

真的勇士，敢于直面惨淡的人生，敢于  
正视淋漓的鲜血。

——鲁迅

# 异常处理

---

北京理工大学计算机学院  
金旭亮

# “Bug”与异常

---

# 一只著名的“飞蛾”

Bug




9/9

0800 Antan started  
1000 " stopped - antan ✓

1300 (032) MP - MC ~~1.582147000~~  
                  (033) PRO 2 2.130476415  
                              concl 2.130676415

Relays 6-2 in 033 failed special speed test  
in relay " 10.00 test.

1100 Started Cosine Tape (Sine check)  
1525 Started Multy Adder Test.

1545  Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 antan started.  
1700 closed down.

Relay 3375  
Relay 3370

# 两个计算机软件技术术语

Bug

程序代码中隐藏的错误

Debug

1. 修正代码中存在的错误的过程
2. 简称“程序调试”。

# 开发中可能会遇到的几种错误

## 语法错误

在编译时发现，修改代码后可以顺利通过编译。

## 运行时错误

程序代码能顺利通过编译，但在运行时出现错误（比如用户输入了无效的数据），如果处理不当，程序可能会非法退出。

## 逻辑错误

程序代码能顺利通过编译，在运行时也没有出现错误，但就是结果不对 .....

异常



运行时错误

# “真实的”异常

这种在程序运行时所引发的错误，被称为“异常（Exception）”，它表明程序执行期间出现了一个导致程序正常执行流程被中断的特殊问题，期望程序进行的某项操作没有能够完成。

```
0 个引用
static void Main(string[] args)
{
    WhatIsException();
    // InputNumber();

    Console.ReadKey();
}

1 个引用
static void WhatIsException()
{
    int i = 100, j = 0;
    Console.WriteLine(i/j);
}

0 个引用
private static void InputNumber()
```

示例：ExceptionDemo



CLR是处理这种“意外”的？



# 有关异常的基础知识

1

当程序运行出现错误时，CLR会创建一个异常对象。在.NET中，异常对象通常是一个Exception类（或其派生的子类）创建的对象。

2

如果异常出现，而应用程序又没有编写代码处理这一异常，CLR会强行结束整个进程。



# 异常的捕获与处理

---

# 异常的捕获与处理

```
try {  
    //可能发生运行错误的代码块 ①  
}  
catch ( 异常类型 异常对象引用 ) {  
    //用于处理异常的代码块 ②  
}  
finally{  
    //用于“善后”的代码块 ③  
}
```

当程序正常运行时，程序的执行流程为：

第 (1) 块 → 第 (3) 块

当第 (1) 块中有代码引发了一个异常时，程序的执行流程为：

第 (1) 块 → 第 (2) 块 → 第 (3) 块

看上去很简单，但实际上不简单：



用户可能犯的错误

未输入而直接敲回车

输入的字符串无法转换为数字

输入过大的数字

输入对特定场景无效的  
数字

.....

1 个引用

```
private static void InputNumber()
{
    Console.WriteLine("请输入一个正整数：");
    try
    {
        //尝试将用户输入的字符串转换为整数
        int value = Convert.ToInt32(Console.ReadLine());
        if (value <= 0)
        {
            //发现非法的数据，“主动”抛出一个异常
            throw new InvalidOperationException("你输入的不是正整数！");
        }
        Console.WriteLine("您输入的数字是：{0}", value);
    }
    catch (FormatException)
    {
        Console.WriteLine("输入的字符串无法转换为数字");
    }
    catch (OverflowException)
    {
        Console.WriteLine("你输入的数字太大了！");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        Console.WriteLine("敲任意键退出.....");
    }
}
```

## Exception是最顶层的异常基类

将可能出现异常的代码，封装到try语句块中

连续多个catch语句块，捕获并处理多种异常，利用了多态特性

不管程序运行时是否出现异常，finally块中的语句总会被执行

## .NET异常基类Exception的重要属性和方法

名称	用途
e.GetType()	获取异常的类型。
e.Message	告诉用户发生了什么事。
e.StackTrace	确定错误发生的位置，如果有可用的调试信息（即有<程序名>.pdb文件存在），还可显示源文件名和程序行号。

注：上表中的e代表Exception对象

```

0
0
0
0
0
0
0
0
0
0
在输出values数组值时发生数组越界错误
异常种类:IndexOutOfRangeException
系统给出的出错信息: 索引超出了数组界限。
系统调用堆栈信息: 在 ExceptionInfo.Program.Main(String[] args)
引发此错误的方法: Void Main(System.String[])

```

示例ExceptionInfo展示了Exception类的重要属性及方法。

# .NET Framework基类库中的常用异常类型

异常	说明
ArithmeticException	在算术运算期间发生的异常（如DivideByZeroException和OverflowException）的基类。
DivideByZeroException	在试图用零作除数时引发。
IndexOutOfRangeException	在试图使用小于零或超出数组界限的下标索引数组时引发。
InvalidCastException	从基类型或接口到派生类型的显式转换在运行时失败，引发此异常。
NullReferenceException	尝试使用未创建的对象，引发此异常。
OutOfMemoryException	分配内存（通过new）失败时引发。
OverflowException	在checked上下文中的算术运算溢出时引发。
StackOverflowException	当执行堆栈由于保存了太多挂起的方法调用而耗尽时，就会引发此异常，这通常表明存在非常深或无限的递归。

# 不要写“鸵鸟”式的代码



```
static void Main(string[] args)
{
    try
    {
        //功能代码
    }
    catch (Exception ex)
    {
        //在此处“吃掉”所有异常，
        //不让操作系统弹出“程序错误”对话框
    }
}
```

异常的发生是件好事，它让程序员知道自己的程序可能存在着Bug，并且异常的出现还会通知用户有错误发生，他的数据有可能被破坏，从而让用户有机会考虑补救措施。这远比将所有“错误”包起来不让用户知道要理智得多。



# 自定义异常与异常处理链

---

实际开发中的异常处理

# 创建自己的异常类

自定义异常通常选择直接派生自Exception:

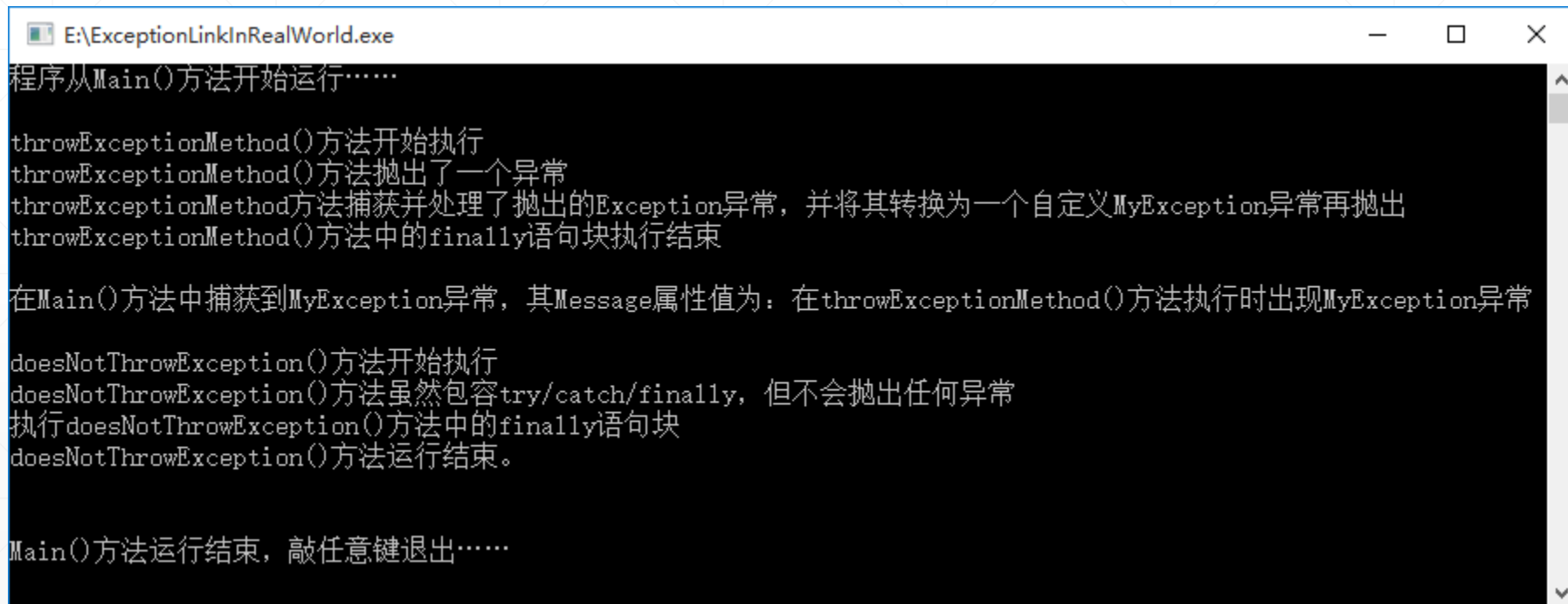
```
Class MyException : Exception
{
    //...
}
```

在合适的地方使用throw语句抛出自定义异常对象:

```
Class MyClass
{
    void someMethod()
    {
        if (条件) throw new MyException();
    }
}
```

# 捕获并转换异常

在实际开发中，经常需要将特定的“过于专业”的异常转换为一个“业务”异常，然后在调用者处进行捕获与处理。



```
E:\ExceptionLinkInRealWorld.exe
程序从Main()方法开始运行……

throwExceptionMethod()方法开始执行
throwExceptionMethod()方法抛出了一个异常
throwExceptionMethod方法捕获并处理了抛出的Exception异常，并将其转换为一个自定义MyException异常再抛出
throwExceptionMethod()方法中的finally语句块执行结束

在Main()方法中捕获到MyException异常，其Message属性值为：在throwExceptionMethod()方法执行时出现MyException异常

doesNotThrowException()方法开始执行
doesNotThrowException()方法虽然包容try/catch/finally，但不会抛出任何异常
执行doesNotThrowException()方法中的finally语句块
doesNotThrowException()方法运行结束。

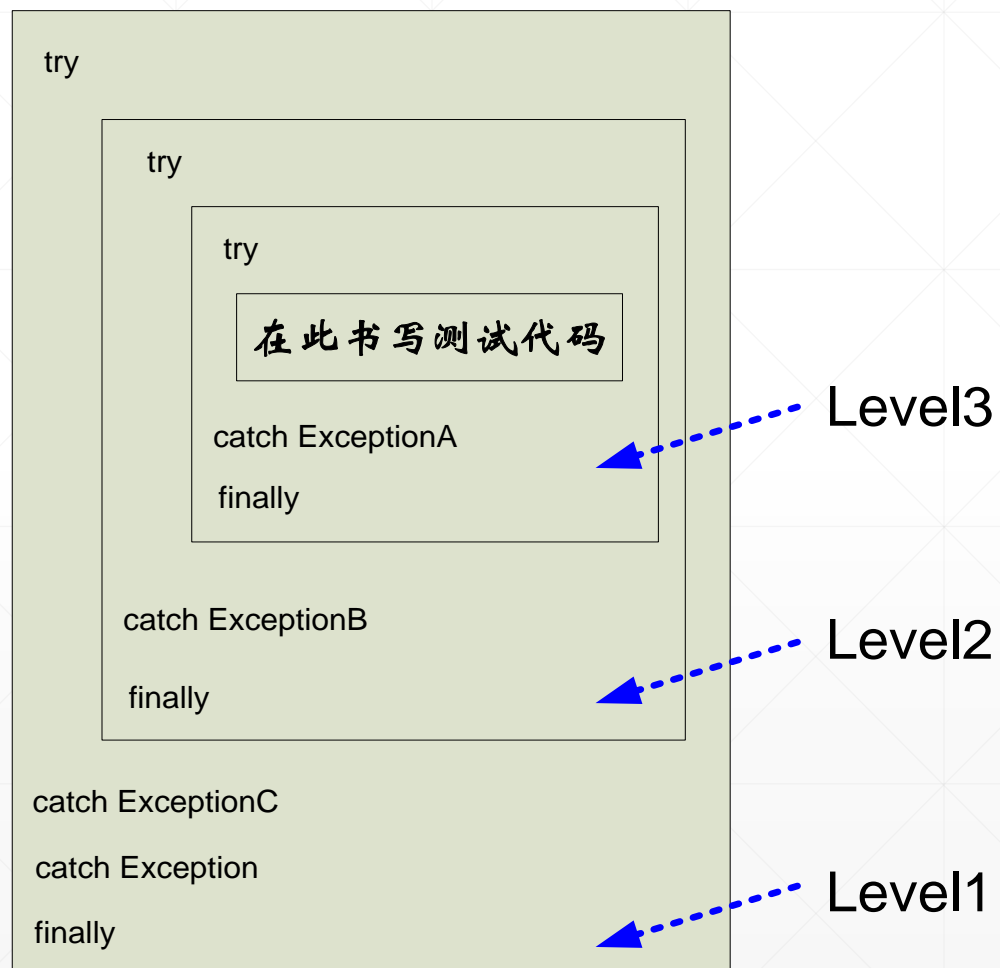
Main()方法运行结束，敲任意键退出……
```

# 异常处理链

在实际开发中，可以参照ExceptionLinkInRealWorld示例的做法，定义一些与业务逻辑相关的自定义异常类，供上层代码进行捕获，从而能更精确地反映系统真实运行情况并及时进行处理。

在实际运行时，最底层的组件捕获一些异常，进行简单处理之后，可以将其转换为自定义的异常类型，再抛出供上层组件处理，而上层组件又可以重复这个工作，再把工作委托给再上层的组件处理……，由此即可构成一个“异常处理链”。

# CLR“两轮遍历”的多层嵌套异常捕获处理流程



“扫描并查找相匹配的catch块”过程，是CLR异常处理流程的第一轮。

“扫描并查找相匹配的finally块”过程，是CLR处理异常流程的第二轮。

示例EmbedException清晰地展示了在应用开发层面CLR异常处理机制所采用的“**两轮遍历**”处理策略

# 实际开发中异常处理与捕获的基本策略

1

预防所有可以预料和防止的错误；

2

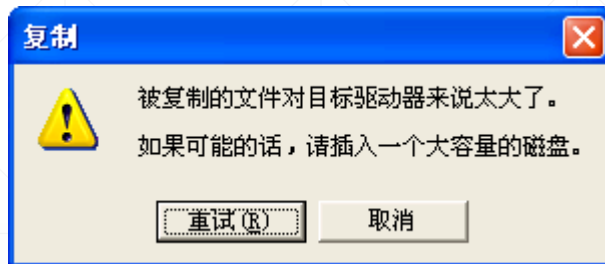
处理所有可以预料但不能防止的错误；

3

捕获所有不能预料的错误；

# 实例：Windows XP中的资源管理器异常处理策略

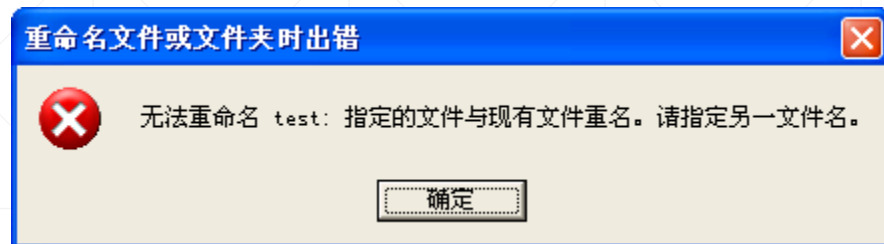
当复制文件时，如果目标驱动器空间不够，提示用户，并允许用户取消操作



这就是第一种情况——**预防所有可以预料和防止的错误**，当检测到这种错误时，程序可直接处理，无需专门引发一个异常。

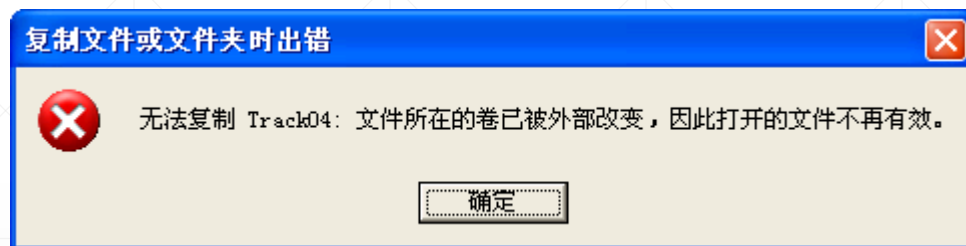


用户在新建文件时不能与同一文件夹下的文件同名，如果用户这样做了，资源管理器给出了这样的提示



这就是第二种情况——**处理所有可以预料但不能防止的错误**。可以提供重试的机会，让用户按照正确的步骤与要求重做操作，在这种情况下也可以不用引发一个异常。

最后一种情况是无法预料的，比如正在向U盘复制文件过程中突然拔出U盘，这时，用户可以看到以下的提示：



这种无法预料的情况最适合于使用异常处理机制。

# 关于开发中异常处理的具体建议-1

引发异常只是为了处理确实异常的情况，而不是为了处理可预知的事件或实现某种程序流程控制。

自身能够处理的异常，不要再向外界抛出。

尽可能地在靠近异常发生的地方捕获并处理异常。

在中间层组件中抛出异常，在界面层组件中捕获异常。

## 关于开发中异常处理的具体建议-2

尽可能地捕获最具体的异常类型，不要在中间层用 `catch(Exception)` “吃掉” 所有异常

在底层组件中捕获CLR抛出的“只有程序员能看懂的”异常，转换为中间层的业务逻辑异常，再由界面层捕获以提供有意义的信息。

在开发阶段捕获并显示所有异常信息，发布阶段要移除部分代码，以避免“过于专业”的异常信息困扰用户，特别地，系统发布之后，不要将服务端异常的详细信息显示给客户端，以免被黑客利用。

# 学以致用

编写一个程序，此程序在运行时要求用户输入一个整数，代表某门课的考试成绩，程序接着给出“不及格”、“及格”、“中”、“良”、“优”的结论。

要求程序必须具备足够的健壮性，不管用户输入什么样的内容，都不会崩溃。