第 4 章　错误及异常处理

本章导读：

程序的编写难免会出现错误，尤其是在开发大型项目中，这些错误是不可避免的。这些错误分为可预知的和不可预知的，程序中的这些错误可能会引发程序抛出异常。在 C#语言中提供了许多预定义的异常类来跟踪程序中产生的问题。

学习目标：

- 理解错误与异常的概念
- 能在程序中捕获异常和处理异常
- 熟悉.NET 开发环境中的代码调试功能

4.1　错误与异常简介

异常用来表示在应用程序执行期间发生的错误，以及其他的意外行为。任何应用程序代码中不可能没有任何问题，可以说，代码中的异常无处不在，通常以下这些情况就有可能引发异常：

- 代码或调用的代码中有错误。
- 操作系统资源不可用。
- 公共语言运行库遇到意外情况。
- 自定义抛出异常。
- 其他。

这些异常有些是可以恢复的，有些是不可以恢复的。代码或调用的代码中的错误引发的异常，开发人员可以通过修改程序更改程序代码消除错误，恢复异常。但公共语言运行库遇到意外情况下引发的异常就是不可恢复的。

【例 4.1】错误与异常示例。

```
namespace _4._1
{
    class Program
    {
        static void Main(string[] args)
        {
            int a;
            a = 6.5;
            int[] array = new int[4];
```

```
            array[5] = 15;
        }
    }
}
```

可以看到，程序调试时会报错，错误原因是：“a = 6.5;”无法将类型 double 隐式转换为 int，存在一个显式转换（是否缺少强制转换?）。这是程序中一个典型的错误，开发人员可以通过修改变量 a 的赋值来更改整个错误。

修改以上错误后，先调试程序，再运行程序。此时，程序可以通过调试，但是程序运行时会弹出一个异常提示，如图 4-1 所示。异常产生的原因是：

```
int[] array = new int[4];
array[5] = 15;
```

代码中设置数组 array 的长度为 5，而“array[5] = 15;”对第 6 个数组元素进行赋值，导致数组下标越界，引发异常。

图 4-1　异常提示

4.2　程序调试技术

Visual Studio 2008 开发环境提供了强大的代码调试功能。在程序开发过程中，常见的错误有两种：语法错误和逻辑错误。

4.2.1　语法错误

在编译过程中出现的错误称为语法错误或编译错误。语法错误是由代码结构中的问题引起的，如拼错关键字，丢掉必要的标点，或者开括弧没有对应的闭括弧等。这些错误通常容易查出，因为编译器会指出它们在哪儿，原因是什么。例如：

```
string 1str='123';
```

这行代码有两个明显的语法错误，一是变量的命名不符合 C#语言的命名规则，二是字符串变量应该用双引号限定起来。通常一个错误会引起很多行编译错误。因此，从最上面的行开始向下调试是很好的习惯。排除了前面出现的错误，可能就改掉了程序中后面出现的重复错误。在使用 Visual Studio 2008 编辑代码时，Visual Studio 2008 会在错误列表中自动提示出现的错误，如图 4-2 所示。

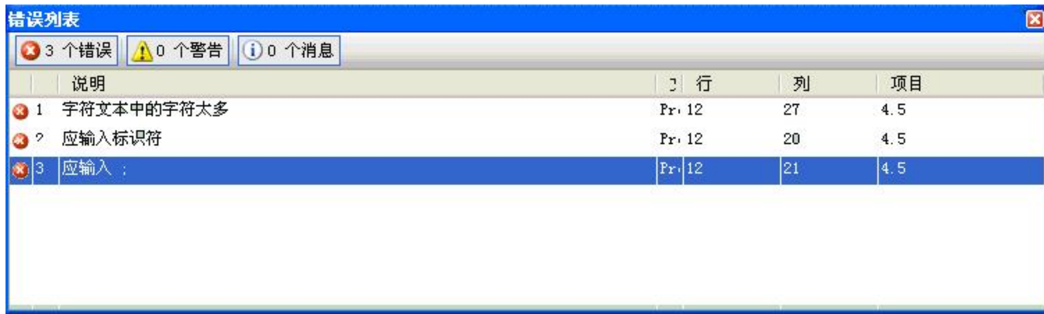

图 4-2 语法错误报告

4.2.2 逻辑错误

逻辑错误是指程序没有按期望的要求执行，它在语法上没有错误。与语法错误相比，逻辑错误更加难以发现和解决。

【例 4.2】纠正逻辑错误势力。

```
namespace _4._2
{

    class Program
    {
        static public void result()
        {
            int sum = 0;
            for (int i = 1; i < 10; i++)
            {
                sum += i;
            }
            Console.WriteLine("从 1 加到 10 的结果为: " + sum);
        }
        static void Main(string[] args)
        {
            result();
        }
    }
}
```

在例 4.2 中，程序没有出现任何的语法错误，但是从输出结果可以看到，程序要求应该循环执行 10 次，然而结果却只循环了 9 次。在这个程序中出现的错误是逻辑错误，它往往很难被发现。针对逻辑错误，Visual Studio 2008 提供了单步执行程序、设置断点等方式来跟踪代码，纠正错误。

（1）单步执行程序。

单步执行是指一次执行一条语句，以便看清每条语句的执行结果，最后找到错误所在。选择“调试”→“逐语句”命令开始单步执行程序，程序首先暂停在主函数的第一行，使用快捷键 F10 或 F11 单步执行程序。F10 与 F11 的区别是：F10 可以跳过一行代码中所调用的方法。

开发人员可以通过监视窗口、局部变量窗口等查看程序运行的细节。在监视窗口中，输入要监视的变量，当这个变量发生改变时，会用红色显示，如图 4-3 所示。在局部变量窗口中，当局部变量的值发生改变时，也会用红色显示，如图 4-4 所示。

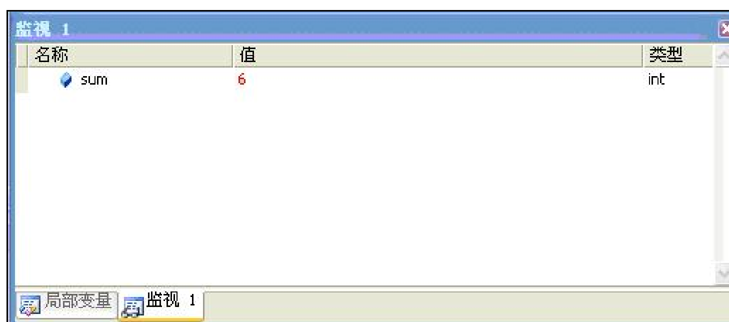

图 4-3　监视窗口

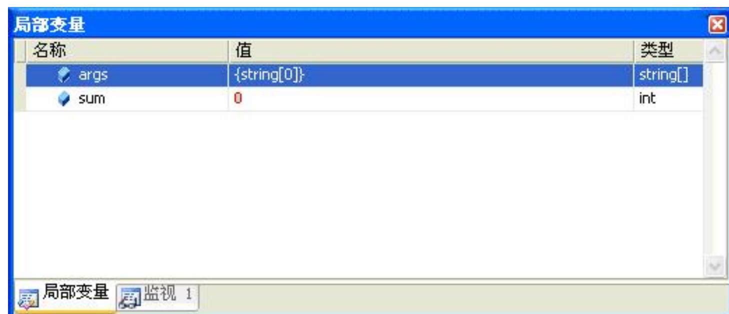

图 4-4　局部变量窗口

（2）设置断点。

单步执行程序对小程序很有用，但对大规模的程序就不可行了。Visual Studio 2008 提供了另外一种方式来解决这个问题，就是使代码暂停在开发人员指定的地方，即设置断点。在设置断点时，首先把光标放置在指定的地方，使用快捷键 F9 或 Ctrl+B，或者选择“调试”→“新建断点”命令，弹出“新建站点”对话框，如图 4-5 所示。

图 4-5 “新建断点”对话框

设置断点后，选中这个断点，右击该断点，选择“条件…”命令，即可出现如图 4-6 所示的“断点条件”对话框。设置断点条件完成后，单击“调试”按钮或使用快捷键 F5 运行程序，当程序运行到指定的断点位置，并满足断点暂停条件时，代码会暂停运行，此时就可以使用单步执行方式来仔细观察程序的运行状态，找出程序的错误点。

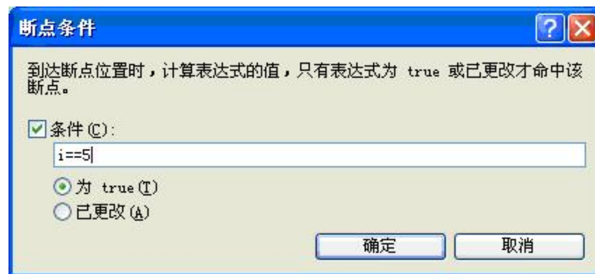

图 4-6 “断点条件”对话框

4.3 捕获异常

C#中可以使用 try-catch 语句来捕获程序抛出的异常。在例 4.1 中可以看到，程序抛出一个 IndexOutOfRangeException 异常。这个异常可以用 try-catch 语句来捕获，其语法规则如下：

```
try
{
处理代码;
}
catch(异常)  //捕获异常
{
捕获异常;
}
```

【例 4.3】演示 try-catch 语句捕获异常。

```
namespace _4._3
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int a;
            a = 15;
            int[] array = new int [4];
           //使用 try-catch 语句捕获异常
            try
            {
                array[5] = 15;  //数组下标越界赋值，引发异常
            }
            catch(IndexOutOfRangeException e)  //捕获异常
            {
               Console.WriteLine("捕获 IndexOutOfRangeException 异常");
            }
        }
    }
}
```

例 4.3 中，首先 try 语句找到产生异常的代码，如果在 try 语句中发生了异常，则执行 catch 语句，本例中 catch 语句捕获程序所抛出的异常，显示捕获的异常；否则将跳出 try-catch 语句块，正常执行程序。程序的运行结果如图 4-7 所示。

图 4-7　例 4.3 的运行结果

例 4.3 中，开发人员知道程序产生的异常是 IndexOutOfRangeException 类型的，所以可以通过代码“catch(IndexOutOfRangeException e)”来准确地捕获异常。如果在开发人员不知道程序产生的异常类型的情况下，可以选择一种更宽泛的异常类型——Exception（异常的基类），如在例 4.3 中代码“catch(IndexOutOfRangeException e)”可由“catch(Exception e)”来代替，程序运行结果一样。而且在开发大型项目时，可能引发的异常不止一种，使用 Exception 可以使所有的异常按照 catch 语句块中的同一种方式处理。

4.4　异常处理

当出现异常后，C#用 try-catch 语句捕获异常，它还提供了对异常处理的内建支持，即通过 try-catch 语句、try-catch-finally 句语和 throw 语句实现结构化、统一化的类型安全的异常处理。

1．try-catch 语句

在 try-catch 语句中，try 语句会捕获到程序出现的异常，然后将程序控制转移到相应的 catch 语句中。如果 try 语句没有遇到任何异常，则程序直接执行 try-catch 语句后面的代码。一个 try-catch 语句中，可以使一个 try 语句后匹配一个或多个 catch 语句，程序根据 try 语句指定的

异常类型与catch语句中指定的异常类型匹配关系，执行相应的catch语句。

2. try-catch-finally语句

在try-catch-finally语句中，无论是否出现异常，是否有catch语句，finally语句一定会执行，其语法规则如下：

```
try
{
处理代码;
}
catch(异常) //捕获异常
{
捕获异常;
}
finally
{
代码块;
}
```

【例4.4】演示try-catch-finally语句处理异常。

```
namespace _4._4
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[4];
            //使用try-catch语句捕获异常
            try
            {
                array[5] = 15;  //数组下标越界赋值，引发异常
            }
            catch (IndexOutOfRangeException e)  //捕获异常
            {
                Console.WriteLine("捕获IndexOutOfRangeException异常");
            }
            finally
            {
                Console.WriteLine("一定执行finally语句块");
            }
        }
    }
}
```

例4.4中，不管try语句、catch语句是否执行，程序一定会执行finally语句。程序的运行结果如图4-8所示。

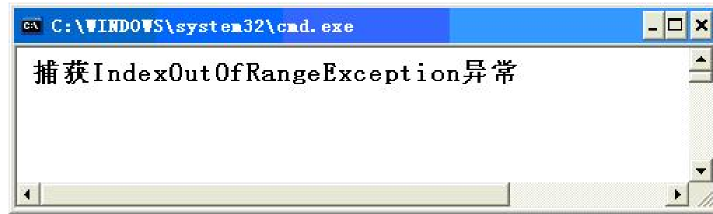

图 4-8 例 4.4 的运行结果

3. throw 语句

throw 语句用于抛出在程序执行期间出现异常情况的信号。throw 语句通常与 try-catch 或者 try-catch-finally 语句配合使用。开发人员可以使用 throw 语句显式抛出异常（即开发人员自定义的异常）。

【例 4.5】应用 throw 抛出异常。

```
namespace _4._5
{
    class Program
    {
        public class EmptyException : Exception//自定义一个异常类，继承于 Exception
        {
            private string error = string.Empty;
            public EmptyException()  //构造函数
            {
            }
            public EmptyException(string message)
                : base(message)
            {
                error = message;
            }
            public EmptyException(string message,Exception inner):base(message, inner)
            {
                error = message;
                inner = null;
            }
        }
        static void Main(string[] args)
        {
            try
            {
                throw new EmptyException("error infomation of use");//抛出异常
            }
            catch (EmptyException ex)
            {
                Console.WriteLine("输出结果为: ");
                Console.WriteLine(ex.Message,ex.InnerException);
            }
        }
    }
}
```

例 4.5 中，自定义一个 EmptyException 异常类继承于 Exception，类中定义了 3 个构造方法，每个构造方法使用不同的参数，通过主方法中的 try-catch 语句块抛出异常。程序的运行结果如图 4-9 所示。

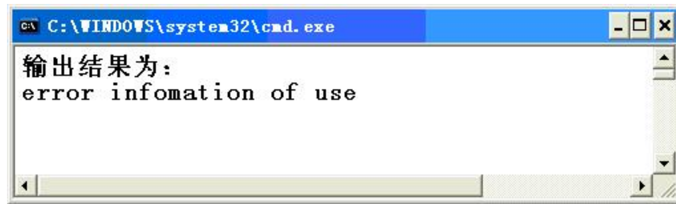

图 4-9　例 4.5 的运行结果

4. 不捕捉异常的后果

捕捉 C#的标准异常是有副作用的：它避免了异常的程序终止。捕捉一个异常时，必须由某些地方的代码片来捕捉。通常，如果程序不捕捉异常，那么将由 C#运行时系统来捕捉异常。此时存在一个问题，运行时系统将报告错误并终止程序的运行。

【例 4.6】不捕捉异常。

```
namespace _4._6
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = new int[4];
            for (int i = 0; i < 10; i++)
            {
                array[i] = i;
                Console.WriteLine("array[{0}] : {1}", i, array[i]);
            }
        }
    }
}
```

例 4.6 中，出现数组下标错误时，会挂起执行过程。程序的运行结果如图 4-10 所示。

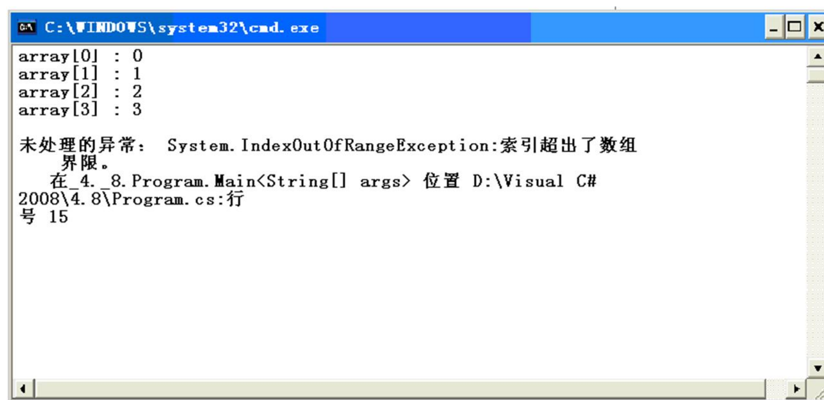

图 4-10　例 4.6 的运行结果

4.5　常用异常

在 C#中，异常用类来表示。所有异常类都必须从内部异常类 Exception 派生而来，而 Exception 是 System 命令空间的一部分。因此，所有异常都是 Exception 的子类。

SystemException 和 ApplicationException 都是从 Exception 派生而来，它们支持 C#定义的两类常规异常：C#运行时系统（即通用语言运行时 CLR）产生的异常，以及应用程序产生的异常。SystemException 和 ApplicationException 都未给 Exception 添加新内容，它们只是定义两个不同异常层次结构的顶层。

C#定义了许多内部异常，它们都是从 System.Exception 派生而来。例如，试图以 0 为除数时，产生的 DivideByZeroException 异常。

System 命令空间定义了一些标准的内部异常。产生运行时错误时，CLR 产生这些异常，所以它们都是从 SystemException 派生而来。C#定义的常用标准异常如表 4-1 所示。

表 4-1　System 命令空间内定义的常用异常

异常	含义
ArrayTypeMismatchException	所存储的数值类型与数组的类型不一致
DivideByZeroException	除数为 0
IndexOutOfRangeException	数组下标越界
InvalidCastException	运行时转换失效
OutOfMemoryException	因为当前内存不足，对 new 的调用失败
OverflowException	发生运行溢出
NullReferenceException	试图对空引用进行操作，也就是说引用没有指向对象
StackOverflowException	栈溢出

【例 4.7】常用异常示例。

```
class A
{
    int a;
    public A(int i)
    {
        a = i;
    }
    public int add(A o)
    {
        return a + o.a;
    }
}
namespace _4._7
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            A d = new A(10);
            A c = null;//c是一个空引用
            int val;

            try
            {
                val=d.add(c);
            }
            catch (NullReferenceException)
            {
                Console.WriteLine("NullReferenceException!");

                c = new A(55);//对C对象重新赋值
                val = d.add(c);
            }
            Console.WriteLine("val={0}", val);
        }
    }
}
```

例 4.7 中，程序定义了一个名为 A 的类，它定义了成员 a 和方法 add()，此方法将 a 的主调对象加入到作为参数传递的对象的 a 中。在 Main()中，创建两个 A 类型的对象：c 和 d。对 d 进行初始化，而未对 c 进行初始化，实际上是显式赋值为 null。然后，将 q 作为变量调用 d.add()。由于 c 未引用任何对象，所以试图获得 c.a 的值时产生 NullReferenceException 异常。程序运行的运行结果如图 4-11 所示。

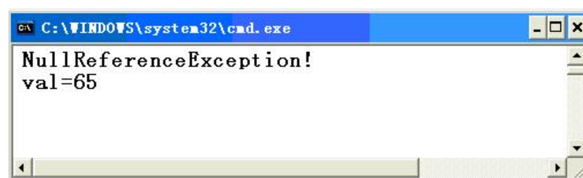

图 4-11　例 4.7 的运行结果