异常处理 Exception Handling

> C++ 的异常处理（**Exception Handling**）是用来处理程序在**运行时**出现的错误的一种机制。
>
> 它可以使程序在遇到错误时不中断，而是跳转到合适的位置进行错误处理，从而提高程序的健壮性和可靠性。

一、基本语法结构

C++ 使用以下三个关键字来实现异常处理：

- `try`：包含可能抛出异常的代码块。
- `throw`：在**检测**到错误时抛出异常。
- `catch`：**处理**抛出的异常。

🌟 例子：

```
#include <iostream>
using namespace std;

int divide(int a, int b) {
    if (b == 0)
        throw "Division by zero!";  // 抛出一个 const char* 类型异常
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        cout << "Result: " << result << endl;
    } catch (const char* msg) {
        cout << "Caught exception: " << msg << endl;
    }

    return 0;
}
```

二、异常处理流程

当执行throw语句后，

中断当前函数的正常执行流程，

并开始寻找合适的**catch块**来**处理异常**

🧩 1. 异常检测（检测错误）

程序在运行过程中，一旦检测到某种错误情况（如非法输入、除以零、文件打不开），就执行：

```
throw 异常对象;
```

- 异常对象可以是基本类型、类对象、字符串、标准异常等

- 异常对象可以是基本类型、类对象、字符串、标准异常等。
- 一旦执行 `throw`，**当前函数的后续代码不会继续执行。**

🧩 2. 栈展开（Stack Unwinding）

- `throw` 后程序会**跳出当前函数**，沿着调用栈**依次回退。**
- 如果当前函数没有处理这个异常，它会退出并继续向上查找谁调用了它。

🧩 3. 寻找匹配的 catch 块

- 程序在调用链中**查找第一个与 `throw` 类型匹配的 `catch` 语句块**：

```
try {
    // 可能抛出异常的代码
} catch (const MyException& e) {
    // 处理异常
}
```

- 一旦匹配，异常被捕获，程序转到 `catch` 中执行。
- 如果**找不到匹配的 `catch`**，最终会调用：

```
std::terminate();  // 程序终止运行
```

🧩 4. 处理异常

- 在 `catch` 块中，你可以：
  - 打印错误信息
  - 记录日志
  - 清理资源
  - 尝试恢复
  - 或者再次抛出异常（`throw;`）

🧩 5. 程序可能继续执行（可选）

- 如果异常处理后，程序仍可继续运行，就可以从 `catch` 块之后的语句继续执行。

三、throw 可以抛出的类型

throw 可以抛出的类型：

- 基本类型（如 `int`, `double`, `const char*`）
- 标准异常类（如 `std::invalid_argument`）
- 自定义类（继承自 `std::exception`）

标准库异常（<stdexcept>）

C++ 标准库中包含许多常用异常类，主要包括：

异常类型	含义
std::exception	所有标准异常的基类
std::runtime_error	运行时错误
std::logic_error	逻辑错误（如编程错误）
std::invalid_argument	无效参数
std::out_of_range	超出范围
std::overflow_error	溢出错误
std::underflow_error	下溢错误

上述异常的继承关系

```
std::exception  //无法确定具体异常类型时使用
├── std::logic_error
|   ├── std::invalid_argument
|   ├── std::domain_error
|   └── ...
└── std::runtime_error
    ├── std::range_error
    ├── std::overflow_error
    └── ...
```

主要的两大类是 std::logic_error 和 std::runtime_error

示例：

```
#include <stdexcept>
throw std::invalid_argument("Invalid input");
```

自定义异常类

推荐继承 std::exception 并重写 what() 方法：

```
#include <exception>
class MyException : public std::exception {//继承自std::exception
public:
    const char* what() const noexcept override {
        return "My custom exception!";
    }
};
```

原封不动地重新抛出

将捕获到的异常原封不动地重新抛出

```
throw;
```

> ⚠ **Warning**
>
> ```
> catch (const exception& e) {
>  throw e; // ❌ 拷贝构造一个新的异常对象 → 可能发生对象切片
> }
> ```
>
> 上面这种方式不是“原封不动”地抛出，而是**抛出异常的拷贝**，可能导致：
>
> - 多态失效（对象切片）
> - 丢失原始类型信息或栈追踪信息

四、异常捕获方式

```
#include <iostream>
#include <stdexcept>
using namespace std;

class DerivedException : public runtime_error {
public:
    DerivedException() : runtime_error("派生类异常") {}
    const char* what() const noexcept override {//重写what
        return "我是派生类!";
    }
};

int main() {
    try {
        throw DerivedException();
    }
    // catch (runtime_error e) { // 按值捕获，不会使用多态
    //     cout << e.what() << endl;
    // }//输出"派生类异常"
    catch(const runtime_error& e){ // 按引用捕获，会使用多态
        cout<<e.what()<<endl;
    }//输出"我是派生类!"
    return 0;
}
```

多个 catch 块与 catch-all

多个 catch：

```
try {
    // ...
} catch (const std::invalid_argument& e) {
    cout << "Invalid argument: " << e.what();
} catch (const std::exception& e) {
    cout << "Standard exception: " << e.what();
}
```

捕获所有异常：

用省略号 `...`(3点)

```
catch (...) {
    cout << "Unknown exception caught!";
}
```

五、异常处理注意事项

- 异常是**运行时错误**处理机制，不应该用于控制程序流程。
- 构造函数抛出异常时，对象不会构造完成，需小心资源管理。
- **RAII**（资源获取即初始化）和智能指针可帮助防止内存泄漏。
- 如果没有被 `catch`，程序会调用 `std::terminate()`，终止运行。

六、使用场景

- 文件打不开
- 除以零
- 数组越界
- 网络中断
- 用户输入非法

七、noexcept关键字

> `noexcept` 是 C++11 引入的一个关键字，用来**声明一个函数不会抛出异常**。它是异常安全性的一个承诺，有助于编译器优化代码，并提升程序的稳定性。

基本用法：

```
void f() noexcept {
    // 该函数保证不抛异常
}
```

特殊情况：

```
void g() noexcept {
    throw std::runtime_error("error");  // 程序会调用 std::terminate
}
```

若一个声明为noexcept的函数**实际抛出了异常**，程序会**立即调用**`std::terminate`