第 21 章　使用 LINQ 查询

了解了基本的 LINQ 基本概念，以及 Lambda 表达式基础后，就能够使用 LINQ 进行应用程序开发。LINQ 使用了 Lambda 表达式，以及底层接口实现了对集合的访问和查询，开发人员能够使用 LINQ 对不同的对象，包括数据库、数据集和 XML 文档进行查询。

21.1　LINQ 查询概述

LINQ 可以对多种数据源和对象进行查询，如数据库、数据集、XML 文档甚至是数组，这在传统的查询语句中是很难实现的。如果有一个集合类型的值需要进行查询，则必须使用 Where 等方法进行遍历，而使用 LINQ 可以仿真 SQL 语句的形式进行查询，极大的降低了难度。

21.1.1　准备数据源

既然 LINQ 可以查询多种数据源和对象，这些对象可能是数组，可能是数据集，也可能是数据库，那么在使用 LINQ 进行数据查询时首先需要准备数据源。

1．数组

数组中的数据可以被 LINQ 查询语句查询，这样就省去了复杂的数组遍历。数组数据源示例代码如下所示。

```
string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" };
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

数组可以看成是一个集合，虽然数组没有集合的一些特性，但是从另一个角度上来说可以看成是一个集合。在传统的开发过程中，如果要筛选其中包含“学习”字段的某个字符串，则需要遍历整个数组 。

2．SQL Server

在数据库操作中，同样可以使用 LINQ 进行数据库查询。LINQ 以其优雅的语法和面向对象的思想能够方便的进行数据库操作，为了使用 LINQ 进行 SQL Server 数据库查询，可以创建两个表，这两个表的结构如下所示。Student（学生表）：

- S_ID：学生 ID。
- S_NAME：学生姓名。
- S_CLASS：学生班级。
- C_ID：所在班级的 ID。

上述结构描述了一个学生表，可以使用 SQL 语句创建学生表，示例代码如下所示。

```
USE [student]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

```
        CREATE TABLE [dbo].[Student](
        [S_ID] [int] IDENTITY(1,1) NOT NULL,
        [S_NAME] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
        [S_CLASS] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
        [C_ID] [int] NULL,
  CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED
  (
  [S_ID] ASC
 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
 ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
 ) ON [PRIMARY]
```

为了更加详细的描述一个学生所有的基本信息，就需要创建另一个表对该学生所在的班级进行描述，班级表结构如下所示。Class（班级表）：

- C_ID：班级 ID。
- C_GREAD：班级所在的年级。
- C_INFOR：班级专业。

上述代码描述了一个班级的基本信息，同样可以使用 SQL 语句创建班级表，示例代码如下所示。

```
USE [student]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Class](
        [C_ID] [int] IDENTITY(1,1) NOT NULL,
        [C_GREAD] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
        [C_INFOR] [nvarchar](50) COLLATE Chinese_PRC_CI_AS NULL,
  CONSTRAINT [PK_Class] PRIMARY KEY CLUSTERED
  (
        [C_ID] ASC
 )WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
 ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
 ) ON [PRIMARY]
```

上述代码在 Student 数据库中创建了一个班级表，开发人员能够向数据库中添加相应的信息以准备数据源。

3．数据集

LINQ 能够通过查询数据集进行数据的访问和整合；通过访问数据集，LINQ 能够返回一个集合变量；通过遍历集合变量可以进行其中数据的访问和筛选。在第 9 章中讲到了数据集的概念，开发人员能够将数据库中的内容填充到数据集中，也可以自行创建数据集。

数据集是一个存在于内存的对象，该对象能够模拟数据库的一些基本功能，可以模拟小型的数据库系统，开发人员能够使用数据集对象在内存中创建表，以及模拟表与表之间的关系。在数据集的数据检索过程中，往往需要大量的 if、else 等判断才能检索相应的数据。

使用 LINQ 进行数据集中数据的整理和检索可以减少代码量并优化检索操作。数据集可以是开发人员自己创建的数据集也可以是现有数据库填充的数据集，这里使用上述 SQL Server 创建的数据库中的数据进行数据集的填充。

21.1.2 使用 LINQ

在传统对象查询中，往往需要很多的 if、else 语句进行数组或对象的遍历，例如在数组中寻找相应的字段，实现起来往往比较复杂，而使用 LINQ 就简化了对象的查询。由于前面已经准备好了数据源，那么就能够分别使用 LINQ 语句进行数据源查询。

1．数组

在前面的章节中，已经创建了一个数组作为数据源，数组示例代码如下所示。

```
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

上述代码是一个数组数据源，如果开发人员需要从其中的元素中搜索大于 5 的数字，传统的方法应该遍历整个数组并判断该数字是否大于 5，如果大于 5 则输出，否则不输出，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;                                                    //使用必要的命名空间
using System.Text;
namespace _21_1
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" };    //定义数组
            int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            for (int i = 0; i < inter.Length; i++)                    //遍历数组
            {
                if (inter[i] > 5)                                     //判断数组元素的值是否大于 5
                {
                    Console.WriteLine(inter[i].ToString());           //输出对象
                }
            }
            Console.ReadKey();
        }
    }
}
```

上述代码非常简单，将数组从头开始遍历，遍历中将数组中的的值与 5 相比较，如果大于 5 就会输出该值，如果小于 5 就不会输出该值。虽然上述代码实现了功能的要求，但是这样编写的代码繁冗复杂，也不具有扩展性。如果使用 LINQ 查询语句进行查询就非常简单，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        string[] str = { "学习", "学习 LINQ", "好好学习", "生活很美好" };    //定义数组
        int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };                  //定义数组
        var st = from s in inter where s > 5    select s;             //执行 LINQ 查询语句
        foreach (var t in st)                                         //遍历集合元素
        {
            Console.WriteLine(t.ToString());                          //输出数组
        }
```

```
            Console.ReadKey();
        }
    }
```

使用 LINQ 进行查询之后会返回一个 IEnumerable 的集合。在上一章讲过，IEnumerable 是.NET 框架中最基本的集合访问器，可以使用 foreach 语句遍历集合元素。使用 LINQ 查询数组更加容易被阅读，LINQ 查询语句的结构和 SQL 语法十分类似，LINQ 不仅能够查询数组，还可以通过.NET 提供的编程语言进行筛选。例如 str 数组变量，如果要查询其中包含“学习”的字符串，对于传统的编程方法是非常冗余和繁琐的。由于 LINQ 是.NET 编程语言中的一部分，开发人员就能通过编程语言进行筛选，LINQ 查询语句示例代码如下所示。

```
            var st = from s in str where s.Contains("学习")   select s;
```

2．使用 SQL Server

在传统的数据库开发中，如果需要筛选某个数据库中的数据，可以通过 SQL 语句进行筛选。在 ADO.NET 中，首先需要从数据库中查询数据，查询后就必须将数据填充到数据集中，然后在数据集中进行数据遍历，示例代码如下所示。

```
            try
            {
                SqlConnection
                con = new SqlConnection("server='(local)';database='student';uid='sa';pwd='sa'"); //创建连接
                con.Open();                                                                       //打开连接
                string strsql = "select * from student,class where student.c_id=class.c_id";      //SQL 语句
                SqlDataAdapter da = new SqlDataAdapter(strsql, con);                              //创建适配器
                DataSet ds = new DataSet();                                                       //创建数据集
                int j = da.Fill(ds, "mytable");                                                   //填充数据集
                for (int i = 0; i < j; i++)                                                       //遍历集合
                {
                    Console.WriteLine(ds.Tables["mytable"].Rows[i]["S_NAME"].ToString());        //输出对象
                }
            }
            catch
            {
                Console.WriteLine("数据库连接错误");                                              //抛出异常
            }
```

上述代码进行数据库的访问和查询。在上述代码中，首先需要创建一个连接对象进行数据库连接，然后再打开连接，打开连接之后就要编写 SELECT 语句进行数据库查询并填充到 DataSet 数据集中，并在 DataSet 数据集中遍历相应的表和列进行数据筛选。如果要查询 C_ID 为 1 的学生的所有姓名，有三个办法，这三个办法分别是：

- 修改 SQL 语句。
- 在循环内进行判断。
- 使用 LINQ 进行查询。

修改 SQL 语句是最方便的方法，直接在 SELECT 语句中添加查询条件 WHERE C-ID=1 就能够实现，但是这个方法扩展性非常的低，如果有其他需求则就需要修改 SQL 语句，也有可能造成其余代码填充数据集后数据集内容不同步。

在循环内进行判断也是一种方法，但是这个方法当循环增加时会造成额外的性能消耗，并且当需要扩展时，还需要修改循环代码。最方便的就是使用 LINQ 进行查询，在 Visual Studio 2008 中提供了 LINQ to SQL 类文件用于将现有的数据抽象成对象，这样就符合了面向对象的原则，同时也能够减少代码，提升

扩展性。创建一个 LINQ to SQL 类文件，直接将服务资源管理器中的相应表拖放到 LINQ to SQL 类文件可视化窗口中即可，如图 21-1 所示。

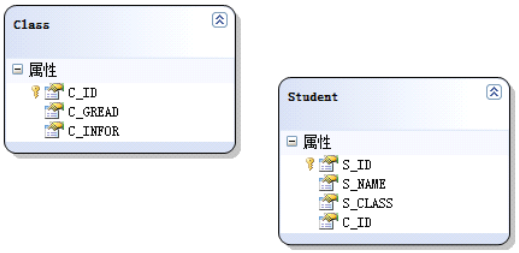

图 21-1　创建 LINQ to SQL 文件

创建了 LINQ to SQL 类文件后，就可以直接使用 LINQ to SQL 类文件提供的类进行查询，示例代码如下所示。

```
linqtosqlDataContext lq = new linqtosqlDataContext();
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l;        //执行查询
foreach (var result in mylq)                                                              //遍历集合
{
    Console.WriteLine(result.S_NAME.ToString());                                         //输出对象
}
```

上述代码只用了很短的代码就能够实现数据库中数据的查询和遍历，并且从可读性上来说也很容易理解，因为 LINQ 查询语句的语法基本与 SQL 语法相同，只要有一定的 SQL 语句基础就能够非常容易的编写 LINQ 查询语句。

3．数据集

LINQ 同样对数据集支持查询和筛选操作。其实数据集也是集合的表现形式，数据集除了能够填充数据库中的内容以外，开发人员还能够通过对数据集的操作向数据集中添加数据和修改数据。前面的章节中已经讲到，数据集可以看作是内存中的数据库。数据集能够模拟基本的数据库，包括表、关系等。这里就将 SQL Server 中的数据填充到数据集即可，示例代码如下所示。

```
try
{
    SqlConnection
    con = new SqlConnection("server='(local)';database='student';uid='sa';pwd='sa'"); //创建连接
    con.Open();                                                                       //打开连接
    string strsql = "select * from student,class where student.c_id=class.c_id";      //执行 SQL
    SqlDataAdapter da = new SqlDataAdapter(strsql, con);                              //创建适配器
    DataSet ds = new DataSet();                                                       //创建数据集
    da.Fill(ds, "mytable");                                                           //填充数据集
    DataTable tables = ds.Tables["mytable"];                                          //创建表
    var dslq = from d in tables.AsEnumerable() select d;                              //执行 LINQ 语句
    foreach (var res in dslq)
    {
        Console.WriteLine(res.Field<string>("S_NAME").ToString());                    //输出对象
    }
}
catch
{
```

```
            Console.WriteLine("数据库连接错误");
        }
```

上述代码使用 LINQ 针对数据集中的数据进行筛选和整理，同样能够以一种面向对象的思想进行数据集中数据的筛选。在使用 LINQ 进行数据集操作时，LINQ 不能直接从数据集对象中查询，因为数据集对象不支持 LINQ 查询，所以需要使用 AsEnumerable 方法返回一个泛型的对象以支持 LINQ 的查询操作，示例代码如下所示。

```
            var dslq = from d in tables.AsEnumerable() select d;                    //使用 AsEnumerable
```

上述代码使用 AsEnumerable 方法就可以让数据集中的表对象能够支持 LINQ 查询。

21.1.3 执行 LINQ 查询

从上一节可以看出 LINQ 在编程过程中极大的方便了开发人员对于业务逻辑的处理代码的编写，在传统的编程方法中复杂、冗余、难以实现的方法在 LINQ 中都能很好的解决。LINQ 不仅能够像 SQL 语句一样编写查询表达式，LINQ 最大的优点也包括 LINQ 作为编程语言的一部分，可以使用编程语言提供的特性进行 LINQ 条件语句的编写，这就弥补了 SQL 语句中的一些不足。在前面的章节中将一些复杂的查询和判断的代码简化成 LINQ 应用后，就能够执行应用程序判断 LINQ 是否查询和筛选出了所需要的值。

1. 数组

在数组数据源中，开发人员希望能够筛选出大于 5 的元素。开发人员将传统的代码修改成 LINQ 代码并通过 LINQ 查询语句进行筛选，示例代码如下所示。

```
            var st = from s in inter where s > 5    select s;                    //执行 LINQ 查询
```

上述代码将查询在inter数组中的所有元素并返回其中元素的值大于5的元素的集合，运行后如图 21-2 所示。

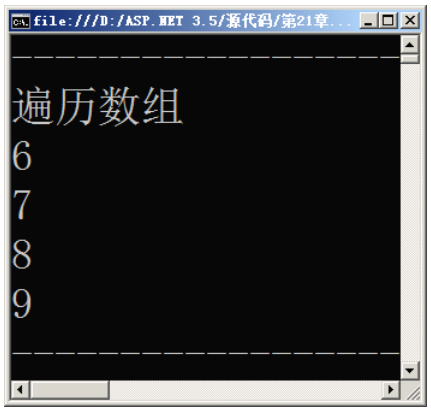

图 21-2　遍历数组

LINQ 执行了条件语句并返回了元素的值大于 5 的元素。LINQ 语句能够方便的扩展，当有不同的需求时，可以修改条件语句进行逻辑判断，例如可以筛选一个平方数为偶数的数组元素，直接修改条件即可，LINQ 查询语句如下所示。

```
            var st = from s in inter where (s*s)%2==0    select s;                    //执行 LINQ 查询
```

上述代码通过条件(s*s)%2==0 将数组元素进行筛选，选择平方数为偶数的数组元素的集合，运行后如图 21-3 所示。

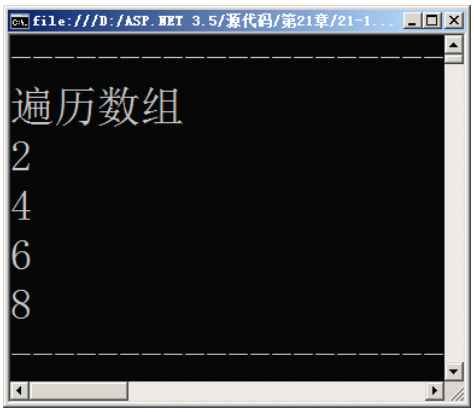

图 21-3　更改筛选条件

2．使用 SQL Server

在 LINQ to SQL 类文件中，LINQ to SQL 类文件已经将数据库的模型封装成一个对象，开发人员能够通过面向对象的思想访问和整合数据库。LINQ to SQL 也对 SQL 做了补充，使用 LINQ to SQL 类文件能够执行更强大的筛选，LINQ 查询语句代码如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l; //执行 LINQ 查询
```

上述代码从 Student 表和 Class 表中筛选了 C_ID 相等的学生信息，这很容易在 SQL 语句中实现。LINQ 作为编程语言的一部分，可以使用更多的编程方法实现不同的筛选需求，例如筛选名称中包含“郭”字的学生的名称在传统的 SQL 语句中就很难通过一条语句实现，而在 LINQ 中就能够实现，示例代码如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID where
l.S_NAME.Contains("郭") select l;                                    //执行 LINQ 条件查询
```

上述代码使用了 Contains 方法判断一个字符串中是否包含某个字符或字符串，这样不仅方便阅读，也简化了查询操作，运行后如图 21-4 和图 21-5 所示。

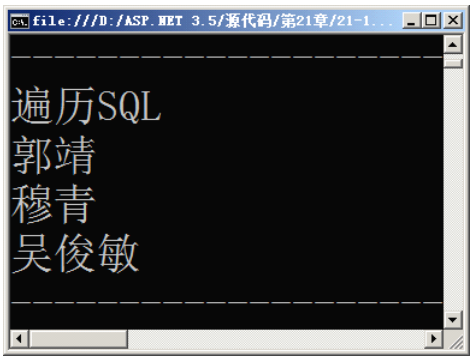

图 21-4　简单查询

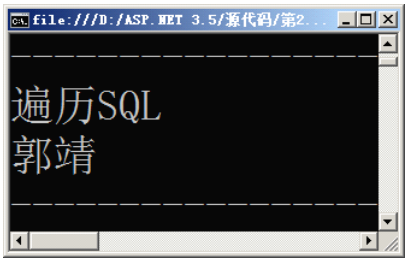

图 21-5　条件查询

LINQ 返回了符合条件的元素的集合，并实现了筛选操作。LINQ 不仅作为编程语言的一部分，简化了开发人员的开发操作，从另一方面讲，LINQ 也补充了在 SQL 中难以通过几条语句实现的功能的实现 。从上面的 LINQ 查询代码可以看出，就算是不同的对象、不同的数据源，其 LINQ 基本的查询语法都非常相似，并且 LINQ 还能够支持编程语言具有的特性从而弥补 SQL 语句的不足。在数据集的查询中，其查询语句也可以直接使用而无需大面积修改代码，这样代码就具有了更高的维护性和可读性。

21.2 LINQ 查询语法概述

从上面的章节中可以看出，LINQ 查询语句能够将复杂的查询应用简化成一个简单的查询语句，不仅如此，LINQ 还支持编程语言本有的特性进行高效的数据访问和筛选。虽然 LINQ 在写法上和 SQL 语句十分相似，但是 LINQ 语句在其查询语法上和 SQL 语句还是有出入的，SQL 查询语句如下所示。

```
select * from student,class where student.c_id=class.c_id                    //SQL 查询语句
```

上述代码是 SQL 查询语句，对于 LINQ 而言，其查询语句格式如下所示。

```
var mylq = from l in lq.Student from cl in lq.Class where l.C_ID==cl.C_ID select l; //LINQ 查询语句
```

上述代码作为 LINQ 查询语句实现了同 SQL 查询语句一样的效果，但是 LINQ 查询语句在格式上与 SQL 语句不同，LINQ 的基本格式如下所示。

```
var <变量> = from <项目> in <数据源> where <表达式> orderby <表达式>
```

LINQ 语句不仅能够支持对数据源的查询和筛选，同 SQL 语句一样，还支持 ORDER BY 等排序，以及投影等操作，示例查询语句如下所示。

```
var st = from s in inter where s==3 select s;                                //LINQ 查询
var st = from s in inter where (s * s) % 2 == 0 orderby s descending select s;  //LINQ 条件查询
```

从结构上来看，LINQ 查询语句同 SQL 查询语句中比较大的区别就在于 SQL 查询语句中的 SELECT 关键字在语句的前面，而在 LINQ 查询语句中 SELECT 关键字在语句的后面，在其他地方没有太大的区别，对于熟悉 SQL 查询语句的人来说非常容易上手。

21.3 基本子句

既然 LINQ 查询语句同 SQL 查询语句一样，能够执行条件、排序等操作，这些操作就需要使用 WHERE、ORDERBY 等关键字，这些关键字在 LINQ 中是基本子句。同 SQL 查询语句中的 WHERE、ORDER BY 操作一样，都为元素进行整合和筛选。

21.3.1 from 查询子句

from 子句是 LINQ 查询语句中最基本也是最关键的子句关键字，与 SQL 查询语句不同的是，from 关键字必须在 LINQ 查询语句的开始。

1．from 查询子句基础

后面跟随着项目名称和数据源，示例代码如下所示。

```
var linqstr = from lq in str select lq;                                      //form 子句
```

from 语句指定项目名称和数据源，并且指定需要查询的内容，其中项目名称作为数据源的一部分而存在，用于表示和描述数据源中的每个元素，而数据源可以是数组、集合、数据库甚至是 XML。值得一提的是，from 子句的数据源的类型必须为 IEnumerable、IEnumerable<T>类型或者是 IEnumerable、IEnumerable<T>的派生类，否则 from 不能够支持 LINQ 查询语句。

在.NET Framework 中泛型编程中，List（可通过索引的强类型列表）也能够支持 LINQ 查询语句的 from 关键字，因为 List 实现了 IEnumerable、IEnumerable<T>类型，在 LINQ 中可以对 List 类进行查询，示例代码如下所示。

```
static void Main(string[] args)
{
```

```
            List<string> MyList = new List<string>();                    //创建一个列表项
            MyList.Add("guojing");                                       //添加一项
            MyList.Add("wujunmin");                                      //添加一项
            MyList.Add("muqing");                                        //添加一项
            var linqstr = from l in MyList select l;                     //LINQ 查询
            foreach (var element in linqstr)                             //遍历集合
            {
                Console.WriteLine(element.ToString());                   //输出对象
            }
            Console.ReadKey();
        }
```

上述代码创建了一个列表项并向列表中添加若干项进行 LINQ 查询。由于 List<T>实现了 IEnumerable、IEnumerable<T>，所以 List<T>列表项可以支持 LINQ 查询语句的 from 关键字，如图 21-6 所示。

图 21-6　from 子句

顾名思义，from 语句可以被理解为“来自”，而 in 可以被理解为“在哪个数据源中”，这样 from 语句就很好理解了，如 from l in MyList select l 语句可以翻译成“找到来自 MyList 数据源中的集合 l”，这样就能够更加方便的理解 from 语句。

2．from 查询子句嵌套查询

在 SQL 语句中，为了实现某一功能，往往需要包含多个条件，以及包含多个 SQL 子句嵌套。在 LINQ 查询语句中，并没有 and 关键字为复合查询提供功能。如果需要进行复杂的复合查询，可以在 from 子句中嵌套另一个 from 子句即可，示例代码如下所示。

```
            var linqstr = from lq in str from m in str2 select lq;       //使用嵌套查询
```

上述代码就使用了一个嵌套查询进行 LINQ 查询。在有多个数据源或者包括多个表的数据需要查询时，可以使用 LINQfrom 子句嵌套查询，数据源示例代码如下所示。

```
            List<string> MyList = new List<string>();                    //创建一个数据源
            MyList.Add("guojing");                                       //添加一项
            MyList.Add("wujunmin");                                      //添加一项
            MyList.Add("muqing");                                        //添加一项
            MyList.Add("yuwen");                                         //添加一项
            List<string> MyList2 = new List<string>();                   //创建另一个数据源
            MyList2.Add("guojing's phone");                              //添加一项
            MyList2.Add("wujunmin's phone ");                            //添加一项
            MyList2.Add("muqing's phone ");                              //添加一项
            MyList2.Add("lupan's phone ");                               //添加一项
```

上述代码创建了两个数据源，其中一个数据源存放了联系人的姓名的拼音名称，另一个则存放了联系人的电话信息。为了方便的查询在数据源中“联系人”和“联系人电话”都存在并且匹配的数据，就需要使用 from 子句嵌套查询，示例代码如下所示。

```
        var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select l; //from 子句嵌套查询
        foreach (var element in linqstr)                                               //遍历集合元素
        {
            Console.WriteLine(element.ToString());                                     //输出对象
        }
        Console.ReadKey();
```

上述代码使用了 LINQ 语句进行嵌套查询，嵌套查询在 LINQ 中会被经常使用到，因为开发人员常常遇到需要面对多个表多个条件，以及不同数据源或数据源对象的情况，使用 LINQ 查询语句的嵌套查询可以方便的在不同的表和数据源对象之间建立关系。

21.3.2 where 条件子句

在 SQL 查询语句中可以使用 where 子句进行数据的筛选，在 LINQ 中同样包括 where 子句进行数据源中数据的筛选。where 子句指定了筛选的条件，这也就是说在 where 子句中的代码段必须返回布尔值才能够进行数据源的筛选，示例代码如下所示。

```
        var linqstr = from l in MyList where l.Length > 5 select l;                    //编写 where 子句
```

LINQ 查询语句可以包含一个或多个 where 子句，而 where 子句可以包含一个或多个布尔值变量，为了查询数据源中姓名的长度在 6 之上的姓名，可以使用 where 子句进行查询，示例代码如下所示。

```
    static void Main(string[] args)
    {
        List<string> MyList = new List<string>();                                     //创建 List 对象
        MyList.Add("guojing");                                                         //添加一项
        MyList.Add("wujunmin");                                                        //添加一项
        MyList.Add("muqing");                                                          //添加一项
        MyList.Add("yuwen");                                                           //添加一项
        var linqstr = from l in MyList where l.Length > 6 select l;                    //执行 where 查询
        foreach (var element in linqstr)                                               //遍历集合
        {
            Console.WriteLine(element.ToString());                                     //输出对象
        }
        Console.ReadKey();
    }
```

上述代码添加了数据源之后，通过 where 子句在数据源中进行条件查询，LINQ 查询语句会遍历数据源中的数据并进行判断，如果返回值为 true，则会在 linqstr 集合中添加该元素，运行后如图 21-7 所示 。

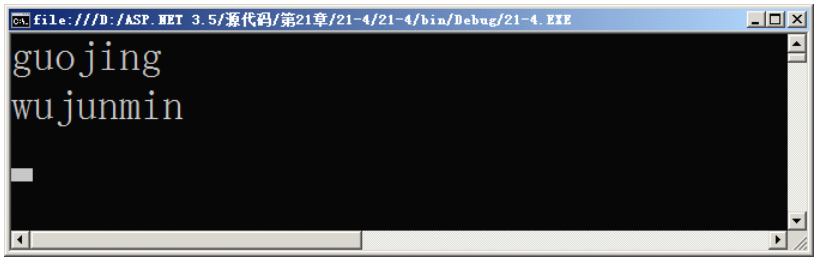

图 21-7　where 子句查询

当需要多个 where 子句进行复合条件查询时，可以使用“&&”进行 where 子句的整合，示例代码如下所示。

```
    static void Main(string[] args)
```

```
        {
            List<string> MyList = new List<string>();                                    //创建 List 对象
            MyList.Add("guojing");                                                       //添加一项
            MyList.Add("wujunmin");                                                      //添加一项
            MyList.Add("muqing");                                                        //添加一项
            MyList.Add("guomoruo");                                                      //添加一项
            MyList.Add("lupan");                                                         //添加一项
            MyList.Add("guof");                                                          //添加一项
            var linqstr = from l in MyList where (l.Length > 6 && l.Contains("guo"))||l=="lupan" select l;//复合查
询
            foreach (var element in linqstr)                                            //遍历集合
            {
                Console.WriteLine(element.ToString());                                  //输出对象
            }
            Console.ReadKey();
        }
```

上述代码进行了多条件的复合查询，查询姓名长度大于 6 并且姓名中包含 guo 的姓或者姓名是“lupan”的人，运行后如图 21-8 所示。

图 21-8　复合 where 子句查询

复合 where 子句查询通常用于同一个数据源中的数据查询，当需要在同一个数据源中进行筛选查询时，可以使用 where 子句进行单个或多个 where 子句条件查询，where 子句能够对数据源中的数据进行筛选并将复合条件的元素返回到集合中。

21.3.3　select 选择子句

select 子句同 from 子句一样，是 LINQ 查询语句中必不可少的关键字，select 子句在 LINQ 查询语句中是必须的，示例代码如下所示。

```
            var linqstr = from lq in str select lq;                                  //编写选择子句
```

上述代码中包括三个变量，这三个变量分别为 linqstr、lq、str。其中 str 是数据源，linqstr 是数据源中满足查询条件的集合，而 lq 也是一个集合，这个集合来自数据源。在 LINQ 查询语句中必须包含 select 子句，若不包含 select 子句则系统会抛出异常（除特殊情况外）。select 语句指定了返回到集合变量中的元素是来自哪个数据源的，示例代码如下所示。

```
        static void Main(string[] args)
        {
            List<string> MyList = new List<string>();                               //创建 List
            MyList.Add("guojing");                                                   //添加一项
            MyList.Add("wujunmin");                                                  //添加一项
            MyList.Add("guomoruo");                                                  //添加一项
```

```
            List<string> MyList2 = new List<string>();                                  //创建 List
            MyList2.Add("guojing's phone");                                             //添加一项
            MyList2.Add("wujunmin's phone ");                                           //添加一项
            MyList2.Add("lupan's phone ");                                              //添加一项
            var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select l;   //select l 变量
            foreach (var element in linqstr)                                            //遍历集合
            {
                Console.WriteLine(element.ToString());                                  //输出集合内容
            }
            Console.ReadKey();                                                          //等待用户按键
        }
```

上述代码从两个数据源中筛选数据，并通过 select 返回集合元素，运行后如图 21-9 所示。

图 21-9　select 子句

如果将 select 子句后面的项目名称更改，则结果可能不同，更改 LINQ 查询子句代码如下所示。

```
            var linqstr = from l in MyList from m in MyList2 where m.Contains(l) select m;      //使用 select
```

上述 LINQ 查询子句并没有 select l 变量中的集合元素，而是选择了 m 集合元素，则返回的应该是 MyList2 数据源中的集合元素，运行后如图 21-10 所示。

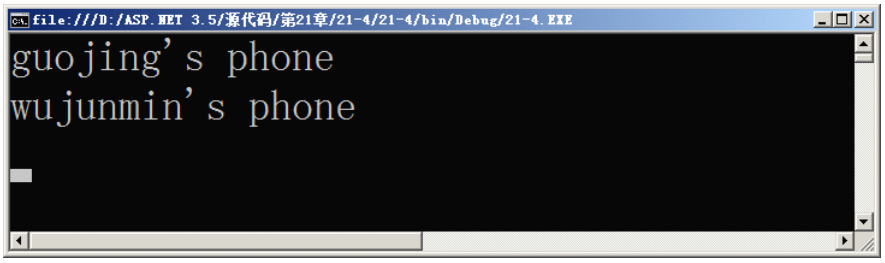

图 21-10　select 子句

对于不同的 select 对象返回的结果也不尽相同，当开发人员需要进行复合查询时，可以通过 select 语句返回不同的复合查询对象，这在多数据源和多数据对象查询中是非常有帮助的。

21.3.4　group 分组子句

在 LINQ 查询语句中，group 子句对 from 语句执行查询的结果进行分组，并返回元素类型为 IGrouping<TKey,TElement>的对象序列。group 子句支持将数据源中的数据进行分组。但进行分组前，数据源必须支持分组操作才可使用 group 语句进行分组处理，示例代码如下所示。

```
    public class Person
    {
        public int age;                                                 //分组条件
        public string name;                                             //创建姓名字段
```

```
        public Person(int age,string name)                          //构造函数
        {
            this.age = age;                                         //构造属性值 age
            this.name = name;                                       //构造属性值 name
        }
    }
```

上述代码设计了一个类用于描述联系人的姓名和年级，并且按照年级进行分组，这样数据源就能够支持分组操作。

注意：虽然数组也可以进行分组操作，因为其绝大部分数据源都能够支持分组操作，但是数组等数据源进行分组操作可能是没有意义的。

这里同样可以通过 List 列表以支持 LINQ 查询，示例代码如下所示。

```
        static void Main(string[] args)
        {
            List<Person> PersonList = new List<Person>();
            PersonList.Add(new Person(21,"limusha"));                   //通过构造函数构造新对象
            PersonList.Add(new Person(21, "guojing"));                  //通过构造函数构造新对象
            PersonList.Add(new Person(22, "wujunmin"));                 //通过构造函数构造新对象
            PersonList.Add(new Person(22, "lupan"));                    //通过构造函数构造新对象
            PersonList.Add(new Person(23, "yuwen"));                    //通过构造函数构造新对象
            var gl = from p in PersonList group p by p.age;             //使用 group 子句进行分组
            foreach (var element in gl)                                 //遍历集合
            {
                foreach (Person p in element)                           //遍历集合
                {
                    Console.WriteLine(p.name.ToString());               //输出对象
                }
            }
            Console.ReadKey();
        }
```

上述代码使用了 group 子句进行数据分组，实现了分组的功能，运行后如图 21-11 所示。

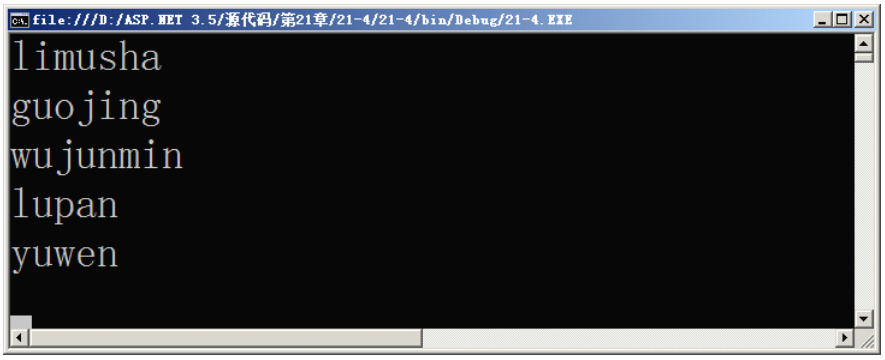

图 21-11　group 子句

正如图 21-11 所示，group 子句将数据源中的数据进行分组，在遍历数据元素时，并不像前面的章节那样直接对元素进行遍历，因为 group 子句返回的是元素类型为 IGrouping<TKey,TElement>的对象序列，必须在循环中嵌套一个对象的循环才能够查询相应的数据元素。

在使用 group 子句时，LINQ 查询子句的末尾并没有 select 子句，因为 group 子句会返回一个对象序列，通过循环遍历才能够在对象序列中寻找到相应的对象的元素，如果使用 group 子句进行分组操作，

可以不使用 select 子句。

21.3.5 orderby 排序子句

在 SQL 查询语句中，常常需要对现有的数据元素进行排序，例如注册用户的时间，以及新闻列表的排序，这样能够方便用户在应用程序使用过程中快速获取需要的信息。在 LINQ 查询语句中同样支持排序操作以提取用户需要的信息。在 LINQ 语句中，orderby 是一个词组而不是分开的，orderby 能够支持对象的排序，例如按照用户的年龄进行排序时就可以使用 orderby 关键字，示例代码如下所示。

```
public class Person                                          //创建对象
{
    public int age;                                          //创建字段
    public string name;                                      //创建字段
    public Person(int age,string name)                       //构造函数
    {
        this.age = age;                                      //赋值字段
        this.name = name;
    }
}
```

上述代码同样设计了一个 Person 类，并通过 age、name 字段描述类对象。使用 LINQ，同样要使用 List 类作为对象的容器并进行其中元素的查询，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        List<Person> PersonList = new List<Person>();                    //创建对象列表
        PersonList.Add(new Person(21,"limusha"));                        //年龄为 21
        PersonList.Add(new Person(23, "guojing"));                       //年龄为 23
        PersonList.Add(new Person(22, "wujunmin"));                      //年龄为 22
        PersonList.Add(new Person(25, "lupan"));                         //年龄为 25
        PersonList.Add(new Person(24, "yuwen"));                         //年龄为 24
        var gl = from p in PersonList orderby p.age select p;            //执行排序操作
        foreach (var element in gl)                                      //遍历集合
        {
            Console.WriteLine(element.name.ToString());                  //输出对象
        }
        Console.ReadKey();
    }
}
```

上述代码并没有按照顺序对 List 容器添加对象，其中数据的显示并不是按照顺序来显示的。使用 orderby 关键字能够指定集合中的元素的排序规则，上述代码按照年龄的大小进行排序，运行后如图 21-12 所示。

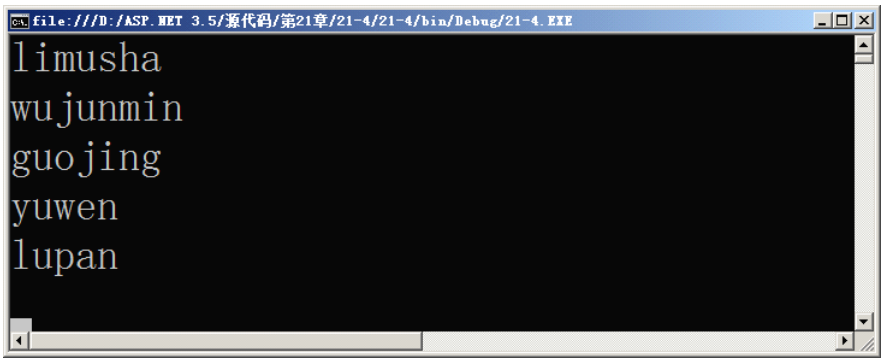

图 21-12　orderby 子句

orderby 子句同样能够实现倒序排列，倒序排列在应用程序开发过程中应用的非常广泛，例如新闻等。用户关心的都是当天的新闻而不是很久以前发布的某个新闻，如果管理员发布了一个新的新闻，显示在最上方的应该是最新的新闻。在 orderby 子句中可以使用 descending 关键字进行倒序排列，示例代码如下所示。

```
var gl = from p in PersonList orderby p.age descending select p;          //orderby 语句
```

上述代码将用户的信息按照其年龄的大小倒序排列，运行如图 21-13 所示。

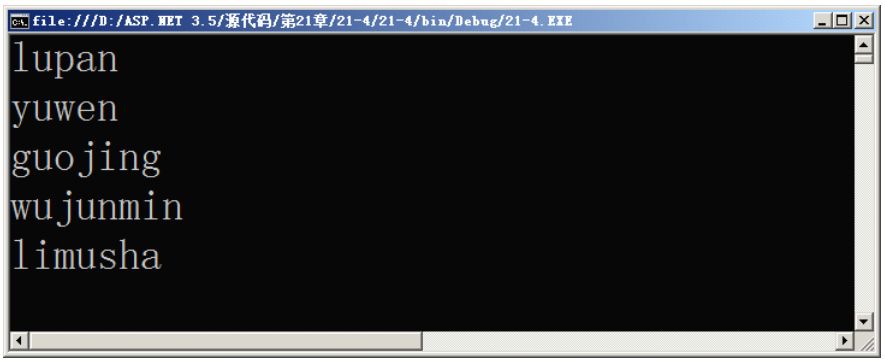

图 21-13　orderby 子句倒序

orderby 子句同样能够进行多个条件排序，如果需要使用 orderby 子句进行多个条件排序，只需要将这些条件用“，”号分割即可，示例代码如下所示。

```
var gl = from p in PersonList orderby p.age descending,p.name select p;     //orderby 语句
```

21.3.6　into 连接子句

into 子句通常和 group 子句一起使用，通常情况下，LINQ 查询语句中无需 into 子句，但是如果需要对分组中的元素进行操作，则需要使用 into 子句。into 语句能够创建临时标识符用于保存查询的集合，示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();                    //创建对象列表
    PersonList.Add(new Person(21, "limusha"));                       //通过构造函数构造新对象
    PersonList.Add(new Person(21, "guojing"));                       //通过构造函数构造新对象
    PersonList.Add(new Person(22, "wujunmin"));                      //通过构造函数构造新对象
    PersonList.Add(new Person(22, "lupan"));                         //通过构造函数构造新对象
```

```
        PersonList.Add(new Person(23, "yuwen"));                    //通过构造函数构造新对象
        var gl = from p in PersonList group p by p.age into x select x;   //使用 into 子句创建标识
        foreach (var element in gl)                                 //遍历集合
        {
            foreach (Person p in element)                           //遍历集合
            {
                Console.WriteLine(p.name.ToString());               //输出对象
            }
        }
        Console.ReadKey();
    }
```

上述代码通过使用 into 子句创建标识，从 LINQ 查询语句中可以看出，查询后返回的是一个集合变量 x 而不是 p，但是编译能够通过并且能够执行查询，这说明 LINQ 查询语句将查询的结果填充到了临时标识符对象 x 中并返回查询集合给 gl 集合变量，运行结果如图 21-14 所示。

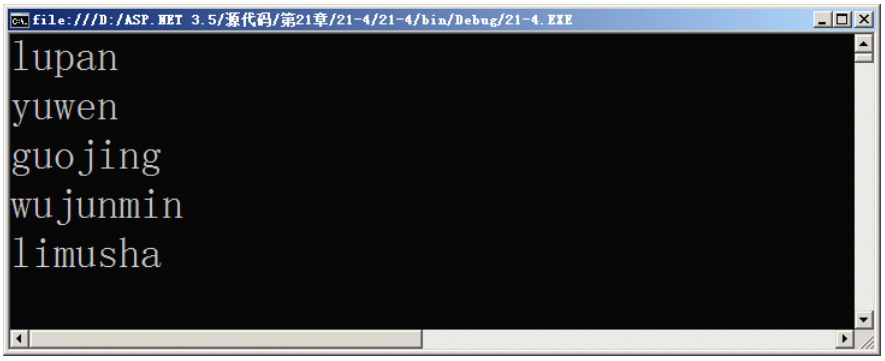

图 21-14　into 子句

注意：into 子句必须以 select、group 等子句作为结尾子句，否则会抛出异常。

21.3.7　join 连接子句

在数据库的结构中，通常表与表之间有着不同的联系，这些联系决定了表与表之间的依赖关系。在 LINQ 中同样也可以使用 join 子句对有关系的数据源或数据对象进行查询，但首先这两个数据源必须要有一定的联系，示例代码如下所示。

```
public class Person                                                 //描述“人”对象
{
    public int age;                                                 //描述“年龄”字段
    public string name;                                             //描述“姓名”字段
    public string cid;                                              //描述“车 ID”字段
    public Person(int age,string name,int cid)                      //构造函数
    {
        this.age = age;                                             //初始化
        this.name = name;                                           //初始化
        this.cid = cid;
    }
}
public class CarInformaion                                          //描述“车”对象
{
```

```
        public int cid;                                          //描述“车 ID”字段
        public string type;                                      //描述“车类型”字段
        public CarInformaion(int cid,string type)                //初始化构造函数
        {
            this.cid = cid;                                      //初始化
            this.type = type;                                    //初始化
        }
    }
```

上述代码创建了两个类，这两个类分别用来描述“人”这个对象和“车”这个对象，CarInformation 对象可以用来描述车的编号以及车的类型，而 Person 类可以用来描述人购买了哪个牌子的车，这就确定了这两个类之间的依赖关系。而在对象描述中，如果将 CarInformation 类的属性和字段放置到 Person 类的属性中，会导致类设计臃肿，同时也没有很好的描述该对象。对象创建完毕就可以使用 List 类创建对象，示例代码如下所示。

```
            List<Person> PersonList = new List<Person>();        //创建 List 类
            PersonList.Add(new Person(21, "limusha",1));         //购买车 ID 为 1 的人
            PersonList.Add(new Person(21, "guojing",2));         //购买车 ID 为 2 的人
            PersonList.Add(new Person(22, "wujunmin",3));        //购买车 ID 为 3 的人
            List<CarInformaion> CarList = new List<CarInformaion>();
            CarList.Add(1, "宝马");                              //车 ID 为 1 的基本信息
            CarList.Add(2, "奇瑞");
```

上述代码分别使用了 List 类进行对象的初始化，使用 join 子句就能够进行不同数据源中数据关联的操作和外连接，示例代码如下所示。

```
        static void Main(string[] args)
        {
            List<Person> PersonList = new List<Person>();        //创建 List 类
            PersonList.Add(new Person(21, "limusha",1));         //购买车 ID 为 1 的人
            PersonList.Add(new Person(21, "guojing",2));         //购买车 ID 为 2 的人
            PersonList.Add(new Person(22, "wujunmin",3));        //购买车 ID 为 3 的人
            List<CarInformaion> CarList = new List<CarInformaion>();  //创建 List 类
            CarList.Add(new CarInformaion(1,"宝马"));            //车 ID 为 1 的车
            CarList.Add(new CarInformaion(2, "奇瑞"));           //车 ID 为 2 的车
            var gl = from p in PersonList join car in CarList on p.cid equals car.cid select p;//使用 join 子句
            foreach (var element in gl)                          //遍历集合
            {
                Console.WriteLine(element.name.ToString());      //输出对象
            }
            Console.ReadKey();
        }
```

上述代码使用 join 子句进行不同数据源之间关系的创建，其用法同 SQL 查询语句中的 INNER JOIN 查询语句相似，运行后如图 21-15 所示。

图 21-15　join 查询子句

21.3.8　let 临时表达式子句

在 LINQ 查询语句中，let 关键字可以看作是在表达式中创建了一个临时的变量用于保存表达式的结果，但是 let 子句指定的范围变量的值只能通过初始化操作进行赋值，一旦初始化之后就无法再次进行更改操作。示例代码如下所示。

```
static void Main(string[] args)
{
    List<Person> PersonList = new List<Person>();                                    //创建 List 类
    PersonList.Add(new Person(21, "limusha",1));                                     //购买车 ID 为 1 的人
    PersonList.Add(new Person(21, "guojing",2));                                     //购买车 ID 为 2 的人
    PersonList.Add(new Person(22, "wujunmin",3));                                    //购买车 ID 为 3 的人
    List<CarInformaion> CarList = new List<CarInformaion>();                         //创建 List 类
    CarList.Add(new CarInformaion(1,"宝马"));                                        //车 ID 为 1 的车
    CarList.Add(new CarInformaion(2, "奇瑞"));                                       //车 ID 为 2 的车
    var gl = from p in PersonList let car = from c in CarList select c.cid select p;        //使用 let 语句
    foreach (var element in gl)                                                      //遍历集合
    {
        Console.WriteLine(element.name.ToString());                                 //输出对象
    }
    Console.ReadKey();
}
```

let 就相当于是一个中转变量，用于临时存储表达式的值，在 LINQ 查询语句中，其中的某些过程的值可以通过 let 进行保存。而简单的说，let 就是临时变量，如 x=1+1、y=x+2 这样，其中 x 就相当于是一个 let 变量，上述代码运行后如图 21-16 所示。

图 21-16　let 子句

21.4　LINQ 查询操作

前面介绍了 LINQ 的一些基本的语法，以及 LINQ 常用的查询子句进行数据的访问和整合，甚至建立数据源对象和数据源对象之间的关联，使用 LINQ 查询子句能够实现不同的功能，包括投影、排序和聚合等，本节开始介绍 LINQ 的查询操作。

21.4.1 LINQ 查询概述

LINQ 不仅提供了强大的查询表达式为开发人员对数据源进行查询和筛选操作提供遍历，LINQ 还提供了大量的查询操作，这些操作通过实现 IEnumerable<T>或 IQueryable<T>提供的接口实现了投影、排序、聚合等操作。通过使用 LINQ 提供的查询方法，能够快速的实现投影、排序等操作。

由于 LINQ 查询操作实现了 IEnumerable<T>或 IQueryable<T>接口，所以 LINQ 查询操作能够通过接口中特定的方法进行查询和筛选，可以直接使用数据源对象变量的方法进行操作。在 LINQ 查询操作的方法中，需要大量的使用 Lambda 表达式实现委托，这就从另一个方面说明了 Lambda 表达式的重要性。示例代码如下所示。

```
int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };          //创建数组
var lint = inter.Select(i => i);                                               //使用 Lambda
```

上述代码使用了 Select 方法进行投影操作，在投影操作的参数中使用 Lambda 表达式表示了如何实现数据筛选。LINQ 查询操作不仅包括 Select 投影操作，还包括排序、聚合等操作，LINQ 常用操作如下所示。

- Count：计算集合中元素的数量，或者计算满足条件的集合的元素的数量。
- GroupBy：实现对集合中的元素进行分组的操作。
- Max：获取集合中元素的最大值。
- Min：获取集合中元素的最小值。
- Select：执行投影操作。
- SelectMany：执行投影操作，可以为多个数据源进行投影操作。
- Where：执行筛选操作。

LINQ 不只提供上述这些常用的查询操作方法，还提供更多的查询方法，由于本书篇幅有限，只讲解一些常用的查询方法。

21.4.2 投影操作

投影操作和 SQL 查询语句中的 SELECT 基本类似，投影操作能够指定数据源并选择相应的数据源，在 LINQ 中常用的投影操作包括 Select 和 SelectMany。

1．Select 选择子句

Select 操作能够将集合中的元素投影到新的集合中去，并能够指定元素的类型和表现形式，示例代码如下所示。

```
static void Main(string[] args)
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };      //创建数组
    var lint = inter.Select(i => i);                                          //Select 操作
    foreach (var m in lint)                                                   //遍历集合
    {
        Console.WriteLine(m.ToString());                                      //输出对象
    }
    Console.ReadKey();
}
```

上述代码将数据源进行了投影操作，使用 Select 进行投影操作非常简单，其作用同 SQL 语句中的 SELECT 语句十分相似，上述代码将集合中的元素进行投影并将符合条件的元素投影到新的集合中 lint

去。

2．SelectMany 多重选择子句

SelectMany 和 Select 的用法基本相同，但是 SelectMany 与 Select 相比可以选择多个序列进行投影，示例代码如下所示。

```
static void Main(string[] args)
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };        //创建数组
    int[] inter2 = { 21, 22, 23, 24, 25, 26};                                   //创建数组
    List<int[]> list = new List<int[]>();                                       //创建 List
    list.Add(inter);                                                            //添加对象
    list.Add(inter2);                                                           //添加对象
    var lint = list.SelectMany(i => i);                                         //SelectMany 操作
    foreach (var m in lint)                                                     //遍历集合
    {
        Console.WriteLine(m.ToString());                                        //输出对象
    }
    Console.ReadKey();
}
```

上述代码通过 SelectMany 方法将不同的数据源投影到一个新的集合中，运行结果如图 21-17 所示。

图 21-17　SelectMany 投影操作

21.4.3　筛选操作

筛选操作使用的是 Where 方法，其使用方法同 LINQ 查询语句中的 where 子句使用方法基本相同，筛选操作用于筛选符合特定逻辑规范的集合的元素，示例代码如下所示。

```
public static void WhereQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };        //创建数组
    var lint = inter.Where(i => i > 5);                                         //使用 where 进行筛选操作
    foreach (var m in lint)                                                     //遍历集合
    {
        Console.WriteLine(m.ToString());                                        //输出对象
    }
    Console.ReadKey();
}
```

上述代码通过 Where 方法和 Lambda 表达式实现了对数据源中数据的筛选操作，其中 Lambda 表达式筛选了现有集合中所有值大于 5 的元素并填充到新的集合中，使用 LINQ 查询语句的子查询语句同样

能够实现这样的功能，示例代码如下所示。

```
var lint = from i in inter where i > 5 select i;                    //执行筛选操作
```

上述代码同样实现了 LINQ 中的筛选操作 Where，但是使用筛选操作的代码更加简洁，上述代码运行后如图 21-18 所示。

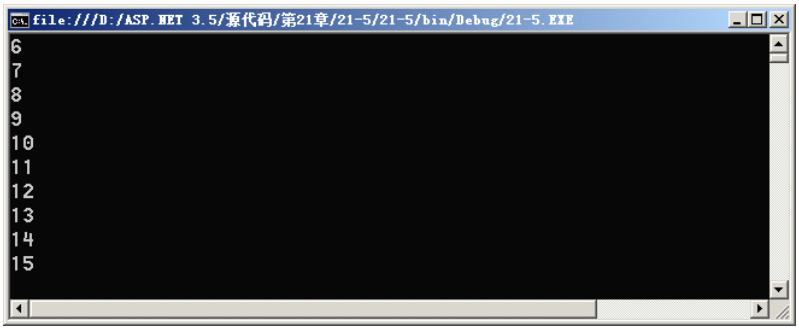

图 21-18　筛选操作

21.4.4　排序操作

排序操作最常使用的是 OrderBy 方法，其使用方法同 LINQ 查询子句中的 orderby 子句基本类似，使用 OrderBy 方法能够对集合中的元素进行排序，同样 OrderBy 方法能够针对多个参数进行排序。排序操作不仅提供了 OrderBy 方法，还提供了其他的方法进行高级排序，这些方法包括：

- OrderBy 方法：根据关键字对集合中的元素按升序排列。
- OrderByDescending 方法：根据关键字对集合中的元素按倒序排列。
- ThenBy 方法：根据次要关键字对序列中的元素按升序排列。
- ThenByDescending 方法：根据次要关键字对序列中的元素按倒序排列。
- Reverse 方法：将序列中的元素的顺序进行反转。

使用 LINQ 提供的排序操作能够方便的进行排序，示例代码如下所示。

```
public static void OrderByQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };          //创建数组
    var lint = inter.OrderByDescending(i => i);                                   //使用倒序方法
    foreach (var m in lint)                                                       //遍历集合
    {
        Console.WriteLine(m.ToString());                                          //输出对象
    }
    Console.ReadKey();
}
```

上述代码使用了 OrderByDescending 方法将数据源中的数据进行倒排，除此之外，还可以使用 Reverse 将集合内的元素进行反转，示例代码如下所示。

```
public static void OrderByQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };          //创建数组
    var lint = inter.Reverse();                                                   //反转集合
    foreach (var m in lint)                                                       //遍历集合
    {
        Console.WriteLine(m.ToString());                                          //输出对象
```

```
        }
        Console.ReadKey();
    }
```

上述代码使用了 Reverse 元素将集合内的元素进行反转，运行结果如图 21-19 所示。

图 21-19　排序操作

注意：排序和反转并不相同，排序是将集合中的元素进行排序，可以是正序也可以是倒序，而反转并没有进行排序，只是讲集合中的元素从第一个放到最后一个，依次反转而已。

21.4.5　聚合操作

在 SQL 中，往往需要统计一些基本信息，例如今天有多少人留言，今天有多少人访问过网站，这些都可以通过 SQL 语句进行查询。在 SQL 查询语句中，支持一些能够进行基本运算的函数，这些函数包括 Max、Min 等。在 LINQ 中，同样包括这些函数，用来获取集合中的最大值和最小值等一些常用的统计信息，在 LINQ 中，这种操作被称为聚合操作。聚合操作常用的方法有：

- Count 方法：获取集合中元素的数量，或者获取满足条件的元素数量。
- Sum 方法：获取集合中元素的总和。
- Max 方法：获取集合中元素的最大值。
- Min 方法：获取集合中元素的最小值。
- Average 方法：获取集合中元素的平均值。
- Aggregate 方法：对集合中的元素进行自定义的聚合计算。
- LongCount 方法：获取集合中元素的数量，或者计算序列满足一定条件的元素的数量。一般计算大型集合中的元素的数量。

1. Max、Min、Count、Average 内置方法

通过 LINQ 提供的聚合操作的方法能够快速的获取统计信息，如要找到数据源中数据的最大值，可以使用 Max 方法，示例代码如下所示。

```
    public static void CountQuery()
    {
        int[] inter = { 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };          //创建数组
        var Maxlint = inter.Max();                                                        //获取最大值
        var Minlint = inter.Min();                                                        //获取最小值
        Console.WriteLine("最大值是" + Maxlint.ToString());                               //输出最大值
        Console.WriteLine("最小值是" + Minlint.ToString());                               //输出最小值
        Console.ReadKey();
    }
```

上述代码在获取最大值和最小值时并没有使用 Lambda 表达式，因为数据源中并没有复杂的对象，

所以可以默认不使用 Lambda 表达式就能够返回相应的值，如果要编写 Lambda 表达式，可以编写相应代码如下所示。

```
var Maxlint = inter.Max(i => i);                                    //获取最大值
var Minlint = inter.Min(i => i);                                    //获取最小值
```

聚合操作还能够获取平均值和获取集合中元素的数量，示例代码如下所示。

```
public static void CountQuery2()
{
    int[] inter = { 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };   //创建数组
    var Countlint = inter.Count(i => i > 5);                                    //获取元素数量
    var Arrlint = inter.Average(i => i);                                        //获取平均值
    Console.WriteLine("复合条件的集合有" + Countlint.ToString()+"项");          //输出项数
    Console.WriteLine("平均值为" + Arrlint.ToString());                         //输出平均值
    Console.ReadKey();
}
```

上述代码通过 Count 方法获得符合相应条件的元素的数量，并通过 Average 方法获取平均值，运行后如图 21-20 所示。

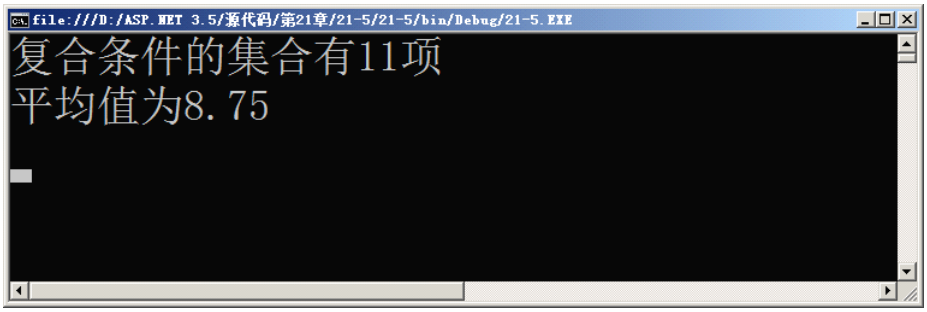

图 21-20　Count 和 Average 方法

在编写查询操作时，可以通过编写条件来规范查询范围，例如上述代码使用 Count 的条件就编写了 i=>i>5 的 Lambda 表达式，该表达式会返回符合该条件的集合再进行方法运算。

2．Aggregate 聚合方法

Aggregate 方法能够对集合中的元素进行自定义的聚合计算，开发人员能够使用 Aggregate 方法实现类似 Sum、Count 等聚合计算，示例代码如下所示。

```
public static void AggregateQuery()
{
    int[] inter = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };   //创建数组
    var aq = inter.Aggregate((x,y)=>x+y);                                   //使用 Aggregate 方法
    Console.WriteLine(aq.ToString());                                       //实现 Sum 方法
    Console.ReadKey();
}
```

上述代码通过编写 Lambda 表达式实现了数据源中所有数据的加法，也就是实现了 Sum 聚合方法，运行后如图 21-21 所示。

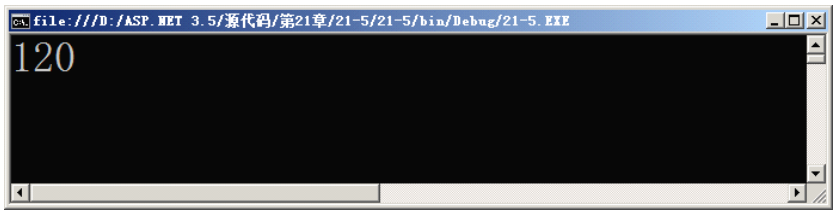

图 21-21　自定义聚合操作

LINQ 不仅仅包括这些查询操作方法，LINQ 还包括集合操作，删除集合中重复的元素，也能够计算集合与集合之间的并集差集等。LINQ 查询操作不仅提供了最基本的投影、筛选、聚合等操作，还能够极大的简化集合的开发，实现集合和集合中元素的操作。

21.5　使用 LINQ 查询和操作数据库

讲解了关于 LINQ 的基本知识，就需要使用 LINQ 进行数据库操作，LINQ 能够支持多个数据库并为每种数据库提供了便捷的访问和筛选方案，本书主要使用 SQL Server 2005 作为数据源进行 LINQ 查询和操作数据示例数据库。

21.5.1　简单查询

LINQ 提供了快速查询数据库的方法，这个方法非常的简单，在前面的章节中已经讲到，这里使用 21.1.1 中准备的 student 数据库作为数据源，其表结构和数据都已经创建完毕，只需要进行简单查询即可。首先创建一个 LINQ to SQL 文件，名称为 MyLinq.dbml，并将需要查询的表拖动到视图中，这里需要拖动 Class 表和 Student 表作为数据源，如图 21-22 所示。

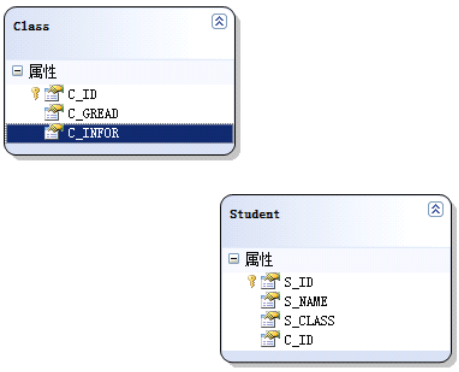

图 21-22　数据库关系图

创建了文件并拖动了相应的数据库关系图后，就可以保存并编写相应的代码进行查询了，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    MyLinqDataContext dc = new MyLinqDataContext();                          //创建对象
    var StudentList = from d in dc.Student orderby d.S_ID descending select d;   //执行查询
    foreach (var stu in StudentList)                                         //遍历元素
```

```
        {
            Response.Write("学生姓名为" + stu.S_NAME.ToString()+"<br/>");          //输出 HTML 字串
        }
    }
```

上述代码直接使用 LINQ to SQL 文件提供的类进行数据查询和筛选，运行后如图 21-23 所示。

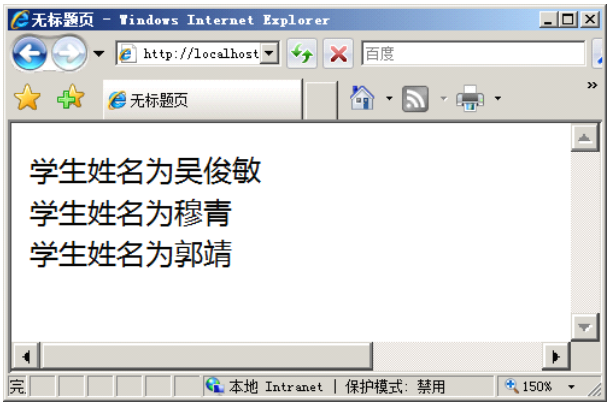

图 21-23　简单查询

查询的原理很简单，在 21.1.1 中就已经讲解了如何创建 LINQ 的 Web 应用，但是那个时候并没有涉及到 LINQ 查询子句，现在回过头再看就会发现其实使用 LINQ 进行数据库访问也并不困难，这里不再作过多解释。

21.5.2　建立连接

上一节中讲解了使用 LINQ 快速的建立数据库之间的连接。在 LINQ to SQL 中，.NET Framework 同样像 ADO.NET 一样为 LINQ 提供了 LINQ 数据库连接类和枚举用于自定义数据连接。建立与 SQL 数据库的连接，就需要使用 DataContext 类，示例代码如下所示。

```
        DataContext db = new DataContext("Data Source=(local);
        Initial Catalog=student;Persist Security Info=True;User ID=sa;Password=sa");//建立连接
```

上述代码通过 DataContext 类进行数据连接。当数据库连接后，就可以获取数据库相应的表显示数据，示例代码如下所示。

```
    protected void Page_Load(object sender, EventArgs e)
    {
        DataContext db = new DataContext("Data Source=(local);
        Initial Catalog=student;Persist Security Info=True;User ID=sa;Password=sa");//建立连接
        try
        {
            Table<Student> stu = db.GetTable<Student>();                          //获取相应表的数据
            var StudentList = from d in stu orderby d.S_ID descending select d;   //执行 LINQ 查询
            foreach (var stud in StudentList)                                     //遍历集合
            {
                Response.Write("学生姓名为" + stud.S_NAME.ToString() + "<br/>");//输出对象
            }
        }
        catch
        {
```

```
                Response.Write("数据库连接失败");                                    //抛出异常
            }
        }
```

上述代码使用 DataContext 类进行了数据库连接的建立，建立连接后可以使用 Table 类获取数据库中的表并填充数据到表里面，这样就无需像 ADO.NET 一样首先建立连接、然后再填充数据集这样进行繁冗的数据操作。开发人员可以直接使用 LINQ 查询语句对数据进行筛选。

21.5.3 插入数据

创建了 DataContext 类对象之后，就能够使用 DataContext 的方法进行数据插入、更新和删除操作。相比 ADO.NET，使用 DataContext 对象进行数据库操作更加方便和简单。使用 LINQ to SQL 类进行数据插入的操作步骤如下。

- ❑ 创建一个包含要提交的列数据的新对象。
- ❑ 将这个新对象添加到与数据库中的目标表关联的 LINQ to SQL Table 集合。
- ❑ 将更改提交到数据库。

上面三个步骤就能够实现数据的插入操作，对数据库的连接可以使用 LINQ to SQL 类文件或者自己创建连接字串。示例代码如下所示。

```
public void InsertSQL()
{
    Student stu = new Student { S_NAME="xixi",C_ID=1,S_CLASS="0502" };    //创建一个数据对象
    MyLinqDataContext dc = new MyLinqDataContext();                       //创建一个数据连接
    dc.Student.InsertOnSubmit(stu);                                       //执行插入数据操作
    dc.SubmitChanges();                                                   //执行更新操作
}
```

上述代码使用了前面创建的 LINQ to SQL 类文件 MyLinq.dbml，使用该类文件快速的创建一个连接 。在 LINQ 中，LINQ 模型将关系型数据库模型转换成一种面向对象的编程模型，开发人员可以创建一个数据对象并为数据对象中的字段赋值，再通过 LINQ to SQL 类执行 InsertOnsubmit 方法进行数据插入就可以完成数据插入，运行后如图 21-24 所示。

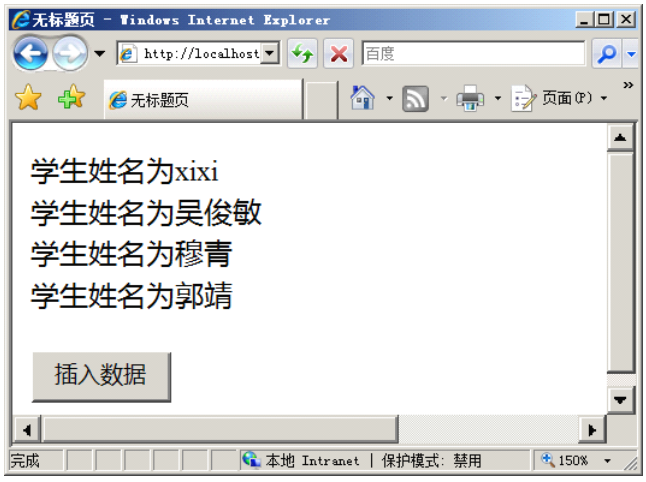

图 21-24 插入数据

使用 LINQ 进行数据插入比 ADO.NET 操作数据库使用的代码更少，而其思想更贴近了面向对象的概念。

21.5.4 修改数据

LINQ 对数据库的修改也是非常的简便的，执行数据库中数据的更新的基本步骤如下所示。

- ❑ 查询数据库中要更新的行。
- ❑ 对得到的 LINQ to SQL 对象中的成员值进行所需的更改。
- ❑ 将更改提交到数据库。

上面三个步骤就能够实现数据的修改更新，示例代码如下所示。

```
public void UpdateSQL()
{
    MyLinqDataContext dc = new MyLinqDataContext();
    var element = from d in dc.Student where d.S_ID == 4 select d;          //查询
    foreach (var e in element)                                              //遍历集合
    {
        e.S_NAME = "xixi2";                                                 //修改值
        e.S_CLASS = "0501";                                                 //修改值
    }
    dc.SubmitChanges();                                                     //更新
}
```

在修改数据库中一条数据之前，必须要查询出这个数据。查询可以使用 LINQ 查询语句和 where 子句进行筛选查询，也可以使用 Where 方法进行筛选查询。筛选查询出数据之后，就能够修改相应的的值并使用 SunmitChanges()方法进行数据更新，运行后如图 21-25 所示。

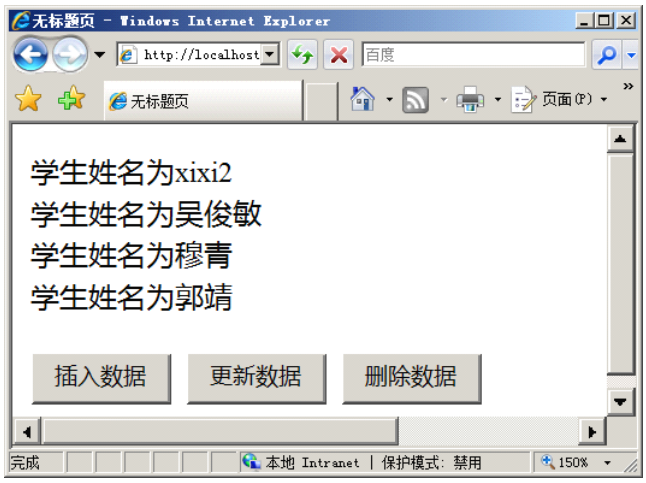

图 21-25 更新 xixi 为 xixi2

21.5.5 删除数据

使用 LINQ 能够快速的删除行，删除行的基本步骤如下所示。

- ❑ 在数据库的外键约束中设置 ON DELETE CASCADE 规则。
- ❑ 使用自己的代码首先删除阻止删除父对象的子对象。

只需要上面两个步骤就能够实现数据的删除，示例代码如下所示。

```
public void DeleteSQL()
{
    MyLinqDataContext dc = new MyLinqDataContext();                         //连接数据源
```

```
        var del = from d in dc.Student where d.S_ID == 4 select d;          //查询要删除的行
        foreach (var e in del)                                              //遍历集合
        {
            dc.Student.DeleteOnSubmit(e);                                   //执行删除操作
        }
    }
```

上述代码使用 LINQ 执行了数据库删除操作，运行后如图 21-26 所示。

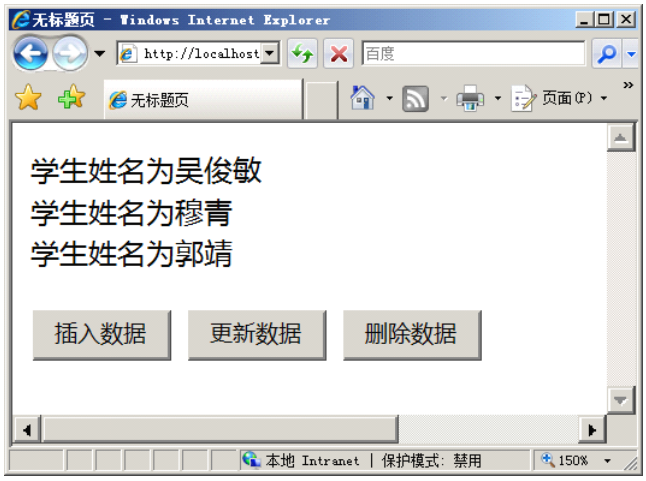

图 21-26　删除数据

在进行数据中表的删除过程时，有些情况需要判断数据库中表与表之间是否包含约束关系，如果包含了子项，首先必须删除子项否则不能删除父项。例如在删除 Class 表时，在 Student 表中有很多项都包含 Class 表的元素，例如 C_ID 等于 1 的元素，当要删除 Class 表中 C_ID 为 1 的元素时，就需要先删除 Student 表中包含 C_ID 为 1 的元素，以保持数据库约束，示例代码如下所示。

```
    public void DeleteSQL()
    {
        MyLinqDataContext dc = new MyLinqDataContext();
        var delf = from d in dc.Class where d.C_ID == 1 select d;           //查询父表
        var del = from d in dc.Student from f in dc.Class where d.S_ID ==
        4 && f.C_ID==1&&d.S_ID==f.C_ID select d;                            //进行约束查询
        foreach (var e in del)                                              //删除子表
        {
            dc.Student.DeleteOnSubmit(e);                                   //删除对象
            dc.SubmitChanges();                                             //更新删除
        }
        foreach (var f in delf)                                             //删除父表
        {
            dc.Class.DeleteOnSubmit(f);                                     //删除对象
            dc.SubmitChanges();                                             //更新删除
        }
    }
```

当数据库包含外键，以及其他约束条件时，在执行删除操作时必须小心进行，否则会破坏数据库约束，也有可能抛出异常。

21.6 LINQ 与 MVC

在 ASP.NET MVC 应用程序中，Models 层通常用于抽象数据库中的表使之成为开发人员能够方便操作的对象，在 Models 层中，开发人员能够使用 LINQ 进行数据库的抽象并通过 LINQ 筛选和查询数据库中的数据用于页面呈现。

21.6.1 创建 ASP.NET MVC 应用程序

在前面的章节中讲到了 ASP.NET MVC 开发模型，在 ASP.NET MVC 应用程序中，开发人员能够很好的将页面进行分离，这样不同的开发人员就能够只关注自身的开发而无需进行页面整合。在 ASP.NET MVC 应用程序中，包括三个基本的模块，这三个模块分别是 Models、Controllers 和 Views。

Controllers 用于实现与 Models 的交互和 Views 的交互，在 Controllers 与 Models 交互时，Controllers 主要是用于从 Models 中进行数据的获取，而 Models 主要关注与数据库进行交互，在 Controllers 与 Views 交互时，Controllers 中的方法同 Views 中的页面一起用于页面呈现。在进行 ASP.NET MVC 应用程序的开发时，在应用程序中读取数据库则需要在 Models 中创建 LINQ 文件与数据进行交互，在创建 LINQ 文件时，首先需要创建 ASP.NET MVC 应用程序，如图 21-27 所示。

单击【确定】按钮创建 ASP.NET MVC 应用程序，系统会默认创建若干文件和文件夹，删除 Views 和 Controllers 文件夹下的文件，自行创建页面进行 ASP.NET MVC 应用程序的开发。首先创建 Views 文件，在创建 Views 文件时首先需要创建一个文件夹，这里创建一个 Blog 文件夹，创建后在该文件夹内创建 Views 文件 Index.aspx，如图 21-28 所示。

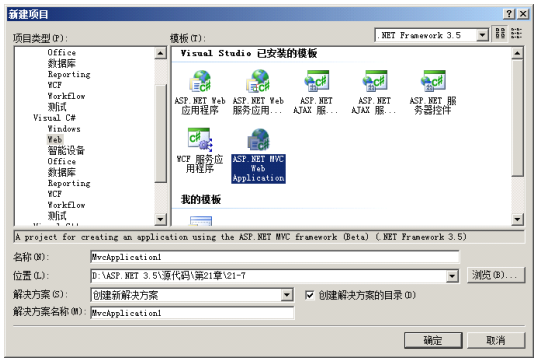

图 21-27 创建 ASP.NET MVC 应用程序

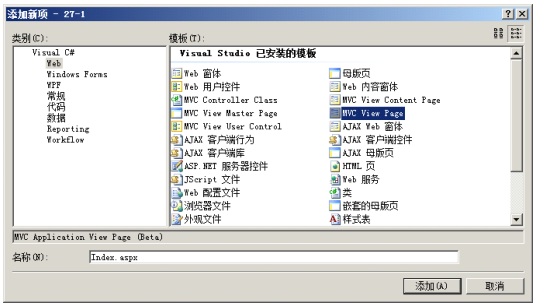

图 21-28 创建 Views 文件

这里创建一个 Index.aspx 的 Views 文件用于页面呈现，Index.aspx 页面代码如下所示。

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title><%=ViewData["title"]%></title>
</head>
<body>
    <div>
        数据库中的数据为:<br/>
        <%=ViewData["contents"]%>
    </div>
</body>
</html>
```

上述代码在 Index.aspx 中使用了两个 ViewData，这两个 ViewData 分别用于呈现标题和数据内容。在 Views 文件中，需要通过 Controllers 文件进行 ViewData 变量的获取，这里还需要创建一个 Controllers 文件。创建该文件时，应该与相应 Views 页面的文件夹同名，并在名称后加入 Controllers.cs，如图 21-29 所示。

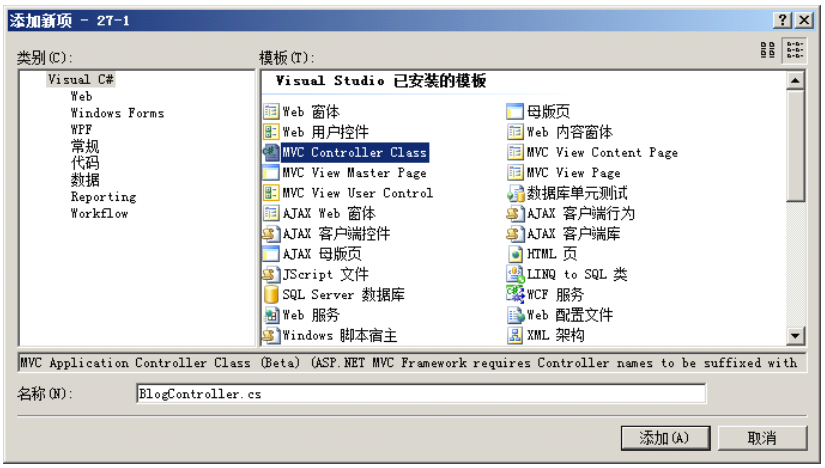

图 21-29　创建 Controllers 文件

由于创建的文件夹是 Blog，所以 Controllers 文件的名称应该为 BlogControllers.cs，创建完成后，为了让用户能够访问 Index.aspx，还需要实现 Index 方法，示例代码如下所示。

```
namespace _27_1.Controllers
{
    public class BlogController : Controller                                    //继承自 Controller
    {
        public ActionResult Index()                                             //实现方法
        {
            ViewData["title"] = "MVC  和  LINQ";                                // ViewData["title"]
            ViewData["contents"] = "数据内容";                                  //ViewData["contents"]
            return View();                                                      //返回视图
        }
    }
}
```

上述代码分别为 ASP.NET MVC 应用程序添加了两个 ViewData，这两个 ViewData 分别用于呈现标题和数据内容，在 Views 相应的文件中能够使用这两个 ViewData 进行数据呈现。

21.6.2　创建 LINQ to SQL

在创建了 ASP.NET MVC 应用程序后并创建了相应的 Views 和 Controllers 文件用于页面呈现和数据获取，在 Controllers 中 ViewData[“content”]是获取数据库中的数据，这里可以在 Models 中创建 LINQ to SQL 类进行数据辅助操作，如图 21-30 所示。

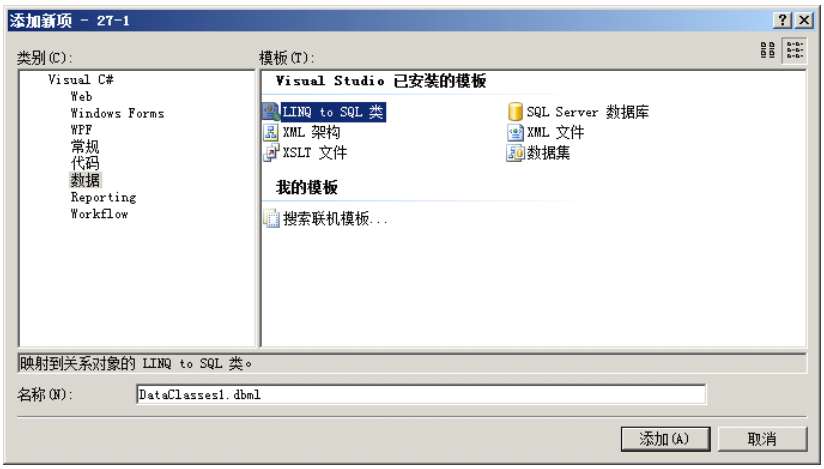

图 21-30　创建 LINQ to SQL 类

创建完成后就能够在服务器资源管理器中拖动相应的表进行 LINQ to SQL 类的创建，如图 21-31 和图 21-32 所示。

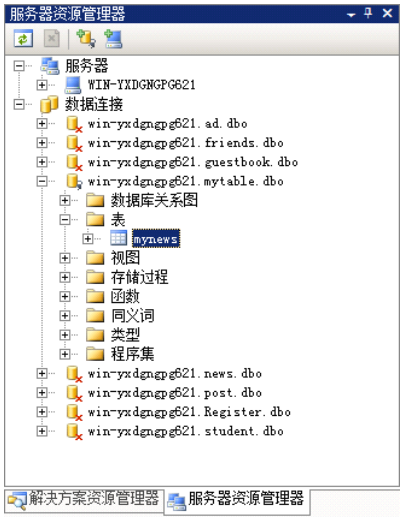

图 21-31　服务器资源管理器

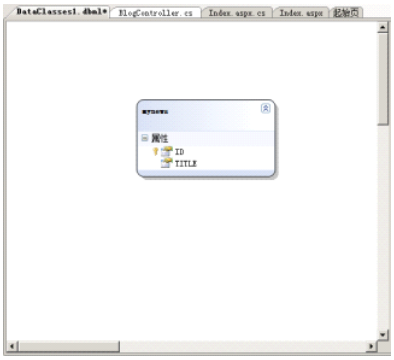

图 21-32　LINQ to SQL 类文件

添加完成并配置 LINQ to SQL 类后，还需要在 Models 中创建相应的类文件，示例代码如下所示。

```
namespace _27_1.Models
{
    public class GetData
    {
        public string build()                                                    //实现方法
        {
            string build="";                                                     //返回字符串
            DataClasses1DataContext dcd = new DataClasses1DataContext();         //使用 LINQ 类
            var d = from dc in dcd.mynews select dc;                             //执行查询
            foreach (var myd in d)                                               //遍历集合
            {
                build += myd.TITLE.ToString()+"<br/>";                           //输出字符串
            }
            return build;
```

```
        }
    }
}
```

上述类文件在 Models 中进行数据查询并返回相应的数据，在 Controllers 中，开发人员可以使用该类进行数据查询和操作。

注意：在 Controllers 中需要使用 Models 命名空间，这也就能够直接在 Controllers 中进行 LINQ 数据查询和操作，但是为了层次分明和简便，推荐在 Models 中进行相应的数据操作类文件编写。

21.6.3 数据查询

在创建了相应的数据操作类后，就能够在 Controllers 中查询数据并将数据呈现在页面中，Controllers 中 Index 方法需要从数据库中进行数据读取，示例代码如下所示。

```
using _27_1.Models;                                                    //使用 Model 命名空间
namespace _27_1.Controllers
{
    public class BlogController : Controller                           //继承自 Controller
    {
        public ActionResult Index()                                    //实现方法
        {
            ViewData["title"] = "MVC 和 LINQ";                         //定义 ViewData
            GetData da=new GetData();                                  //创建对象
            ViewData["contents"] =da.build();                          //赋值给 ViewData
            return View();                                             //返回默认视图
        }
    }
}
```

上述代码则使用了 Models 中的类进行数据呈现，在创建对象后，可以使用对象中的 build 方法进行数据获取。在运行前，还需要修改 URL Routing 的默认值进行默认页面的呈现，示例代码如下所示。

```
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",                          //编写路由规则
                new { controller = "Blog", action = "Index", id = "" } //编写默认值
            );
        }
```

上述代码编写了路由规则和默认值，当用户访问网站时，如果 Controllers 和方法都没有指定，则会访问 Blog/Index 方法。修改路由规则和默认值后，就能够运行 ASP.NET MVC 应用程序，运行后如图 21-33 所示。

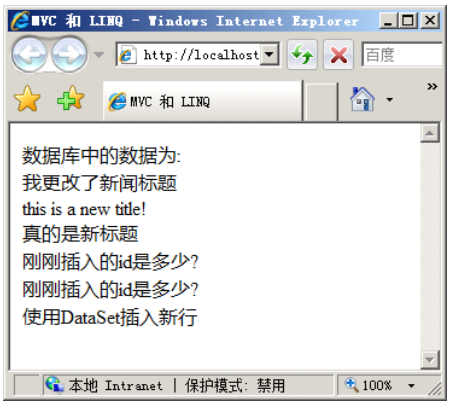

图 21-33　运行 MVC 应用程序

21.7　小结

LINQ 是.NET 3.5 框架里的新特性，使用 LINQ 能够极大的方便开发人员进行数据操作。不仅如此，LINQ 还支持多种数据源中数据的筛选和查询，这些数据源可能是数组、数据库、数据集甚至是 XML 文档。本章着重的讲解了 LINQ 查询语法，以及 LINQ 查询子句，可以由浅入深的了解 LINQ 查询语句是如何编写的。LINQ 查询语句的语法非常简单，熟悉 SQL 查询语法的人在一定程度上很容易就能够上手投入到开发中。本章还包括：

- 准备数据源：准备了 3 种类型的数据源以演示如何使用 LINQ 进行多种数据源查询。
- 基本子句：讲解了 from、where、let、join 等基本子句，基本子句在 LINQ 中是非常基础也是非常重要的，熟练的编写基本子句不仅能够提高性能也能够方便筛选。
- 投影操作：讲解了如何使用 LINQ 提供的 Select 方法进行投影操作。
- 排序操作：讲解了如何使用 LINQ 提供的 Where 方法进行排序操作。
- 聚合操作：讲解了如何使用 LINQ 提供的 Sum、Count 等方法进行聚合操作。
- 建立连接：讲解了如何不使用 LINQ to SQL 提供的类而使用方法建立与 SQL 数据源的连接。
- 数据操作：讲解了如何使用 LINQ 进行数据插入、修改和删除等操作。

本章在最后几节中详细的讲解了如何使用 LINQ 进行数据插入、修改和删除等操作以及对比了 LINQ 与 ADO.NET 的优劣，使用 LINQ 能够减少数据操作的代码量，使代码更像是使用面向对象的思想进行开发的，并再与 ASP.NET MVC 应用程序进行应用整合。LINQ 技术现在在国内是一门新技术，但是发展也有一定的时间了，熟练掌握 LINQ 基础能够在未来的开发潮流中占有一席之地。