5.8 异常处理

- 1．异常
- 异常表明程序执行期间发生了问题。之所以叫异常，是源于这样一个事实：尽管问题有可能发生，但不频繁。如果说规则是一个语句，通常能正确执行，那么规则的异常就是发生了一个问题。通过异常处理，程序员创建的应用程序能解决(或处理)异常。许多情况下，通过对异常进行处理，程序可继续执行，如同从来没有出现过问题一样。更严重的问题可能阻止程序继续正常执行(而不是要求程序向用户通知问题)，然后采取一种得到控制的方式加以终止。

- 2．异常类
- 当代码出现诸如除数为零、分配空间失败等错误时，就会自动创建异常对象，它们大多是C#异常类的实例。System.Exception类是异常类的基类，一般不直接使用System.Exception，它没有反映具体的异常信息，而一般用它的派生类。System.Exception类的常用属性见表5-7，常用异常类如表5-8所示。

表5-7 System.Exception类的常用属性

属　性	描　述
Message	描述当前异常的消息，即发生异常的原因
InnerException	导致当前异常的Exception类的实例
Source	获取或设置导致错误的应用程序或对象的名称
Data	用户定义的其他异常信息
HelpLink	与当前异常相关的帮助文件的链接
HResult	分配给特定异常的编码数值
TargetSite	获取引发当前异常的方法
StackTrace	获取当前异常发生所经历的方法的名称和签名

表5-8 常用异常类

异 常 类	描 述
System.ArithmeticException	在算术运算期间发生的异常的基类
System.ArrayTypeMismatchException	当存储数组时，如果由于被存储的元素的实际类型与数组的实际类型不兼容而导致存储失败，就会引发该异常
System.DivideByZeroException	当除数为零时，引发该异常
System.IndexOutOfRangeException	如果使用小于零或超出数组界限的下标访问数组，引发该异常
System.InvalidCastException	当从基类型或接口的派生类型的显式转换失败时，发生此异常
System.NullReferenceException	在需要使用引用对象的场合，如果使用Null引用，就会引起此异常
System.OutOfMemoryException	在分配内存(通过new)的尝试失败时引发异常
System.OverflowException	在checked上下文中的算术运算溢出时引发异常
System.StackOverflowException	当堆栈溢出(如无限递归)时引发异常
System.TypeInitializetionException	在静态构造函数引发异常且没有可以捕捉到它的catch子句时，引发此异常

3. 异常处理语句

- (1) try语句
- 将有可能发生异常的代码写在try语句块内，处理try语句中出现的异常代码放在catch语句块。不管try语句块中有没有异常发生，都要执行的语句放在finally语句块中，如一些资源释放、关闭文件等代码。
- 不能单独使用try或catch语句块，它们必须同时使用。通常情况下，try可配合多个catch子句，每个catch子句对应一种特定的异常，就好像switch…case语句一样。当两个catch语句的异常类有派生关系的时候，要将包括派生的异常类的catch语句放到前面，包括基类的异常的catch语句放到后面。

【例5-27】 try-catch-finally语句实例

```
static void Main(string[] args)
    {
          try
          {
              int x ;
              int y ;
              Console.WriteLine("请输入被除数：");
              x = Convert.ToInt32(Console.ReadLine());
              //从键盘输入的数转换成整型，可能有异常
              Console.WriteLine("请输入除数：");
              y = Convert.ToInt32(Console.ReadLine());
              int z = x / y;                          //若除数为0，会产生异常
              Console.WriteLine(z);
           }
```

```
 catch (FormatException)        //格式错误产生的异常执行此子句
        { Console.WriteLine("Error occurred, FormatException"); }
        catch(DivideByZeroException) //除数为0时，执行此子句
        {Console.WriteLine("Error occurred, DivideByZeroException");}
    //其他异常执行此子句，DivideByZeroException类是从Exception派生，所以
            //包含DivideByZeroException异常的catch子句在前面
        catch (Exception)
        {Console.WriteLine("Error occurred, Exception");   }
        finally  //无论是否发生异常，都执行此子句
        {    Console.WriteLine("Thank you for using the program"); }
        Console.ReadLine();
    }
```

(2) throw语句

- 抛出异常有两种方式，一种是在.NET Framework体系类库方法运行出现错误或某些运算不能正常运行时，系统为应用程序自动抛出异常(前面所述的程序产生的异常都是系统自动抛出的)；另一种是在程序中使用throw语句人为地抛出异常。其格式为：
- throw [异常类的对象]；　　//异常类的对象可以省略, 表示再次抛出异常
- throw语句的作用就是人为地产生(抛出)异常，但throw抛出的是异常类的对象，而不是异常类。

【例5-28】人为抛出的异常示例。运行结果如图5.24所示，代码如下：

```
class yich : Exception  //自定义的异常类，从Exception派生而来
  { public string tt="自己定义的异常类";}
  class throwclass
  {
        public static void bornexcept()
        {
            int[] nums1={10,22,58,9,88};
            int[] nums2={3,0,6,0};
            for (int i = 0; i < nums1.Length; i++)
            {
               try
               { Console.WriteLine(nums1[i] + " / " + nums2[i] + "=" +
nums1[i] / nums2[i]); }
```

```
catch (DivideByZeroException)
                { Console.WriteLine("不能被零除"); }
                catch (IndexOutOfRangeException)
                { Console.WriteLine("数组下标出界");
                throw new yich(); //人为抛出刚刚自定义的异常类的对象
                }
            }
        }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            throwclass.bornexcept();     //调用方法
        }
        catch (yich ) //捕获人为抛出的异常类的对象
        {
            yich a=new yich ();
            Console.WriteLine(a.tt);
        }
        Console.ReadLine();
     }
}
```

【例5-29】再次抛出异常示例

```
class throwclass
    {
        public static void bornexcept()
        {
            int[] nums1={10,22,58,9,88};
            int[] nums2={3,0,6,0};
            for (int i = 0; i < nums1.Length; i++)
            {
                try
                { Console.WriteLine(nums1[i]+" / "+nums2[i] + "=" +
nums1[i] / nums2[i]); }
                catch (DivideByZeroException)
                { Console.WriteLine("不能被零除"); }
                catch (IndexOutOfRangeException)
                { Console.WriteLine("数组下标出界");
                throw;  //再次抛出异常
                }
            }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
        throwclass.bornexcept();                    //调用方法
        }
        catch (IndexOutOfRangeException)            //捕获被再次抛出的异常
        {
        Console.WriteLine("出界了");
        }
        Console.ReadLine();
    }
}
```

5.9 集合与索引器

- 5.9.1 集合
- 介绍集合前先回忆一下数组。数组一旦定义并创建，则其大小就固定了，不能伸缩。想要在数组里添加或删除数据，必须创建一个新的数组。通过使用集合，可以把相互联系的数据组合到一个集合内，这样就能够有效地处理这些密切相关的数据。

1. 自定义集合

- 集合是组合在一起的数据组，在System.Collections命名空间提供了一系列的接口，所有的集合类都实现了其中的一些接口。通过实现上述接口，可以设计出自定义的集合类。在System.Collections命名空间提供基本集合功能的接口。
- ● IEnumerable：用于迭代集合中的项。
- ● ICollection：用于获取集合项的个数，并把项复制到一个数组类型中。
- ● IList：提供集合项列表，并访问这些项，以及一些与项列表相关的功能。
- ● IDictionary：与IList类似，但能够通过键码来访问项列表。

- 所有的集合都是基于ICollection 接口、IList接口、IDictionary接口或其相应的泛型接口。IList接口、IDictionary接口都是从ICollection接口派生的，因此ICollection接口是所有集合的基接口。ICollection接口继承了IEnumerable接口，并没有添加任何成员，所以它等价于IEnumerable接口，即所有的集合都必须实现IEnumerable接口。类实现了这个接口后，就能一次列举集合类内所包含的数据元素了。
- IEnumerable接口定义：
- public interface IEnumerable
- { IEnumerator GetEnumerator(); }
- IEnumerable接口只包含一个方法GetEnumerator()，它返回一个能够访问数组的列举器对象。列举器必须实现IEnumerator接口，其定义如下：

```
public interface IEnumerator
{
        object Current
        {  get;  }
        bool MoveNext();
        void Reset();
}
```

- 列举器的功能是读取集合中的数据，即循环访问集合的对象。但是列举器不能修改基础集合，实现IEnumerator接口必须实现方法Reset、MoveNext及属性Current。Reset方法用于初始化枚举；Current属性用于获取当前的项；MoveNext方法用于移至下一项。当创建集合对象后，列举器应该首先定位在集合的第一个元素前，方法Reset将列举器返回到此位置，但此时调用Current属性将发生异常。所以在读取Current前必须调用函数MoveNext，将列举器定位在集合的第一个元素。若用foreach语句，执行顺序是Reset、MoveNext、Current，再MoveNext、Current，一直循环到读完集合的元素为止。

【例5-30】自定义一个集合。该例中要引用System.Collections，运行界面如图5.26所示

```
namespace jiheexamp
{
    //定义集合中的元素MyClass类
    class MyClass
    {
    public string Name;
    public int Age;
    //带参构造函数
    public MyClass(string name, int age)
    {
        this.Name = name;
        this.Age = age;
    }
}
```

```
public class Iterator:IEnumerator,IEnumerable  //从两个接口继承而来的自定义的类
{     //初始化MyClass 类型的集合
      private MyClass[] ClassArray;
      int Cnt;
      public Iterator()
      {          //使用带参构造函数
                  ClassArray = new MyClass[4];
                  ClassArray[0] = new MyClass("Kith",23);
                  ClassArray[1] = new MyClass("Smith",30);
                  ClassArray[2] = new MyClass("Geo",19);
                  ClassArray[3] = new MyClass("Greg",14);
                  Cnt = -1;
               }
```

```
//实现IEnumerator的Reset()方法
    public void Reset()
    {     //指向第一个元素之前，Cnt为-1，遍历从0开始
          Cnt = -1;
    }
  //实现IEnumerator的MoveNext()方法
  public bool MoveNext()
  {
    return (++ Cnt < ClassArray.Length);
    //Cnt先自加1，如果小于数组长度，则返回真
  }
  //实现IEnumerator的Current属性
  public object Current
  {   get
      {      return ClassArray[Cnt];              }
  }
  //实现IEnumerable的GetEnumerator()方法
  public IEnumerator GetEnumerator()
  { return (IEnumerator)this; }
```

```
static void Main()
        {
            Iterator It = new Iterator();
            //像数组一样遍历集合
            foreach (MyClass MY in It)
            {
                Console.WriteLine("Name : " + MY.Name.ToString());
                Console.WriteLine("Age : " + MY.Age.ToString());
            }
        }
}
```

2. 使用集合类

- 另一种使用集合的方法是使用系统已经定义的集合类。下面只简单介绍Stack类和Queue类集合。
- (1) Stack类
- Stack类表示对象的后进先出集合。Stack类常用的方法有：
- ● Clear：从Stack中移除所有对象；
- ● Pop：移除并返回位于Stack顶部的对象；
- ● Push：将对象插入Stack的顶部；
- ● Peek：返回位于Stack顶部的对象，但不将其移除。

- Stack类将它的对象存储在数组。只要数组足够大，大到可以存储新的对象，调用Push方法就是有效的。但如果内部数组不是足够大，需要调整大小，就必须分配新数组并把现有的对象复制到新数组中。为避免这一昂贵的操作成本，可以采用预选分配一个大的内部数组，或定义满足需要的合适的增长系数。

【例5-31】 Stack集合示例

```
static void Main(string[] args)
{
        Stack s = new Stack();
        s.Push("先");
        s.Push("进");
        s.Push("后");
        s.Push("出");
        s.Push("堆栈");
        Console.WriteLine("堆栈的内容为(从栈顶到栈底)：");
        foreach (string str in s)
         { Console.Write(str       );} //读集合中的每个元素
        Console.Write("\n");      Console.WriteLine("出栈的顺序为：");
        while( s.Count > 0)
        {  string str  = (string)s.Pop();
           Console.WriteLine("Poping {0}",str);//弹出元素，后进先出
        }
        Console.WriteLine("结束");
        Console.ReadLine();
}
```

(2) Queue类

- Queue类是按照先进先出的顺序处理的集合类，此类将队列作为循环数组实现。其主要有以下三种操作：
- ● Enqueue：将一个元素添加到Queue尾。
- ● Dequeue：从Queue的开始处移除旧的元素。
- ● Peek：返回Queue开始处的对象，但不将其从Queue中移除。

5.9.2 索引器

- 索引器允许对象像数组一样进行索引，通过对对象元素的下标的索引，可以访问指定的对象。索引器类似于属性，也使用get关键字和set关键字来定义对被索引元素的读/写权限；它们之间不同的是索引器有索引参数。索引器又称为带参数的属性。除表5-9所示的差别外，为属性访问器定义的所有规则同样适用于索引器的访问器

表5-9 索引器与属性的差别

属　性	索 引 器
允许像调用公共数据成员一样调用方法	允许对对象本身使用数组表示法来访问该对象内部集合中的元素
可以通过简单的名称进行访问	可通过索引器进行访问
可以为静态成员或实例成员	必须为实例成员
属性的get访问器没有参数	索引器的get访问器具有与索引器相同的形参表
属性的set访问器包含隐式value参数	除值参数外，索引器的set访问器还具有与索引器相同的形参表

- 索引器的声明与属性的声明基本相同，不同之处在于：索引器的名称固定为关键字this，且必须指定索引的参数表。声明索引器的基本形式：
- [修饰符]类型 this [参数表]
- {
- [get{get访问器}]
- [set{set访问器}]
- }[;]

【例5-32】索引器示例

```
using System;
class MyClass
{
        private string [] data = new string[5];
        //索引器定义，根据下标访问data
        public string this [int index]
        {       get
                {       return data[index];             }
                set
                {       data[index] = value; }
        }
}
```

```
class MyClient
{
        public static void Main()
        {
            MyClass mc = new MyClass();
            //调用索引器set赋值
            mc[0] = "Rajesh";
            mc[1] = "A3-126";
            mc[2] = "Snehadara";
            mc[3] = "Irla";
            mc[4] = "Mumbai";
            //调用索引器get读出
            Console.WriteLine("{0},{1},{2},{3},{4}",
            mc[0],mc[1],mc[2],mc[3],mc[4]);
        }
}
```