束，不会再恢复。

### 21.2.4　错误处理策略

过去，Web 应用程序的错误处理策略基本上是在服务器上落地。错误处理策略涉及很多错误和错误处理考量，包括日志记录和监控系统。这些主要是为了分析模式，以期找到问题的根源并了解有多少用户会受错误影响。

在 Web 应用程序的 JavaScipt 层面落地错误处理策略同样重要。因为任何 JavaScript 错误都可能导致网页无法使用，所以理解这些错误会在什么情况下发生以及为什么会发生非常重要。绝大多数 Web 应用程序的用户不懂技术，在碰到页面出问题时通常会迷惑。为解决问题，他们可能会尝试刷新页面，也可能会直接放弃。作为开发者，应该非常清楚自己的代码在什么情况下会失败，以及失败会导致什么结果。另外，还要有一个系统跟踪这些问题。

### 21.2.5　识别错误

错误处理非常重要的部分是首先识别错误可能会在代码中的什么地方发生。因为 JavaScript 是松散类型的，不会验证函数参数，所以很多错误只有在代码真正运行起来时才会出现。通常，需要注意 3 类错误：

- 类型转换错误
- 数据类型错误
- 通信错误

上面这几种错误会在特定情况下，在没有对值进行充分检测时发生。

#### 1. 静态代码分析器

不得不说的是，通过在代码构建流程中添加静态代码分析或代码检查器（linter），可以预先发现非常多的错误。这样的代码分析工具有很多，详见 GitHub Gist 网站 All Gists 页面。常用的静态分析工具

是 JSHint、JSLint、Google Closure 和 TypeScript。

静态代码分析器要求使用类型、函数签名及其他指令来注解 JavaScript，以此描述程序如何在基本可执行代码之外运行。分析器会比较注解和 JavaScript 代码的各个部分，对在实际运行时可能出现的潜在不兼容问题给出提醒。

> **注意** 随着代码数量的增长，代码分析器会变得越来越重要，尤其是协作开发者也在增加的情况下。所有主流技术公司都有着庞大的 JavaScript 库，并会在构建流程中使用稳健的静态分析工具。

#### 2. 类型转换错误

**类型转换错误**的主要原因是使用了会自动改变某个值的数据类型的操作符或语言构造。使用等于（==）或不等于（!=）操作符，以及在 if、for 或 while 等流控制语句中使用非布尔值，经常会导致类型转换错误。

第 3 章曾讨论过，相等和不相等操作符会自动把执行比较的两个不同类型的值转换为相同类型。在非动态语言中，符号之间是直接比较的，因此很多开发者在 JavaScript 中也会以相同方式来错误地比较值。大多数情况下，最好使用严格相等（===）和严格不相等（!==）操作符来避免类型转换。来看下面的例子：

```
console.log(5 == "5");   // true
console.log(5 === "5");  // false
console.log(1 == true);  // true
console.log(1 === true); // false
```

这个例子分别使用了相等和严格相等操作符比较了数值 5 和字符串"5"。相等操作符会把字符串"5"转换为数值 5，然后再进行比较，结果是 true。严格相等操作符发现两个值的数据类型不同，因而直接返回 false。同样，对于 1 和 true 的比较也类似。相等操作符认为它们相等，但严格相等操作符认为它们不相等。使用严格相等和严格不相等操作符可以避免比较过程的类型转换错误，强烈推荐用它们代替相等和不相等操作符。

> **注意** 代码风格指南通常会指出什么时候应使用===，什么时候应使用==。有些风格指南认同只要始终使用===，类型转换就不再是个问题。另一些则认为除了可能发生字符串/布尔值转换的情形，在其他时候使用===均是用力过猛的表现。

类型转换错误也会发生在流控制语句中。比如，if 语句会自动把条件表达式转换为布尔值，然后再决定下一步的走向。在实践中，if 语句是问题比较多的。来看下面的例子：

```
function concat(str1, str2, str3) {
  let result = str1 + str2;
  if (str3) { // 不要!
    result += str3;
  }
  return result;
}
```

这个函数的用意是把两个或三个字符串拼接起来并返回结果。第三个字符串是可选的，因此必须检测它是否存在。如第 3 章所说，命名变量如果没有被赋值就会自动被赋予 undefined 值。而在默认转

换中，undefined 会被转换为布尔值 false。因此这个函数的用意是在提供了第三个参数的情况下，才会在拼接时带上它。问题在于并非只有undefined会转换为false,字符串也不是唯一可转换为true的值。假如第三个参数是数值 0，if 条件判断就会失败，而数值 1 则会导致满足条件。

在流控制语句中使用非布尔值作为条件是很常见的错误来源。为避免这类错误，需要始终坚持使用布尔值作为条件。这通常可以借助某种比较来实现。例如，可以把前面的函数改写为如下形式：

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

在这个重写的版本中，if 语句的条件会基于比较操作返回布尔值。这个函数相对更安全，受错误值影响的可能性也更小。

### 3. 数据类型错误

因为 JavaScript 是松散类型的，所以变量和函数参数都不能保证会使用正确的数据类型。开发者需要自己检查数据类型，确保不会发生错误。数据类型错误常发生在将意外值传给函数的时候。

在前面的例子中，代码检查了第三个参数的数据类型，以确保它是字符串，但根本没有检查另外两个参数。如果函数必须返回一个字符串，那么只传入两个数值，忽略第三个参数就会破坏约定。下面的函数也存在类似问题：

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

这个函数的用途是返回给定 URL 的查询字符串。为此，它先用 indexOf()在字符串中寻找问号，如果找到则使用 substring()方法返回问号后面的所有内容。这两个方法都是只有字符串才有的，因此传入其他类型的值就会导致错误。下面的简单类型检查可以保证函数少出错：

```
function getQueryString(url) {
  if (typeof url === "string") { // 通过类型检查保证安全
    let pos = url.indexOf("?");
    if (pos > -1) {
      return url.substring(pos +1);
    }
  }
  return "";
}
```

在这个重写的版本中，第一步检查了传入的值确实是字符串。这样可以保证函数永远不会因为非字符串值而出错。

如上一节所述，因为存在类型转换，所以应该避免在流控制语句中使用非布尔值作为条件。另外这也是可能导致类型错误的一个做法。来看下面的函数：

```
// 不安全的函数，非数组值可能导致错误
function reverseSort(values) {
  if (values) { // 不要!
    values.sort();
    values.reverse();
  }
}
```

reverseSort()函数可以使用数组的 sort()和 reverse()方法，将数组反向排序。由于 if 语句中的控制条件，任何非数组值都会被转换为 true，从而导致错误。另一个常见的错误是将参数与 null 比较，比如：

```
// 还是不安全的函数，非数组值可能导致错误
function reverseSort(values) {
  if (values != null){ // 不要!
    values.sort();
    values.reverse();
  }
}
```

用参数值与 null 比较只会保证不是两个值：null 和 undefined（对于使用相等和不相等操作符而言是等价的）。与 null 比较不足以保证适当的值，因此不要使用这种方式。出于同样的原因，也不推荐与 undefined 比较。

另一个错误的做法是在检测特性时只检查使用的特性。下面是一个例子：

```
// 仍是不安全的函数，非数组值可能导致错误
function reverseSort(values) {
  if (typeof values.sort === "function") { // 不要!
    values.sort();
    values.reverse();
  }
}
```

在这个例子中，代码检查了参数上是否存在 sort()方法。假如传入的参数确实有一个 sort()方法，但参数本身不是数组，那么在执行 reverse()时也会报告错误。如果知道预期的确切类型，那么最好使用 instanceof 来确定值的正确类型，如下所示：

```
// 安全，非数组值被忽略
function reverseSort(values) {
  if (values instanceof Array) { // 修复
    values.sort();
    values.reverse();
  }
}
```

最后一个 reverseSort()是安全的，它测试了 values 参数是不是 Array 的实例。这样，函数可以保证忽略非数组参数。

一般来说，原始类型的值应该使用 typeof 检测，而对象值应该使用 instanceof 检测。根据函数的用法，不一定要检查每个参数的数据类型，但对外的任何 API 都应该做类型检查以保证正确执行。

#### 4. 通信错误

随着 Ajax 编程的出现，Web 应用程序在运行期间动态加载数据和功能成为常见的情形。JavaScript 和服务器之间的通信也会出现错误。

第一种错误是 URL 格式或发送数据的格式不正确。通常，在把数据发送到服务器之前没有用