第 10 章

抛出自定义错误

当年，最令我迷惑的是编程语言有“创建”错误的能力。看到 Java 中的 throw 操作符，我的第一反应是：“嗯，愚蠢！为什么你要引发一个错误呢？”错误是敌人——是需要竭力避免的——所以“创建”错误的能力在我看来是无益的，是语言的危险面。我曾想，在 JavaScript 中包含那个操作符也是愚蠢的，何况起初人们还不怎么了解这门语言。现在，在积累了大量的经验之后，我已然很热衷于抛出自己的错误。

在 JavaScript 中抛出错误是一门艺术。摸清楚代码中哪里合适抛出错误是需要时间的。因此，一旦搞清楚了这一点，调试代码的时间将大大缩短，对代码的满意度将急剧提升。

10.1 错误的本质

当某些非期望的事情发生时程序就引发一个错误。也许是给一个函数传递了一个不正确的值，或者是数学运算碰到了一个无效的操作数。编程语言定义了一组基本的规则，当偏离了这些规则时将导致错误，然后开发者能修复代码。如果错误没有被抛出或者报告给你的话，调试是非常困难的。如果所有的失败都是悄无声息的，首要的问题是那必将消耗你大量的时间才能发现它，更不要说单独隔离并修复它了。所以，错误是开发者的朋友，而不是敌人。

错误常常在非期望的地点、不恬当的时机跳出来，这很麻烦。更糟糕的是，默认的错误消息通常太简洁而无法解释到底什么东西出错了。JavaScript 错误消息以信息

稀少、隐晦含糊而臭名昭著（特别是在老版本的 IE 中），这只会让问题更加复杂化。想象一下，如果跳出一个错误能这样描述：“由于发生这些情况，该函数调用失败”。那么，调试任务马上就会变得更加简单，这正是抛出自己的错误的好处。

像内置的失败案例一样来考虑错误是非常有帮助的。在代码某个特殊之处计划一个失败总比要在所有的地方都预期失败简单的多。在产品设计上，这是非常普遍的实践经验，而不仅仅是在代码编写方面。汽车尚有碰撞力吸收区域，这些区域框架的设计旨在撞击发生时以可预测的方式崩塌。知道一个碰撞到来时这些框架将如何反应——特别是，哪些部分将失败——制造商将能保证乘客的安全。你的代码也可以用这种方法来创建。

10.2 在JavaScript中抛出错误

毫无疑问，在 JavaScript 中抛出错误要比在任何其他语言中做同样的事情更加有价值，这归咎于 Web 端调试的复杂性。可以使用 throw 操作符，将提供的一个对象作为错误抛出。任何类型的对象都可以作为错误抛出，然而，Error 对象是最常用的。

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

内置的 Error 类型在所有的 JavaScript 实现中都是有效的，它的构造器只接受一个参数，指代错误消息（message）。当以这种方式抛出错误时，如果没有通过 try-catch 语句来捕获的话，浏览器通常直接显示该消息（message 字符串）。当今大多数浏览器都有一个控制台 （console），一旦发生错误都会在这里输出错误信息。换言之，任何你抛出的和没抛出的错误都被以相同的方式来对待。

缺乏经验的开发者有时直接将一个字符串作为错误抛出，如：

```
// 不好的写法
throw "message";
```

这样做确实能够抛出一个错误，但不是所有的浏览器做出的响应都会按照你的预期。Firefox、Opera 和 Chrome 都将显示一条 “uncaught exception”消息，同时它们包含上述消息字符串。Safari 和 IE 只是简陋地抛出一个“uncaught exception”错误，完全不提供上述消息字符串，这种方式对调试无益。

显然，如果愿意，你可以抛出任何类型的数据。没有任何规则约束不能是特定的数据类型。

```
throw { name: "Nicholas" };
throw true;
throw 12345;
throw new Date();
```

就一件事情需要牢记，如果没有通过 try-catch 语句捕获，抛出任何值都将引发一个错误。Firefox、Opera 和 Chrome 都会在该抛出的值上调用 String()函数，来完成错误消息的显示逻辑，但 Safari 和 IE 不是这样的。针对所有的浏览器，唯一不出差错的显示自定义的错误消息的方式就是用一个 Error 对象。

10.3 抛出错误的好处

抛出自己的错误可以使用确切的文本供浏览器显示。除了行和列的号码，还可以包含任何你需要的有助于调试问题的信息。我推荐总是在错误消息中包含函数名称，以及函数失败的原因。考察下面的函数。

```
function getDivs(element) {
    return element.getElementsByTagName("div");
}
```

这个函数旨在获取 element 元素下所有后代元素中的（div）元素。传递给函数要操作的 DOM（元素）为 null 值可能是件很常见的事情，但实际需要的是 DOM 元素。如果给这个函数传递 null 会发生什么情况呢？你会看到一个类似“object expected”的含糊的错误消息。然后，你要去看执行栈，再实际定位到源文件中的问题。通过抛出一个错误，调试会更简单。

```
function getDivs(element) {
    if (element && element.getElementsByTagName) {
        return element.getElementsByTagName("div");
    } else {
        throw new Error("getDivs(): Argument must be a DOM element.");
    }
}
```

现在给 getDivs()函数抛出一个错误，任何时候只要 element 不满足继续执行的条件，就会抛出一个错误明确地陈述发生的问题。如果在浏览器控制台中输出该错误，你

马上能开始调试，并知道最有可能导致该错误的原因是调用函数试图用一个值为 null 的 DOM 元素去做进一步的事情。

我倾向于认为抛出错误就像给自己留下告诉自己为什么失败的便签。

10.4 何时抛出错误

理解了如何抛出错误只是等式的一个部分，另外一部分就是要理解什么时候抛出错误。由于 JavaScript 没有类型和参数检查，大量的开发者错误地假设他们自己应该实现每个函数的类型检查。这种做法并不实际，并且会对脚本的整体性能造成影响。考察下面的函数，它试图实现充分的类型检查。

```
// 不好的写法：检查了太多的错误
function addClass(element, className) {
    if (!element || typeof element.className != "string") {
        throw new Error("addClass(): First argument must be a DOM element.");
    }
    if (typeof className != "string") {
        throw new Error("addClass(): Second argument must be a string.");
    }
    element.className += " " + className;
}
```

这个函数本来只是简单地给一个给定的元素增加一个 CSS 类名（className），因此，函数的大部分工作变成了错误检查。纵然它能在每个函数中检查每个参数（模仿静态类型语言），在 JavaScript 中这么做也会引起过度的杀伤。辨识代码中哪些部分在特定的情况下最有可能导致失败，并只在那些地方抛出错误才是关键所在。

在上例中，最有可能引发错误的是给函数传递一个 null 引用值。如果第二个参数是 null 或者一个数字或者一个布尔量是不会抛出错误的，因为 JavaScript 会将其强制转换为字符串。那意味着导致 DOM 元素的显示不符合期望，但这并不至于提高到严重错误的程度。所以，我只会检查 DOM 元素。

```
// 好的写法
function addClass(element, className) {
    if (!element || typeof element.className != "string") {
        throw new Error("addClass(): First argument must be a DOM element.");
    }
    element.className += " " + className;
}
```

如果一个函数只被已知的实体调用，错误检查很可能没有必要（这个案例是私有函数）；如果不能提前确定函数会被调用的所有地方，你很可能需要一些错误检查。这就更有可能从抛出自己的错误中获益。抛出错误最佳的地方是在工具函数中，如addClass()函数，它是通用脚本环境中的一部分，会在很多地方使用。更准确的案例是 JavaScript 类库。

针对已知条件引发的错误，所有的 JavaScript 类库都应该从它们的公共接口里抛出错误。如 jQuery、YUI 和 Dojo 等大型的库，不可能预料你在何时何地调用了它们的函数。当你做错事的时候通知你是它们的责任，因为你不可能进入库代码中去调试错误的原因。函数调用栈应该在进入库代码接口时就终止，不应该更深了。没有比看到由一打库代码中函数调用时发生一个错误更加糟糕的事情了吧，库的开发者应该承担起防止类似情况发生的责任。

私有 JavaScript 库也类似。许多 Web 应用程序都有自己专用的内置的 JavaScript 库或“拿来”一些有名的开源类库（类似 jQuery）。类库提供了对脏的实现细节的抽象，目的是让开发者用得更爽。抛出错误有助于对开发者安全地隐藏这些脏的实现细节。

这里有一些关于抛出错误很好的经验法则。

- 一旦修复了一个很难调试的错误，尝试增加一两个自定义错误。当再次发生错误时，这将有助于更容易地解决问题。
- 如果正在编写代码，思考一下：“我希望[某些事情]不会发生，如果发生，我的代码会一团糟糕”。这时，如果“某些事情”发生，就抛出一个错误。
- 如果正在编写的代码别人（不知道是谁）也会使用，思考一下他们使用的方式，在特定的情况下抛出错误。

请牢记，我们目的不是防止错误，而是在错误发生时能更加容易地调试。

10.5 try-catch语句

JavaScript 提供了 try-catch 语句，它能在浏览器处理抛出的错误之前来解析它。可能引发错误的代码放在 try 块中，处理错误的代码放在 catch 块中。例如：

```
try {
    somethingThatMightCauseAnError();
} catch (ex) {
    handleError(ex);
}
```

当在 try 块中发生了一个错误时，程序立刻停止执行，然后跳到 catch 块，并传入一个错误对象。检查该对象可以确定从错误中恢复的最佳动作。

当然，还可以增加一个 finally 块。finally 块中的代码不管是否有错误发生，最后都会被执行。例如：

```
try {
    somethingThatMightCauseAnError();
} catch (ex) {
    handleError(ex);
} finally {
    continueDoingOtherStuff();
}
```

在某些情况下，finally 块工作起来有点复杂（微妙）。例如，如果 try 块中包含了一个 return 语句，实际上它必须等到 finally 块中的代码执行后才能返回。由于这个原因，finally 其实不太常用，但如果处理错误必要，它仍然是处理错误的一个强大的工具。

使用throw还是try-catch

通常，开发者很难敏锐地判断是抛出一个错误还是用 try-catch 来捕获一个错误。错误只应该在应用程序栈中最深的部分抛出，如上面的讨论，就像 JavaScript 类库的代码。任何处理应用程序特定逻辑的代码都应该有错误处理的能力，并且捕获从底层组件中抛出的错误。

应用程序逻辑总是知道调用某个特定函数的原因，因此也是最适合处理错误的。千万不要将 try-catch 中的 catch 块留空，你应该总是写点什么来处理错误。例如，不要像下面这样做。

```
// 不好的写法
try {
    somethingThatMightCauseAnError();
} catch (ex) {
```

```
        // 空
    }
```

如果知道可能要发生错误，那肯定知道如何从错误中恢复。确切地说，如何从错误中恢复在开发模式中与实际放到生产环节中是不一样的，这没关系。最重要的是，你实实在在地在处理错误，而不是忽略它。

10.6 错误类型

ECMA-262 规范指出了 7 种错误类型。当不同错误条件发生时，这些类型在 JavaScript 引擎中都有用到，当然我们也可以手动创建它们。

`Error`

所有错误的基本类型。实际上引擎从来不会抛出该类型的错误。

`EvalError`

通过 eval()函数执行代码发生错误时抛出。

`RangeError`

一个数字超出它的边界时抛出——例如，试图创建一个长度为−20 的数组（new Array(−20)）。该错误在正常的代码执行中非常罕见。

`ReferenceError`

期望的对象不存在时抛出——例如，试图在一个 null 对象引用上调用一个函数。

`SyntaxError`

给 eval()函数传递的代码中有语法错误时抛出。

`TypeError`

变量不是期望的类型时抛出。例如，new 10 或“prop” in true。

`URIError`

给 encodeURI()、encodeURIComponent()、decodeURI()或者 decodeURIComponent()等函数传递格式非法的 URI 字符串时抛出。

理解错误的不同类型可以帮助我们更容易地处理它。所有的错误类型都继承自Error，所以用 instanceof Error 检查其类型得不到任何有用的信息。通过检查特定的错误类型可以更可靠地处理错误。

```
try {
    // 有些代码引发了错误
} catch (ex) {
    if (ex instanceof TypeError) {
        // 处理 TypeError 错误
    } else if (ex instanceof ReferenceError) {
        // 处理 ReferenceError 错误
    } else {
        // 其他处理
    }
}
```

如果抛出自己的错误，并且是数据类型而不是一个错误，你可以非常轻松地区分自己的错误和浏览器的错误类型的不同。但是，抛出实际类型的错误与抛出其他类型的对象相比，有几大优点。

首先，如上讨论，在浏览器正常错误处理机制中会显示错误消息。其次，浏览器给抛出的 Error 对象附加了一些额外的信息。这些信息不同浏览器各不相同，但它们为错误提供了如行、列号等上下文信息，在有些浏览器中也提供了堆栈和源代码信息。当然，如果用了 Error 的构造器，你就丧失了区分自己抛出的错误和浏览器错误的能力。

解决方案就是创建自己的错误类型，让它继承自 Error。这种做法允许你提供额外的信息，同时可区别于浏览器抛出的错误。可以用如下的模式来创建自定义的错误类型。

```
function MyError(message) {
    this.message = message;
}
MyError.prototype = new Error();
```

这段代码有两个重要的部分：message 属性，浏览器必须要知道的错误消息字符串；设置 prototype 为 Error 的一个实例，这样对 JavaScript 引擎而言就标识它是一个错误对象了。接下来就可以抛出一个 MyError 的实例对象，使得浏览器能像处理原生错误一样做出响应。

```
throw new MyError("Hello world!");
```

提醒一下，该方法在 IE 8 和更早的浏览器中不显示错误消息。相反，会看见那个通用的“Exception thrown but not caught”消息。这个方法最大的好处是，自定义错误类型可以检测自己的错误。

```
try {
    // 有些代码引发了错误
} catch (ex) {
    if (ex instanceof MyError) {
        // 处理自己的错误
    } else {
        // 其他处理
    }
}
```

如果总是捕获你自己抛出的所有错误，那么 IE 的那点儿小愚蠢也不足为道了。在一个正确的错误处理系统中获得的好处是巨大的。该方法可以给出更多、更灵活的信息，告知开发者如何正确地处理错误。