第 5 章　委托、Lambda 表达式与 LINQ 技术

在实际开发过程中，处理和操作数据是一项重要的任务。经常需要进行查询、筛选、排序、转换，以及把方法作为参数传递等操作来满足特定的需求。为了更高效地处理这些操作需求，委托、Lambda 表达式、LINQ 就是常用的技术。本章就来学习这些技术的基础知识与使用方法。

学习目标

（1）理解委托的含义。
（2）能利用委托解决实际问题。
（3）理解匿名方法。
（4）能在实际应用中运用 Lambda 表达式。
（5）理解 LINQ 的含义。
（6）能根据需求编写各类 LINQ 语句。

思政目标及设计建议

根据新时代软件工程师应该具备的基本素养，挖掘课程思政元素，有机融入教学中，本章思政目标及设计建议如表 5-1 所示。

表 5-1　第 5 章思政目标及设计建议

思 政 目 标	思政元素及融入
培养学生学以致用的能力	通过例子分析与实际操作培养学以致用的能力
培养自主探索、敬业、专注的工匠精神	通过课前自主学习，培养自主探索、敬业、专注的工匠精神

5.1　委托的基本认识

前面的章节中定义过不少方法，大多数方法会定义一些形参，用于接收实际要传递的参数，但是能不能把方法作为参数进行传递呢？也就是定义形参时有没有一种数据类型的变量可以接收方法？

答案是有的，这就是委托。也就是说，委托是一种数据类型，这种数据类型是用来接收方法的。

官方给出的委托定义：委托是一种引用类型，表示对具有特定参数列表和返回类型的方法的引用。

通俗地理解，委托相当于对具有相同返回类型和参数列表这一类方法进行了封装。

由于委托本质上也是一个派生自 Delegate 类的类，因此类可以声明在哪里，委托就可以声明在哪里。

例 1：委托的基本认识——定义一个能接收无参数、无返回值方法的委托。

打开 Visual Studio 2002，创建一个控制台应用(.NET Framework)程序，解决方案名称为 Delegate_Lambda_Linq，项目名称为 DelegateDemo1，如图 5-1 所示。

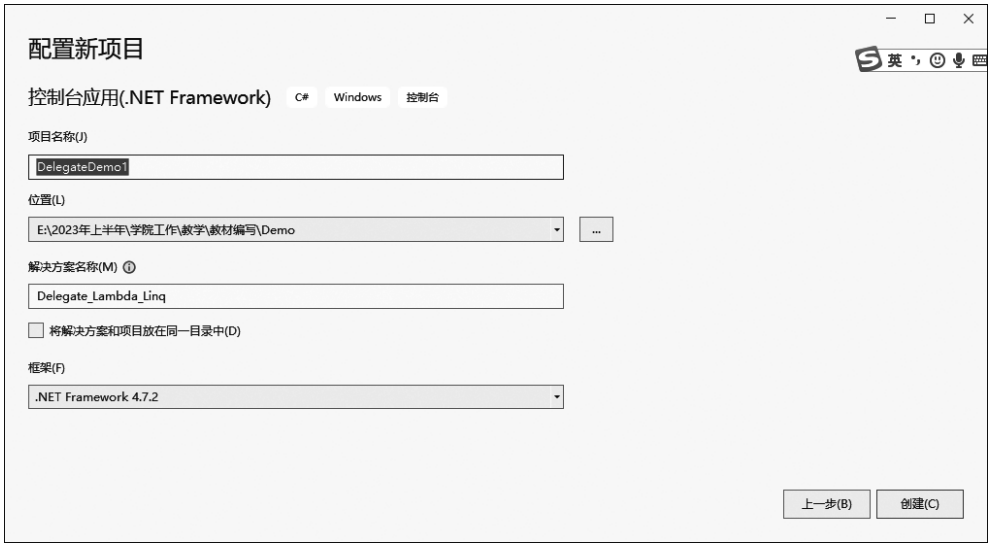

图 5-1　创建解决方案及项目

接下来定义委托，可是委托定义的位置在哪里呢？因为委托本质也是类，因此要定义在与类并列的位置。完整代码及注释如下。

```
namespace DelegateDemo1
{
    //(1)定义委托。即定义一个委托类型(委托是一种数据类型,能接收方法的数据类型),
    //用来保存无参数、无返回值的方法
    //委托要定义在命名空间中,和类是同一个级别
    public delegate void MyDelegate();//像定义抽象方法一样,没有实现(方法体)
    internal class Program
    {
        static void Main(string[] args)
        {
            //(3)使用委托:声明委托变量,并赋值
            //声明了一个委托变量 md,新建了一个委托对象,并且把方法 M1 传递了进去
            //即 md 委托保存了 M1 方法
            //MyDelegate md =new MyDelegate(M1);//第 1 种写法
            MyDelegate md =M1;////第 2 种写法

            //(4)执行委托:调用 md 委托时就相当于调用 M1 方法
```

```
            md();//与下面等同
            //md.Invoke();//Invoke()的作用是执行指定的委托
            Console.WriteLine("ok");
            Console.ReadKey();
        }
        //(2)定义一个方法,由于该方法要传递给 MyDelegate 委托,所以只能是无参数无返回
        //值的方法
        static void M1()
        {
            Console.WriteLine("我是一个没有参数没有返回值的方法");
        }
    }
}
```

以上代码首先定义了一个委托类型 MyDelegate(),它是一个无参数无返回值的类型,因此也只能接收无参数无返回值的方法。然后在 Program 类中定义了一个方法,这个方法是要传递给委托类型的,所以只能是无参数无返回值的方法。接着是使用委托,定义一个委托类型变量,然后可以采用代码中的第 1 种写法或第 2 种写法,第 2 种写法更简洁。最后执行委托,也有两种写法,第 1 种委托变量名(),第 2 种委托变量名.Invoke()。上面的代码执行结果如图 5-2 所示。

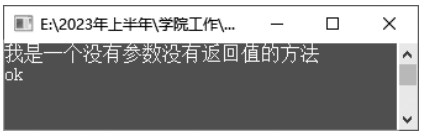

图 5-2　例 1 执行结果

提示：以上代码要注意书写位置,委托定义在哪里？方法定义在哪里？在哪里使用委托？

5.2　委托的基本应用举例

例 2：假设一件事情在前面和后面要做的事情比较固定(这里假设输出“=======”),但是中间要做的事情经常发生变化(有可能是①要输出系统当前时间到控制台;②要输出系统当前是星期几;③要把系统时间写入文本文件等)。

实现思路：定义一个方法,前后固定的动作用代码写好,中间的动作是不固定的,可以用委托传递不同方法做不同的事情,因此定义的方法形参要是委托类型参数。具体实现步骤和代码如下。

(1) 在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用(.NET Framework)新项目,名称为 DelegateDemo2。在 DelegateDemo2 项目下添加一个类 TestClass,该类中编写的代码如下。

```
//定义一个委托
public delegate void middleDelegate();
internal class TestClass
{
```

```
    public void DoSomething(middleDelegate middleThing) //委托类型作为参数,即调用
                                                         //此方法要传递一个方法进来
    {
        Console.WriteLine("==========================");
        Console.WriteLine("==========================");
        if (middleThing !=null)   //委托是一个对象,就有可能为 null,所以先判断下是否
                                  //为 null
        {
            middleThing();        //执行委托,根据传递的方法,执行得到不同效果
        }
        Console.WriteLine("==========================");
        Console.WriteLine("==========================");
    }
}
```

实例在 TestClass 类中定义了一个委托类型,然后定义了一个方法,该方法用定义的委托类型作为形参。

(2) 在 Program.cs 类中编写代码。首先根据需求定义三个不同的静态方法,其次在 Main()方法中编写测试代码,完整代码如下。

```
internal class Program
    {
        static void Main(string[] args)
        {
            TestClass tc =new TestClass();
            //传递不同方法做不同的事情
            tc.DoSomething(WriteTimeToFile);
            Console.WriteLine("OK");
            Console.ReadKey();
        }
        //把系统当前时间输出到控制台
        public static void PrintTimeToConsole()
        {
            Console.WriteLine(System.DateTime.Now.ToString());
        }
        //把系统当前时间输出到文件 time.txt 中。time.txt 文件位置是指当前项目的可执
        //行文件所在位置,即当前项目的 bin\Debug 目录
        public static void WriteTimeToFile()
        {
            //需要导入 System.IO 命名空间
            File.WriteAllText("time.txt", System.DateTime.Now.ToString());
        }
        //把系统当前星期几输出到控制台
        public static void PrintWeekToConsole()
        {
            Console.WriteLine(System.DateTime.Now.DayOfWeek.ToString());
        }
    }
```

测试运行效果,可以看到向 DoSomething()方法传递不同方法作为实参,可以实现执行

不同的方法，产生不同效果。

提示：启动运行前，需要设置解决方案的启动项为当前选定内容。

例 3：对字符串的处理经常要发生变化，例如，在字符串两端加“=”、加“★”，把字符串字母全部转换为大写等。

实现思路：定义一个方法，该方法需要一个参数用于接收要处理的字符串，第二个参数就是对字符串的处理方式，由于对字符串的处理方式是不固定的，所以第二个参数可定义为委托类型。由于这里需要接收要处理的字符串，最后还要把处理后的字符串输出，所以定义的委托有一个 string 类型的形参，返回值为 string。

具体实现步骤和代码如下。

(1) 在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用(.NET Framework)新项目，名称为 DelegateDemo3。在 DelegateDemo3 项目下添加一个类 TestClass，该类中编写的代码如下。

```
namespace DelegateDemo3
{
    //定义一个委托(委托是一种数据类型，接收方法的数据类型)
    public delegate string GetStringDelegate(string str);
    internal class TestClass
    {
        public void ChangeStrings(string[] strs, GetStringDelegate GetString)
        {
            for (int i =0; i <strs.Length; i++)
            {
                strs[i] =GetString(strs[i]);//由于对字符串的需求有很多种，所以把对
                                            //字符串变化部分用委托封装成一个方法
            }
        }
    }
}
```

(2) 在 Program.cs 类中编写代码。首先根据需求定义三个不同的静态方法，其次在 Main()方法中编写测试代码，完整代码如下。

```
internal class Program
    {
        static void Main(string[] args)
        {
            TestClass tc =new TestClass();
            //要处理的字符串数组
            string[] names =new string[] { "ZhangSan", "LiSi", "WangWu", "LaoLiu" };
            //第 1 个参数为要处理的字符串数组，第 2 个参数为处理的方式(根据需求可以传递
            //定义好的三个方法之一，这里传递两端加★的处理方式)
            tc.ChangeStrings(names,ChangeStrings2);
            //把变化后的字符串数组中的字符串输出
            for (int i =0; i <names.Length; i++)
            {
```

```
                Console.WriteLine(names[i]);
            }
            Console.ReadKey();
        }

        static string ChangeStrings1(string strs)//需求 1:在字符串两端加=
        {
            return "=" +strs +"=";
        }
        static string ChangeStrings2(string strs)//需求 2:在字符串两端加★
        {
            return "★" +strs +"★";
        }
        static string ChangeStrings3(string str)//需求 3:把字符串中的字母转换为大写
        {
            return str.ToUpper();
        }
    }
```

测试运行效果如图 5-3 所示。

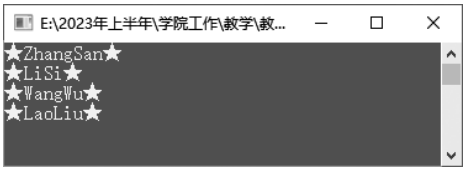

图 5-3　例 3 运行效果

说明：如果上面加粗的代码改为 ChangeStrings1，即调用 ChangeStrings1()方法，那么输出的每个字符串是在两端加“=”。如果改为 ChangeStrings3，即调用 ChangeStrings3()方法，那么输出的每个字符串中字母都将改为大写字母。

委托要点总结如下。

委托是一种数据类型，像类一样的一种数据类型。一般都是直接在命名空间中定义。

定义委托时，需要指明返回值类型、委托名与参数列表，这样就能确定该类型的委托能存储(接收)什么样的方法。定义委托后一般需要定义处理需求的方法，该方法需要用定义的委托作为形参。

使用委托前需要根据不同的需求定义好不同的方法。

5.3　内置委托

ASP.NET 内置了三种委托，所以实际开发中一般不需要自己去定义委托，直接用系统内部提供的就可以。

(1) Action——Action 委托的非泛型版本就是一个无参数无返回值的委托。

(2) Action<T>——Action<T>委托的泛型版本是一个无返回值，但是参数个数及类型可以改变的委托。

（3）Func＜T＞——Func＜T＞委托只有泛型版本的，接收的参数个数可以是若干个，也可以是没有，但是一定有返回值的方法。

1. Action 委托（非泛型版本）

例 4：对例 1 改进。

在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用（.NET Framework）新项目，名称为 DelegateDemo4。在 Program.cs 类中只需要编写如下代码。

```
internal class Program
    {
        static void Main(string[] args)
        {
            //Action 是内置委托，直接使用
            Action md =M1;
            md();
            Console.ReadKey();
        }
        static void M1()
        {
            Console.WriteLine("我是一个没有参数没有返回值的方法");
        }
    }
```

运行效果与例 1 一样。可以看到这里没有自己定义委托，而是直接使用内置委托 Action。

2. Action＜T＞委托（泛型版本）

如何使委托能接收参数个数及类型都不固定的方法呢？——使用泛型委托。

例 5：自定义泛型委托。

假设方法的参数可以是 string、int、bool 三种数据类型，但方法均无返回值。如果不使用泛型委托，那么就需要针对三种参数类型，定义三个不同的委托，分别用于接收三种类型参数的方法，实例代码如下。

```
public delegate void Mydelegate1(string msg);
public delegate void Mydelegate2(int i);
public delegate void Mydelegate3(bool b);
```

那么如果参数类型有更多种呢？这种定义显然很麻烦，怎么办呢？

这个时候就可以定义泛型委托。如何定义及使用呢？下面通过例子来演示。

在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用（.NET Framework）新项目，名称为 DelegateDemo5。在 Program.cs 类中编写如下代码。

```
namespace DelegateDemo5
{
    public delegate void MyGenericdelegate<T>(T args);//这个委托就可以接收 1 个参
//数、无返回值的方法，但是这个参数数据类型可以任意，这里一般用 T 表示——这就是自定义的
//泛型委托
    internal class Program
    {
```

```
        static void Main(string[] args)
        {
            MyGenericdelegate<string>  md1 =M1;
            md1("一个参数");
            MyGenericdelegate<int>  md2 =M1;
            md2(1);
            MyGenericdelegate<bool>  md3 =M1;
            md3(true);
            Console.ReadKey();
        }
        //三个参数类型不同的方法可以定义为重载方法
        static void M1(string msg)
        {
            Console.WriteLine(msg);
        }
        static void M1(int i)
        {
            Console.WriteLine(i);
        }
        static void M1(bool b)
        {
            Console.WriteLine(b);
        }
    }
}
```

上面的加粗代码就是自定义的泛型委托，该泛型委托参数只能有 1 个，无返回值，但是参数类型不固定，一般用 T 表示，而且委托名称后面要加<T>。

对于这种自定义的泛型委托可以使用内置的泛型委托 Action<T>。也就是说，上面的加粗代码可以注释或删除，然后把 MyGenericdelegate 换成 Action 即可。除此之外，Action<T>泛型委托参数不仅类型不固定，个数也可以不固定，也就是说，Action<T>泛型版本是一个无返回值，但是参数个数及类型可以改变的委托。

例 6：Action<T>泛型委托应用演示。

在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用(.NET Framework)新项目，名称为 DelegateDemo6。在 Program.cs 类中编写如下代码。

```
static void Main(string[] args)
    {
        Action<string,int>  action1 =M1;
        action1("内置无返回值的泛型委托应用", 2);//两个参数
        Action<int>  action2 =M1;
        action2(1);
        Action<bool>  action3 =M1;
        action3(true);
        Console.ReadKey();
    }
    static void M1(string msg,int i)
    {
```

```
            Console.WriteLine(msg+i);
        }
        static void M1(int msg)
        {
            Console.WriteLine(msg);
        }
        static void M1(bool b)
        {
            Console.WriteLine(b);
        }
```

内置委托 Action 和 Action＜T＞都不支持带返回值的方法，那么有没有内置的带返回值的委托呢？有，这就是 Func＜T＞，该委托只有泛型版本的，接收的参数个数可以是若干个，也可以是没有，但是一定有返回值的方法。常见形式如下。

Func＜TResult＞表示没有参数，只有返回值。

Func＜T，TResult＞表示有一个参数，有返回值。

Func＜T1，T2，TResult＞表示有两个参数（前两个参数 T1，T2 表示参数类型，最后的 TResult 表示返回值类型），有返回值。

Func＜T1，T2，T3，TResult＞表示有三个参数（前三个参数 T1，T2，T3 表示参数类型，最后的 TResult 表示返回值类型），有返回值。

总之，Func 委托最后一个 TResult 表示返回值类型，前面的不管多少个 T 都是表示参数类型。

例 7：Func＜T＞泛型委托应用演示。

在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用（.NET Framework）新项目，名称为 DelegateDemo7。在 Program.cs 类中编写如下代码。

```
static void Main(string[] args)
        {
            #region 无参数有返回值的 Func 委托
            Func<int> fun1 =M1;
            int n1 =fun1();
            Console.WriteLine(n1);
            #endregion
            #region 有三个参数有返回值的 Func 委托
            Func<int, int, int, int>  fun2 =M1;
            int n2 =fun2(1, 2, 3);
            Console.WriteLine(n2);
            #endregion
            #region 有两个参数有返回值的 Func 委托
            Func<string, int, string>  fun3 =M1;
            string str =fun3("Func 委托应用", 2);
            Console.WriteLine(str);
            #endregion
            Console.ReadKey();
        }
        static int M1()
```

```
        {
            return 1;
        }
        static int M1(int n1, int n2, int n3)
        {
            return n1 +n2 +n3;
        }
        static string M1(string msg, int i)
        {
            return msg+i;
        }
```

运行效果如图 5-4 所示。

图 5-4　例 7 运行效果

5.4　多播委托

多播委托就是一个委托同时绑定多个方法，多播委托也叫委托链、委托组合。绑定方法时也就是为多播委托变量多次赋值。除第一次赋值直接用“＝”外，后面的赋值都可以用“＋＝”，如果是解绑就用“－＝”。

1. 绑定无返回值的多个委托

例 8：绑定无返回值的多个委托。

在解决方案 Delegate_Lambda_Linq 下添加一个控制台应用(.NET Framework)新项目，名称为 DelegateDemo8。在 Program.cs 类中编写如下代码，含义参见注释。

```
    static void Main(string[] args)
        {
            #region 绑定无返回值的多个委托
            Action<string> action =M1;//这句话只绑定一个 M1 方法(绑定第一个方法不
                                      //能用+=复制,因为开始 action 为 null,所以只
                                      //能用=赋值),下面再给 action 绑定方法
            action +=M2;              //后面再为 action 绑定方法时都可以用+=
            action +=M3;
            action +=M4;
            action +=M5;
            action -=M3;              //解除绑定 M3 方法(即用-=赋值为解除绑定方法)
            action("多播委托");
            #endregion
            Console.ReadKey();
        }
        static void M1(string msg)
        {
```