错误处理机制

目录

- Error 实例对象
- 原生错误类型
  - SyntaxError 对象
  - ReferenceError 对象
  - RangeError 对象
  - TypeError 对象
  - URIError 对象
  - EvalError 对象
  - 总结
- 自定义错误
- throw 语句
- try...catch 结构
- finally 代码块
- 参考连接

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。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。

- **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）

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。

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 还定义了其他 6种错误对象。也就是说，存在 `Error`的 6个派生对象。

SyntaxError 对象

`SyntaxError`对象是解析代码时发生的语法错误。

```
// 变量名错误
var 1a;
// Uncaught SyntaxError: Invalid or unexpected token

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

上面代码的错误，都是在语法解析阶段就可以发现，所以会抛出`SyntaxError`。第一个错误提示是“token 非法”，第二个错误提示是“字符串不符合要求”。

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
```

上面代码对函数`console.log`的运行结果赋值，结果引发了`ReferenceError`错误。

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
```

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: 123 is not a constructor

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。

URIError 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
```

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。

总结

以上这 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！"
```

自定义错误

除了 JavaScript 原生提供的七种错误对象，还可以定义自己的错误对象。

```
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！');
```

throw 语句

`throw` 语句的作用是手动中断程序执行，抛出一个错误。

```
var x = -1;

if (x <= 0) {
  throw new Error('x 必须为正数');
}
// Uncaught Error: x 必须为正数
```

上面代码中，如果变量 `x` 小于等于 `0`，就手动抛出一个错误，告诉用户 `x` 的值不正确，整个程序就会在这里中断执行。可以看到，`throw` 抛出的错误就是它的参数，这里是一个 `Error` 对象的实例。

`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"}
```

上面代码中，`throw`抛出的是一个`UserError`实例。

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: f}
```

对于 JavaScript 引擎来说，遇到`throw`语句，程序就中止了。引擎会接收到`throw`抛出的信息，可能是一个错误实例，也可能是其他类型的值。

try...catch 结构

一旦发生错误，程序就中止执行了。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（上例用的是`throw`语句），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。

如果你不确定某些代码是否会报错，就可以把它们放在`try...catch`代码块之中，便于进一步对错误进行处理。

```
try {
  f();
} catch(e) {
  // 处理错误
}
```

上面代码中，如果函数`f`执行报错，就会进行`catch`代码块，接着对错误进行处理。

`catch`代码块捕获错误之后，程序不会中断，会按照正常流程继续执行下去。

```
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`代码块捕获后，程序会继续向下执行。

`catch`代码块之中，还可以再抛出错误，甚至使用嵌套的`try...catch`结构。

```
var n = 100;

try {
  throw n;
```

```
} catch (e) {
  if (e <= 50) {
    // ...
  } else {
    throw e;
  }
}
// Uncaught 100
```

上面代码中，`catch`代码之中又抛出了一个错误。

为了捕捉不同类型的错误，`catch`代码块之中可以加入判断语句。

```
try {
  foo.bar();
} catch (e) {
  if (e instanceof EvalError) {
    console.log(e.name + ": " + e.message);
  } else if (e instanceof RangeError) {
    console.log(e.name + ": " + e.message);
  }
  // ...
}
```

上面代码中，`catch`捕获错误之后，会判断错误类型（`EvalError`还是`RangeError`），进行不同的处理。

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
```

上面代码说明，`return`语句里面的`count`的值，是在`finally`代码块运行之前就获取了。

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
```

上面代码中，`catch`代码块结束执行之前，会先执行`finally`代码块。

`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; // 直接返回
  }
}

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

//   捕捉到内部错误
```

上面代码中，进入`catch`代码块之后，一遇到`throw`语句，就会去执行`finally`代码块，其中有`return false`语句，因此就直接返回了，不再会回去执行`catch`代码块剩下的部分了。

`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
```

上面代码中，`try`里面还有一个`try`。内层的`try`报错（`console`拼错了），这时会执行内层的`finally`代码块，然后抛出错误，被外层的`catch`捕获。

参考连接

- Jani Hartikainen, JavaScript Errors and How to Fix Them

版本号 #2
由 壳壳虫 创建于 2018-01-13 09:38:32 UTC
由 壳壳虫 更新于 2021-05-12 00:08:11 UTC