07 错误处理

在执行JavaScript代码的时候，有些情况下会发生错误。

错误分两种，一种是程序写的逻辑不对，导致代码执行异常。例如：

```
var s = null;
var len = s.length; // TypeError：null变量没有length属性
```

对于这种错误，要修复程序。

一种是执行过程中，程序可能遇到无法预测的异常情况而报错，例如，网络连接中断，读取不存在的文件，没有操作权限等。

对于这种错误，我们需要处理它，并可能需要给用户反馈。

错误处理是程序设计时必须要考虑的问题。对于C这样贴近系统底层的语言，错误是通过错误码返回的：

```
int fd = open("/path/to/file", O_RDONLY);
if (fd == -1) {
    printf("Error when open file!");
} else {
    // TODO
}
```

通过错误码返回错误，就需要约定什么是正确的返回值，什么是错误的返回值。上面的`open()`函数约定返回`-1`表示错误。

显然，这种用错误码表示错误在编写程序时十分不便。

因此，高级语言通常都提供了更抽象的错误处理逻辑try ... catch ... finally，JavaScript也不例外。

try ... catch ... finally

使用try ... catch ... finally处理错误时，我们编写的代码如下：

```
'use strict';
var r1, r2, s = null;
try {
    r1 = s.length; // 此处应产生错误
    r2 = 100; // 该语句不会执行
} catch (e) {
    console.log('出错了：' + e);
} finally {
    console.log('finally');
}
console.log('r1 = ' + r1); // r1应为undefined
console.log('r2 = ' + r2); // r2应为undefined
// 直接运行
```

```
出错了：TypeError: Cannot read property 'length' of null
finally
r1 = undefined
r2 = undefined
```

运行后可以发现，输出提示类似“出错了：TypeError: Cannot read property 'length' of null”。

我们来分析一下使用try ... catch ... finally的执行流程。

当代码块被`try { ... }`包裹的时候，就表示这部分代码执行过程中可能会发生错误，一旦发生错误，就不再继续执行后续代码，转而跳到`catch`块。`catch (e) { ... }`包裹的代码就是错误处理代码，变量`e`表示捕获到的错误。最后，无论有没有错误，`finally`一定会被执行。

所以，有错误发生时，执行流程像这样：

1. 先执行`try { ... }`的代码；
2. 执行到出错的语句时，后续语句不再继续执行，转而执行`catch (e) { ... }`代码；
3. 最后执行`finally { ... }`代码。

而没有错误发生时，执行流程像这样：

1. 先执行`try { ... }`的代码；
2. 因为没有出错，`catch (e) { ... }`代码不会被执行；
3. 最后执行`finally { ... }`代码。

最后请注意，`catch`和`finally`可以不必都出现。也就是说，`try`语句一共有三种形式：

完整的try ... catch ... finally：

```
try {
    ...
} catch (e) {
    ...
} finally {
    ...
}
```

只有try ... catch，没有finally：

```
try {
    ...
} catch (e) {
    ...
}
```

只有try ... finally，没有catch：

```
try {
    ...
} finally {
    ...
}
```

错误类型

JavaScript有一个标准的`Error`对象表示错误，还有从`Error`派生的`TypeError`、`ReferenceError`等错误对象。我们在处理错误时，可以通过`catch(e)`捕获的变量`e`访问错误对象：

```
try {
    ...
} catch (e) {
    if (e instanceof TypeError) {
        alert('Type error!');
    } else if (e instanceof Error) {
        alert(e.message);
    } else {
        alert('Error: ' + e);
    }
}
```

使用变量`e`是一个习惯用法，也可以以其他变量名命名，如`catch(ex)`。

抛出错误

程序也可以主动抛出一个错误，让执行流程直接跳转到`catch`块。抛出错误使用`throw`语句。

例如，下面的代码让用户输入一个数字，程序接收到的实际上是一个字符串，然后用`parseInt()`转换为整数。当用户输入不合法的时候，我们就抛出错误：

```
'use strict';
var r, n, s;
try {
    s = prompt('请输入一个数字');
    n = parseInt(s);
    if (isNaN(n)) {
        throw new Error('输入错误');
    }
    // 计算平方:
    r = n * n;
    console.log(n + ' * ' + n + ' = ' + r);
} catch (e) {
    console.log('出错了: ' + e);
}
```

```
出错了: Error: 输入错误
```

实际上，JavaScript允许抛出任意对象，包括数字、字符串。但是，最好还是抛出一个Error对象。

最后，当我们用catch捕获错误时，一定要编写错误处理语句：

```
var n = 0, s;
try {
    n = s.length;
} catch (e) {
    console.log(e);
}
console.log(n);
```

哪怕仅仅把错误打印出来，也不要什么也不干：

```
var n = 0, s;
try {
    n = s.length;
} catch (e) {
}
console.log(n);
```

因为catch到错误却什么都不执行，就不知道程序执行过程中到底有没有发生错误。

处理错误时，请不要简单粗暴地用`alert()`把错误显示给用户。教程的代码使用`alert()`是为了便于演示。

错误传播

如果代码发生了错误，又没有被try ... catch捕获，那么，程序执行流程会跳转到哪呢？

```
function getLength(s) {
    return s.length;
}

function printLength() {
    console.log(getLength('abc')); // 3
    console.log(getLength(null)); // Error!
}

printLength();
```

如果在一个函数内部发生了错误，它自身没有捕获，错误就会被抛到外层调用函数，如果外层函数也没有捕获，该错误会一直沿着函数调用链向上抛出，直到被JavaScript引擎捕获，代码终止执行。

所以，我们不必在每一个函数内部捕获错误，只需要在合适的地方来个统一捕获，一网打尽：

```
'use strict';
function main(s) {
    console.log('BEGIN main()');
    try {
        foo(s);
    } catch (e) {
        console.log('出错了: ' + e);
    }
    console.log('END main()');
}

function foo(s) {
    console.log('BEGIN foo()');
    bar(s);
    console.log('END foo()');
}

function bar(s) {
    console.log('BEGIN bar()');
    console.log('length = ' + s.length);
    console.log('END bar()');
}

main(null);
```

```
BEGIN main()
BEGIN foo()
BEGIN bar()
出错了：TypeError: Cannot read property 'length' of null
END main()
```

当`bar()`函数传入参数`null`时，代码会报错，错误会向上抛给调用方`foo()`函数，`foo()`函数没有try ... catch语句，所以错误继续向上抛给调用方`main()`函数，`main()`函数有try ... catch语句，所以错误最终在`main()`函数被处理了。

至于在哪些地方捕获错误比较合适，需要视情况而定。

异步错误处理

编写JavaScript代码时，我们要时刻牢记，JavaScript引擎是一个事件驱动的执行引擎，代码总是以单线程执行，而回调函数的执行需要等到下一个满足条件的事件出现后，才会被执行。

例如，`setTimeout()`函数可以传入回调函数，并在指定若干毫秒后执行：

```
function printTime() {
    console.log('It is time!');
}

setTimeout(printTime, 1000);
console.log('done');
```

上面的代码会先打印`done`，1秒后才会打印`It is time!`。

如果`printTime()`函数内部发生了错误，我们试图用try包裹`setTimeout()`是无效的：

```
'use strict';
function printTime() {
    throw new Error();
}

try {
    setTimeout(printTime, 1000);
    console.log('done');
} catch (e) {
    console.log('error');
}
```

```
done
```

原因就在于调用`setTimeout()`函数时，传入的`printTime`函数并未立刻执行！紧接着，JavaScript引擎会继续执行`console.log('done');`语句，而此时并没有错误发生。直到1秒钟后，执行`printTime`函数时才发生错误，但此时除了在`printTime`函数内部捕获错误外，外层代码并无法捕获。

所以，涉及到异步代码，无法在调用时捕获，原因就是在捕获的当时，回调函数并未执行。

类似的，当我们处理一个事件时，在绑定事件的代码处，无法捕获事件处理函数的错误。

例如，针对以下的表单：

```
<form>
    <input id="x"> + <input id="y">
    <button id="calc" type="button">计算</button>
</form>
```

我们用下面的代码给button绑定click事件：

```
'use strict';

var $btn = $('#calc');

// 取消已绑定的事件:
$btn.off('click');
try {
    $btn.click(function () {
        var
            x = parseFloat($('#x').val()),
            y = parseFloat($('#y').val()),
            r;
        if (isNaN(x) || isNaN(y)) {
            throw new Error('输入有误');
        }
        r = x + y;
        alert('计算结果: ' + r);
    });
} catch (e) {
    alert('输入有误! ');
}
```

但是，用户输入错误时，处理函数并未捕获到错误。请修复错误处理代码。