# 错误处理机制

## 1. Error 实例对象

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"
```

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

### 2.1 SyntaxError 对象

`SyntaxError` 对象是解析代码时发生的语法错误。

```
const a; // Uncaught SyntaxError: Missing initializer in const declaration, 定义常量时光声明没赋值
var 1a; // Uncaught SyntaxError: Invalid or unexpected token, 变量名不能以数字开头
console.log 'hello'); // Uncaught SyntaxError: Unexpected string, 缺少括号
```

### 2.2 ReferenceError 对象

`ReferenceError` 对象是引用一个不存在的变量时发生的错误。

```
o1; // Uncaught ReferenceError: o1 is not defined
```

### 2.3 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。

```
const a = new Array(-1); // Uncaught RangeError: Invalid array length
```

**当使用构造函数声明一个数组时，如果参数只有一个值，那这个值表示函数的长度。参数大于一个时，表示数组各个位置上的成员。**

```
const a = new Array(3); // 3 表示长度
a; // (3) [empty × 3]
```

```
a.length; // 3

const a2 = new Array(3, 4); // 3、4 表示数组成员
a2; // (2) [3, 4]
```

### 2.4 TypeError 对象

TypeError 对象是变量或参数不是预期类型时发生的错误。

```
var o4 = {
  valueOf: function () {
    return {};
  },
  toString: function () {
    return {};
  },
};
Number(o4); // Uncaught TypeError: Cannot convert object to primitive value
```

Number() 函数期望参数是一个原始类型的值，或者可以转换为原始类型的值。上面代码无法将对象 o4 转换为原始类型的值，因为先调用 valueOf() 函数，再调用 toString() 函数均无法转换成原始类型。

### 2.5 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
```

### 2.6 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。

### 2.7 总结

以上 6 中派生错误连同 Error 对象，都是构造函数。可以利用构造函数的特性实例化，手动生成错误。

```
let typeErr = new TypeError("类型错误! ");
typeErr.message; // 类型错误!
```

## 3. throw 语句

throw 语句的作用是中断程序执行，抛出一个错误。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。

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

throw 42; // 抛出一个数值 // Uncaught 42

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

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

## 4. 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 TypeError("类型出错了!");
} catch (e) {
  console.log(e.name + ": " + e.message);
}
// TypeError: 类型出错了!
```

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。

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 new SyntaxError("有一个语法错误");
} catch (e) {
  console.log(e.name + ': ' + e.message);
  console.log("打印错误之后");
}
console.log("try...catch块之后");
// SyntaxError: 有一个语法错误
// 打印错误之后
// try...catch块之后
```

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。

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 {
  throw new TypeError("类型错误! ");
```

```
} catch (e) {
  if (e instanceof TypeError) {
    console.log("isTypeError", e.name + ": " + e.message);
  } else {
    console.log("isNotTypeError", e.name + ": " + e.message);
  }
}
// isTypeError TypeError: 类型错误!
```

## 5. finally

`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。

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。

`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
```

`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。