错误处理机制

《JavaScript 教程》作者：阮一峰

1 Error 实例对象

JavaScript 解析或运行时，一旦发生错误，引擎就会抛出一个错误对象。JavaScript 原生提供Error构造函数，所有抛出的错误都是这个构造函数的实例。

```
var err = new Error('出错了');
err.message // "出错了"
```

上面代码中，我们调用Error构造函数，生成一个实例对象err。Error构造函数接受一个参数，表示错误提示，可以从实例的message属性读到这个参数。抛出Error实例对象以后，整个程序就中断在发生错误的地方，不再往下执行。

JavaScript 语言标准只提到，Error实例对象必须有message属性，表示出错时的提示信息，没有提到其他属性。

- message：错误提示信息
- name：错误名称（非标准属性）
- stack：错误的堆栈（非标准属性）

大多数 JavaScript 引擎，对Error实例还提供name和stack属性，分别表示错误的名称和错误的堆栈，但它们是非标准的，不是每种实现都有。

使用name和message这两个属性，可以对发生什么错误有一个大概的了解。

```
if (error.name) {
  console.log(error.name + ': ' + error.message);
}
```

stack属性用来查看错误发生时的堆栈.

```
function throwit() {
  throw new Error('');
}

function catchit() {
  try {
    throwit();
  } catch(e) {
    console.log(e.stack); // print stack trace
  }
}
```

```
catchit()
// Error
//    at throwit (~/examples/throwcatch.js:9:11)
//    at catchit (~/examples/throwcatch.js:3:9)
//    at repl:1:5
```

上面代码中，错误堆栈的最内层是throwit函数，然后是catchit函数，最后是函数的运行环境。

2 原生错误类型

Error 实例对象是最一般的错误类型，在它的基础上，JavaScript 还定义了 Error 的6个派生对象。

这6种派生错误，连同原始的Error对象，都是构造函数。开发者可以使用它们，手动生成错误对象的实例。这些构造函数都接受一个参数，代表错误提示信息（message）。

```
var err1 = new Error('出错了！');
var err2 = new RangeError('出错了，变量超出有效范围！');
var err3 = new TypeError('出错了，变量类型无效！');

err1.message // "出错了！"
err2.message // "出错了，变量超出有效范围！"
err3.message // "出错了，变量类型无效！"
```

1. SyntaxError 对象

SyntaxError 对象是解析代码时发生的语法错误。例如：

```
// 变量名错误
var 5a;
// Uncaught SyntaxError: Invalid or unexpected token

// 缺少括号
console.log 'hello');
// Uncaught SyntaxError: Unexpected string
```

2. ReferenceError 对象

ReferenceError 对象是引用一个不存在的变量时发生的错误。

```
// 使用一个不存在的变量
unknownVariable
// Uncaught ReferenceError: unknownVariable is not defined
```

另一种触发场景是，将一个值分配给无法分配的对象，比如对函数的运行结果赋值。

```
// 等号左侧不是变量
console.log() = 1
// Uncaught ReferenceError: Invalid left-hand side in assignment
```

3. RangeError 对象

RangeError对象是一个值超出有效范围时发生的错误。主要有几种情况，一是数组长度为负数，二是Number对象的方法参数超出范围，以及函数堆栈超过最大值。

```
// 数组长度不得为负数
new Array(-1)
// Uncaught RangeError: Invalid array length
```

4. TypeError 对象

TypeError对象是变量或参数不是预期类型时发生的错误。

比如，对字符串、布尔值、数值等原始类型的值使用new命令，就会抛出这种错误，因为new命令的参数应该是一个构造函数。

```
new 123
// Uncaught TypeError: number is not a func

var obj = {};
obj.unknownMethod()
// Uncaught TypeError: obj.unknownMethod is not a function
```

上面代码的第二种情况，调用对象不存在的方法，也会抛出TypeError错误，因为obj.unknownMethod的值是undefined，而不是一个函数。

5. URLError 对象

URIError对象是 URI 相关函数的参数不正确时抛出的错误，主要涉及encodeURI()、decodeURI()、encodeURIComponent()、decodeURIComponent()、escape()和unescape()这六个函数。

```
decodeURI('%2')
// URIError: URI malformed
```

6. EvalError 对象

eval函数没有被正确执行时，会抛出EvalError错误。该错误类型已经不再使用了，只是为了保证与以前代码兼容，才继续保留。

3 自定义错误

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

UserError.prototype = new Error();
UserError.prototype.constructor = UserError;
```

上面代码自定义一个错误对象UserError，让它继承Error对象。然后，就可以生成这种自定义类型的错误了。

```
new UserError('这是自定义的错误！');
```

4 throw 语句

throw 语句的作用是抛出一个错误。对于 JS 引擎来说，遇到 throw 语句 程序就终止了。

```
if (x <= 0) {
  throw new Error('x 必须为正数');
}
// Uncaught ReferenceError: x is not defined
```

throw也可以抛出自定义错误。

```
function UserError(message) {
  this.message = message || '默认信息';
  this.name = 'UserError';
}

throw new UserError('出错了！');
// Uncaught UserError {message: "出错了！", name: "UserError"}
```

实际上，throw可以抛出任何类型的值。也就是说，它的参数可以是任何值。

```
// 抛出一个字符串
throw 'Error！';
// Uncaught Error!

// 抛出一个数值
throw 42;
// Uncaught 42

// 抛出一个布尔值
throw true;
// Uncaught true

// 抛出一个对象
throw {
  toString: function () {
    return 'Error!';
  }
};
// Uncaught {toString: ƒ}
```

5 try ... catch 结构

JavaScript 提供了try...catch结构，允许对错误进行处理，选择是否往下执行。

```
try {
  throw new Error('出错了!');
} catch (e) {
  console.log(e.name + ": " + e.message);
  console.log(e.stack);
}
// Error: 出错了!
//   at <anonymous>:3:9
//   ...
```

上面代码中，try代码块抛出错误，JavaScript 引擎就立即把代码的执行，转到catch代码块，或者说错误被catch代码块捕获了。catch接受一个参数，表示try代码块抛出的值。

```
try {
  throw "出错了";
} catch (e) {
  console.log(111);
}
console.log(222);
// 111
// 222
```

上面代码中，try代码块抛出的错误，被catch代码块捕获并处理后，程序会继续向下执行。

6 finally 代码块

try...catch结构允许在最后添加一个finally代码块，表示不管是否出现错误，都必需在最后运行的语句。

```
function cleansUp() {
  try {
    throw new Error('出错了……');
    console.log('此行不会执行');
  } finally {
    console.log('完成清理工作');
  }
}

cleansUp()
// 完成清理工作
// Uncaught Error: 出错了……
//     at cleansUp (<anonymous>:3:11)
//     at <anonymous>:10:1
```

上面代码中，由于没有catch语句块，一旦发生错误，代码就会中断执行。中断执行之前，会先执行finally代码块，然后再向用户提示报错信息。

```
function idle(x) {
  try {
    console.log(x);
```

```
    return 'result';
  } finally {
    console.log('FINALLY');
  }
}

idle('hello')
// hello
// FINALLY
```

上面代码中，try代码块没有发生错误，而且里面还包括return语句，但是finally代码块依然会执行。而且，这个函数的返回值还是result。

下面的例子说明，return语句的执行是排在finally代码之前，只是等finally代码执行完毕后才返回。

```
var count = 0;
function countUp() {
  try {
    return count;
  } finally {
    count++;
  }
}

countUp()
// 0
count
// 1
```

下面是finally代码块用法的典型场景。

```
openFile();

try {
  writeFile(Data);
} catch(e) {
  handleError(e);
} finally {
  closeFile();
}
```

上面代码首先打开一个文件，然后在try代码块中写入文件，如果没有发生错误，则运行finally代码块关闭文件；一旦发生错误，则先使用catch代码块处理错误，再使用finally代码块关闭文件。

下面的例子充分反映了try...catch...finally这三者之间的执行顺序。

```
function f() {
  try {
    console.log(0);
    throw 'bug';
  } catch(e) {
```

```
    console.log(1);
    return true; // 这句原本会延迟到 finally 代码块结束再执行
    console.log(2); // 不会运行
  } finally {
    console.log(3);
    return false; // 这句会覆盖掉前面那句 return
    console.log(4); // 不会运行
  }

  console.log(5); // 不会运行
}

var result = f();
// 0
// 1
// 3

result
// false
```

catch代码块之中，触发转入finally代码块的标志，不仅有return语句，还有throw语句。

```
function f() {
  try {
    throw '出错了! ';
  } catch(e) {
    console.log('捕捉到内部错误');
    throw e; // 这句原本会等到finally结束再执行
  } finally {
    return false; // return 语句后面的语句不再执行
  }
}

try {
  f();
} catch(e) {
  // 此处不会执行
  console.log('caught outer "bogus"');
}

//  捕捉到内部错误
```

try代码块内部，还可以再使用try代码块。

```
try {
  try {
    consle.log('Hello world!'); // 报错
  }
  finally {
    console.log('Finally');
  }
  console.log('Will I run?');
```

```
} catch(error) {
  console.error(error.message);
}
// Finally
// consle is not defined
```