第 17 章　LINQ 开发技术

语言集成查询（LINQ）是.NET Framework 3.5 中一项具有突破性的创新，它在对象和数据之间架起了一座桥梁。LINQ 会成为开发代码的重要组成部分。在编写代码时，LINQ 可以提供语法检查、丰富的元数据、智能感知、静态类型等强类型语言的优点。LINQ 还可以方便地对内存中的数据进行查询，而不仅仅是查询外部数据。

本章旨在让读者掌握 LINQ 的两项主要相关技术：LINQ 到 SQL 和 LINQ 到 XML。在本章读者还将学习查询运算符和查询表达式的作用，这些技术允许定义语句来对数据源进行查询，获取所请求的结果集。与此同时，读者可以动手创建许多 LINQ 案例，学习与集合类型、关系数据库和 XML 文档里的数据进行交互的方式。

【本章示例参考：\源代码\C17\LinqSample】

17.1　定义 LINQ 的作用

为什么要使用 LINQ 技术呢？虽然许多功能可以直接按传统方式实现，但它们的益处只有在 LINQ 的场景中才会更加清楚地展现出来。

一个完整的项目永远离不开各种数据，程序员需要获取、存储数据，或编写代码来访问、修改数据，或设计网页来采集、汇总数据。程序中处理的数据来源主要有两方面：一是包含在关系数据库中的信息；另一个是很流行的数据格式 XML 文档，如*.config 文件。

除了以上两个常见的数据源外，还能在很多地方用到数据。例如，有一个 List<>泛型，内含 100 个整型数据。又如创建了一个数组，建立一个表达式从数组中查询满足条件的数据元素。显而易见，数据在程序中处处遇到。作为程序员如何与各种各样的数据打交道呢？在.NET 2.0 中，以特定命名空间的特定类型来操作各种数据源。主要的数据源及其处理方式如表 17-1 所示。

表 17-1　操作各种数据的方式

需要的数据	数据的处理方式
对象集合	System.Array 命名空间和 System.Collections/System.Collections.Generics 命名空间
数据元素	System.Reflection 命名空间
XML 文档数据	System.Xml.dll
关系数据	System.Data.dll

当然，这些操作数据的方法读者已经非常熟悉，基本问题在于，这些应用程序接口（API）提供很少的集成方式。虽然可以把 ADO.NET DataSet 保存成 XML，然后通过 System.Xml 命名

空间来操作，可是所需的操作要通过转换才可以实现，实际的问题就是数据操作相当不对称。

LINQ 的目的是提供一种统一且对等的方式，让程序员在广义的数据上操作“数据”。通过使用 LINQ，能够在编程语言内直接创建查询表达式的实体。这些查询表达式是基于许多查询运算符的，而且是有意设计成类似 SQL 表达式的样子。LINQ 允许查询表达式以统一的方式来操作任何实现了 IEnumerable<T>接口的对象、关系数据库或 XML 文档。

严格地说，LINQ 是用来描述数据访问总体方式的术语。LINQ 到 SQL（LINQ to SQL）是针对关系数据库的 LINQ；LINQ 到 XML（LINQ to XML）是针对 XML 文档的 LINQ。

说明：LINQ 查询表达式跟传统的 SQL 语句不同，它是强类型的，所以 C＃编译器会让编程者保证这些表达式在语法上是合法的。

17.2　核心 LINQ 程序集

本节通过一个基于 Web 的 LINQ 应用程序，来讲述 LINQ 程序集，主要介绍程序集如何引用，以及各个程序集下有哪些主要的命名空间。下面通过创建使用 LINQ 的 Web 应用程序，来看看 LINQ 程序集是如何添加引用的。操作步骤如下。

（1）新建一个网站，打开“新建网站”窗口，如图 17-1 所示。在“模板”选项组中选择“ASP.NET 网站”选项，并在“模板”下拉列表框中选择“.NET Framework 3.5”选项。

注意：由于 LINQ 只被.NET Framework 3.5 所支持，因此若要创建使用 LINQ 的 Web 应用程序（或 Windows From 应用程序），必须使用.NET Framework 3.5。

（2）单击“确定”按钮，创建 LinqSample 应用程序。在“解决方案资源管理器”面板中查看当前应用程序，如图 17-2 所示。

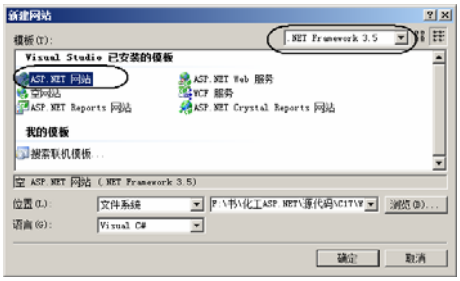

图 17-1　“新建网站”窗口

图 17-2　查看应用程序

（3）双击“解决方案资源管理器”面板中的 Default.aspx 分支后，再双击 Default.aspx.cs 项可查看其功能代码。在新建的一个 Web 应用程序的模板中，可看到一些自动添加的程序集合，如代码 17-01 所示。

代码 17-01　一个 Web 程序模板

```
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
```

```
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    { }
}
```

从上面的代码中可以看到，项目模板为命名空间添加了许多 using 指令。如 using System.Linq 和 using System.Xml.Linq 等这些命名空间属于哪个程序集？各个命名空间下又有哪些主要的类和对象？核心的 LINQ 程序集如表 17-2 所示。

表 17-2　核心的 LINQ 程序集

程序集	内容描述
System.Core.dll	该程序集包含 System.Linq 和 System.Linq.Expressions 命名空间等，其中 System.Linq 包含用于 LINQ 查询的基础结构中的标准查询运算符、类型和接口的集合
System.Xml.Linq.dll	该程序集定义了 3 个命名空间 System.Xml.Linq、System.Xml.Schema 和 System.Xml.Xpath，其中最核心的是 System.Xml.Linq 命名空间
System.Query.dll	定义了代表核心 LINQ API 的类型
System.Data.Linq.dll	包括在那些使用 AllowPartiallyTrustedCallersAttribute 属性标记的.NET Framework 程序集之中

（4）应用程序在配置文件 web.config 中，配置了与 LINQ 相关的程序集 System.Xml.Linq 和 System.Data.DataSetExtensions。这些配置保存在配置文件 web.config 的<compilation>元素的<assemblies>子元素下，程序代码如代码 17-02 所示。

代码 17-02　配置文件 web.config 文件

```
<compilation debug="true">
        <assemblies>
        ……
        <!--配置 System.Core 程序集-->
            <add assembly="System.Core, Version=3.17.0.0, Culture=neutral,
PublicKeyToken=B77A5C561934E089"/>
        <!--配置 System.Web.Extensions 程序集-->
            <add assembly="System.Web.Extensions, Version=3.17.0.0, Culture=neutral,
PublicKeyToken=31BF3856AD364E35"/>
        <!--配置 Sytem.Data.DataSetExtensions 程序集-->
            <add assembly="System.Data.DataSetExtensions, Version=3.17.0.0, Culture=neutral,
PublicKeyToken=B77A5C561934E089"/>
        <!--配置 System.Xml.Linq 程序集-->
            <add assembly="System.Xml.Linq, Version=3.17.0.0, Culture=neutral,
PublicKeyToken=B77A5C561934E089"/>
        </assemblies>
  </compilation>
```

17.3　LINQ 查询表达式初览

为了研究 LINQ 编程模型，先建立一些简单的表达式来操作包含在各种数组里的数据，以体验一下 LINQ 表达式的一些特性。要创建一个 LINQ 查询操作，一般需要以下步骤。

（1）准备数据源。准备 LINQ 查询操作所需要的数据源，如需要查询的集合、关系数据库、XML 文档等。

（2）创建查询。创建查询数据的 LINQ 查询表达式，如“from u in artics select u;”等。

（3）执行查询。执行上述步骤创建的 LINQ 查询表达式，并获取相应的结果。

说明：LINQ 包括 LINQ to Objects、LINQ to SQL、LINQ to DataSet 和 LINQ to XML 4 个组件，它们分别能够查询和处理集合类型、关系数据库类型、DataSet 对象类型和 XML 类型数据。

本节主要介绍 LINQ 查询操作中 3 个步骤的具体实现方法，并进一步学习如何创建查询集合、关系数据库、DataSet 对象和 XML 的 LINQ 查询表达式。

17.3.1　创建查询集合的 LINQ 表达式

首先准备集合类型的数据源，下面的实例代码创建了两个集合类型的数据源，一个为整型数据 dataSource，另一个为泛型列表 students（元素的类型为 StudentBaseInfo）。StudentBaseInfo 类包括 id、name、age、email 和 aliasName5 个属性，其详细信息视图如图 17-3 所示。

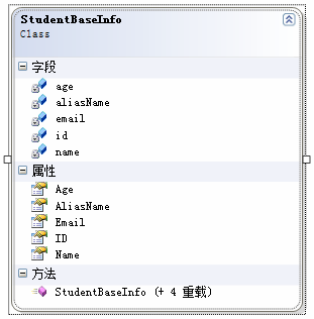

图 17-3　StudentBaseInfo 类的类视图

创建 StudentBaseInfo 类的方法如代码 17-03 所示。

代码 17-03　StudentBaseInfo 类的定义

```
using System.Collections.Generic;
/// <summary>
///StudentInfo 的摘要说明
/// </summary>
/// <summary>
///学生的的基本信息
/// </summary>
public class StudentBaseInfo
{
```

```
private int id;                          //学号
private string name;                     //姓名
private List<string> aliasName;          //别名
private int age;                         //年龄
private string email;                    //电子邮件
/// <summary>
///学生的学号
/// </summary>
public int ID
{
    get { return id; }
    set { id = value; }
}
/// <summary>
///学生的姓名
/// </summary>
public string Name
{
    get { return name; }
    set { name = value; }
}
/// <summary>
///学生的别名
/// </summary>
public List<string> AliasName
{
    get { return aliasName; }
    set { aliasName = value; }
}
/// <summary>
///学生的年龄
/// </summary>
public int Age
{
    get { return age; }
    set { age = value; }
}
///<summary>
///学生的电子邮件
/// </summary>
public string Email
{
    get { return email; }
    set { email = value; }
}
public StudentBaseInfo( )
{
    ///
}
/// <summary>
///构造函数，初始化学生的学号、姓名和电子邮件
/// </summary>
/// <param name="id"></param>
/// <param name="name"></param>
/// <param name="email"></param>
public StudentBaseInfo(int id, string name, string email)
{
```

```
            this.id = id;
            this.name = name;
            this.email = email;
        }
        /// <summary>
        ///构造函数，初始化学生的 ID 值、名称、电子邮件和年龄
        /// </summary>
        /// <param name="id">id </param>参数学号
        /// <param name="name">name</param>参数姓名
        /// <param name="email">email</param>参数电子邮件
        /// <param name="age">age</param>年龄
        public StudentBaseInfo(int id, string name, string email, int age)
        {
            this.id = id;
            this.name = name;
            this.email = email;
            this.age = age;
        }

        /// <summary>
        ///构造函数，初始化学生的学号、姓名、电子邮件和别名
        /// </summary>
        /// <param name="id">id</param>学号
        /// <param name="name">name</param>姓名
        /// <param name="email">email</param>电子邮件
        /// <param name="aliasName">aliasName</param>别名
        public StudentBaseInfo(int id, string name, string email, List<string> aliasName)
        {
            this.id = id;
            this.name = name;
            this.email = email;
            this.aliasName = aliasName;
        }
        /// <summary>
        ///构造函数，初始化学生的学号、姓名、电子邮件、年龄和别名
        /// </summary>
        /// <param name="id">id</param>学号
        /// <param name="name">name</param>姓名
        /// <param name="email">email</param>电子邮件
        /// <param name="aliasName">aliasName</param>别名
        /// <parm name="age">age</parm>年龄
        public StudentBaseInfo(int id, string name, string email, int age, List<string> aliasName)
        {
            this.id = id;
            this.name = name;
            this.email = email;
            this.age = age;
            this.aliasName = aliasName;
        }
    }
```

整型数组 DataSource 的长度为 100，元素值的范围是 0～99。泛型列表 Students 包含 9 个元素，每个元素的 ID 属性的范围是 1～9。如代码 17-04 所示，将这个方法添加到 Default.aspx 中，并在 Page_Load 事件中调用这个方法。

代码 17-04　准备集合类型的数据源

```
private void ReadyCollectionData( )
{
        //准备数据源，创建一个整型数组
        int[ ] dataSource = new int[100];
        for (int i = 0; i < 100; i++)
        {
            dataSource[i] = i;
        }
        //准备数据源，创建一个泛型列表，元素类型为 StudentInfo
        List<StudentBaseInfo> students = new List<StudentBaseInfo>();
        for (int i = 1; i < 10; i++)
        {
            students.Add(new StudentBaseInfo(i, "Student0" + i.ToString( ),"Student0" +
i.ToString( ) + "@web.com"));
        }
        //以下为 LINQ 查询表达式
    }
```

其次，在创建 ReadyCollectionData( )函数准备的集合类型的数据源之后，创建一个 LINQ 查询表达式。该查询表达式查询 ID 属性的值大于 3 的元素。创建集合类型的查询表达式的代码如代码 17-05 所示。

代码 17-05　创建结合类型的查询表达式

```
//以下为 LINQ 查询表达式
        var result = from s in students
                     where s.ID > 3
                     select s;
        foreach (var v in result)
            Response.Write(v.Name+"</br>");//输出查询内容
```

最后执行 LINQ 查询，浏览效果如图 17-4 所示。

一般情况下，查询变量本身不会保存查询结果，而是存储查询的命令。执行 LINQ 查询存在两种方式，即延迟执行和立即执行。

- 延迟执行。查询表达式不是在其被创建时执行，而是当需要访问该查询的结果时，才执行该查询表达式。
- 立即查询。查询表达式将在其被创建时就立即执行，如包含聚合查询（如 COUNT、MAX、MIN、SUM、Average 操作等）。

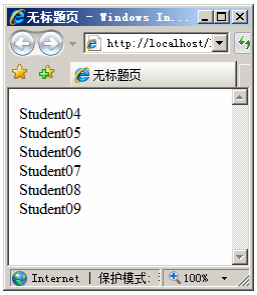

图 17-4　浏览查询效果

说明：包含 COUNT、MAX、MIN、SUM、Average 操作等将在后面章节中介绍。

17.3.2　创建查询 DataSet 的查询表达式

首先，创建一个 DataSet 类型的数据源 ds。该数据源使用 SQL 语句“Select * from StudentInfo”从 LinqDB 数据库的 StudentInfo 表中获取数据。获取的数据填充到 DataSet 中的表 StudentInfo 中。准备 DataSet 类型数据源的代码如代码 17-06 所示。

代码 17-06　准备 DataSet 类型数据源：userDateSet.aspx

```
using System.Data. SqlClient;
private void ReadyDataSetData( )
    {    ///准备数据源，创建 DataSet 类型的数据源
         ///创建连接
         SqlConnection con = new SqlConnection(ConfigurationManager.ConnectionStrings["Linq
DBConnectionString"].ConnectionString);
         ///创建 SQL 语句
         string cmdText = "SELECT * FROM StudentInfo";
         ///创建执行 SQL 语句的命令
         SqlDataAdapter da = new SqlDataAdapter(cmdText, con);
         ///打开数据库的连接
         con.Open( );
         ///执行查询操作
         DataSet ds = new DataSet( );
         da.Fill(ds, "StudentInfo");
         con.Close( );
         ///以下为 LINQ 查询表达式
...
```

其次，为上面 ReadDataSetData( )函数准备的集合类型的数据源之后，创建一个 LINQ 查询表达式。该查询表达式查询学号 ID 列的值大于 3，且 StudentName 列值的长度为大于 10 的元素。创建 DataSet 类型查询表达式代码如代码 17-07 所示。

代码 17-07　创建 Dataset 类型的查询表达式

```
Private void ReadyDatabaseData( )
{
//以下为 LINQ 查询表达式
         var result = from u in ds.Tables["StudentInfo"].AsEnumerable( )
                      where u.Field<int>("ID")>3    && u.Field<string>("StudentName").Length>10
                       select u;
         foreach (var v in result)
              Response.Write(v);//输出查询内容
}
```

17.3.3　创建 SQL Server 数据库的查询表达式

准备 SQL Server 数据库类型的数据源。为了准备 SQL Server 数据库类型的数据源，首先需要创建一个 SQL Server 数据库。

下面创建一个 LinqDB 数据库，并为该数据库创建一个 DBML 文件（在“添加新项”窗口中，选择添加“LINQ TO SQL 类”）。在 Visual Studio 2008 中查看 LinqDB 数据库的 DBML 文件 LinqDB.dbml，如图 17-5 所示。Linq.dbml 文件只映射了 LinqDB 数据库中的 StudentInfo 表。

图 17-5　LinqDb 数据库的 DMBL 文件

创建一个 SQL Server 数据库类型的数据源——数据上下文的实例 db，并获取 StudentInfo 表的数据，保存在类型为 List <StudentInfo>的 students

集合中。准备 SQL Server 数据库类型的数据源代码如代码 17-08 所示。

代码 17-08　准备 SQL Server 数据库类型的数据源：UseSQL.aspx.cs

```
private void ReaddatabaseData( )
{
      ///准备数据源，创建 LinqDB 数据库的数据上下文类的实例
           LinqDBDataContext db = new LinqDBDataContext(ConfigurationManager.ConnectionStrings
["LinqDBConnectionString"].ConnectionString);
           List<StudentInfo> students = db.StudentInfos.ToList( );
       ///以下为 LINQ 查询表达式
}
```

其次，为 ReadyDataBaseData( )函数准备的集合类型的数据源，创建一个 LINQ 查询表达式。该查询表达式查询学生学号 ID 属性的值大于 3 且小于 10 的元素。创建查询 SQL Server 数据库的代码如代码 17-09 所示。

代码 17-09　创建查询 SQL Server 数据库

```
Private void ReadyDataBaseData( )
{
      ///准备数据源
      ///以下为 LINQ 查询表达式
           var result = from s in students
                              where s.ID > 3 && s.ID < 10
                              select s;
}
```

17.3.4　创建查询 XML 类型的查询表达式

下面实例代码创建 XML 类型的数据源 xmlString。该数据源是一个具有 XML 结构的字符串，包含 Students、Student 等元素。该实例调用 XElement 类将具有 XML 结构的字符串 xmlString 导入内存中，为查询做准备。准备 XML 类型的数据源代码如代码 17-10 所示。

代码 17-10　准备 XML 类型的数据源：UseXML.aspx.cs

```
private void ReadyXmlData()
     {///准备数据源，创建 XML 类型的数据源
          string xmlString =
          "<Books>"
            + "<Book ID=\"101\">"
                + "<No>00001</No>"
                + "<Name>Book 0001</Name>"
                + "<Price>100</Price>"
                + "<Remark>This is a book 00001.</Remark>"
             + "</Book>"
             + "<Book ID=\"102\">"
                + "<No>00002</No>"
                + "<Name>Book 0002</Name>"
                + "<Price>200</Price>"
                + "<Remark>This is a book 00002.</Remark>"
             + "</Book>"
             + "<Book ID=\"103\">"
                + "<No>0006</No>"
                + "<Name>Book 0006</Name>"
```

```
                + "<Price>600</Price>"
                + "<Remark>This is a book 0006.</Remark>"
            + "</Book>"
        + "</Books>";
        ///导入 XML 文件
        XElement xmlDoc = XElement.Parse(xmlString);
        ///以下为 LINQ 查询表达式
    }
```

其次，下面的实例代码为 ReadyXmlData( )函数准备的集合类型的数据源之后，创建一个 LINQ 查询表达式。该查询表达式从 xmlString 字符串中查询包含子元素的值是书名 Book0002 的元素。创建查询 XML 类型的查询表达式代码如代码 17-11 所示。

代码 17-11　创建查询 XML 类型的查询表达式

```
Private void ReadyXmlData( )
{
    ///准备数据源
    ///以下为 LINQ 查询表达式
    ///导入 XML 文件
        XElement xmlDoc = XElement.Parse(xmlString);
        ///以下为 LINQ 查询表达式
        var result = from e in xmlDoc.Elements("Book")
                     where (string)e.Element("Name") == "Book 0002"
                     select e;
        foreach (var v in result)
            //输出查询内容
            Response.Write(v.Value);
}
```

测试上述表达式，浏览效果如图 17-6 所示。

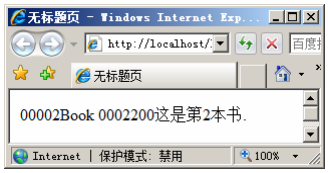

图 17-6　测试效果

17.4　LINQ 查询泛型集合

除了从简单的数据数组里抽出结果以外，LINQ 查询表达式也可以操作 System.Collections.Generic 命名空间的成员类型数据，如 List<T>类型。首先在项目中添加一个名为“UseCreateQuery.aspx”的窗体，这里还是引用基本的 StudentBaseInfo 类。然后在 Page_Load( )过程里定义一个类型为 students 的局部 List<T>变量，使用新的对象初始化语法，给这个表填充几个新的 student 对象。

```
protected void Page_Load(object sender, EventArgs e)
    {
        List<StudentBaseInfo> students = new List<StudentBaseInfo>( );
        students.Add(new StudentBaseInfo(200801,"Student01","Student01@web.com"));
        students.Add(new StudentBaseInfo(200802, "Student02", "Student02@web.com"));
        students.Add(new StudentBaseInfo(200803, "Student03", "Student03@web.com"));
        students.Add(new StudentBaseInfo(200804, "Student04", "Student04@web.com"));
        students.Add(new StudentBaseInfo(200805, "Student05", "Student05@web.com"));
        students.Add(new StudentBaseInfo(200806, "Student05", "Student06@web.com"));
}
```

17.4.1 定义 LINQ 查询

下面建立一个查询表达式，从 students 列表中挑选学号大于 200803 的项。在得到这个子集后将输出每个 student 对象的名字。完整的后台功能代码如代码 17-12 所示。

代码 17-12 建立查询表达式

```
...
using System.Collections.Generic;
public partial class UseGeneric : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //构建数据源
        List<StudentBaseInfo> students = new List<StudentBaseInfo>( );
        students.Add(new StudentBaseInfo(200801,"Student01","Student01@web.com"));
        students.Add(new StudentBaseInfo(200802, "Student02", "Student02@web.com"));
        students.Add(new StudentBaseInfo(200803, "Student03", "Student03@web.com"));
        students.Add(new StudentBaseInfo(200804, "Student04", "Student04@web.com"));
        students.Add(new StudentBaseInfo(200805, "Student05", "Student05@web.com"));
        students.Add(new StudentBaseInfo(200806, "Student05", "Student06@web.com"));
        //LINQ 查询表达式
        var LargeID = from s in students
                      where s.ID >200803
                      select s;
        foreach (var s in LargeID)
            //输出查询结果
            Response.Write(s.Name+"</br>");
    }
}
```

图 17-7 浏览效果

查询表达式只从 List<>里选取学号 ID 属性大于 200803 的项。运行程序，会发现只有 3 项符合搜索标准。查询结果如图 17-7 所示。

注意：如果要建立一个复杂些的查询，读者也许会想只找到那些学号大于“200803”且姓名为“小王”的学生，这样，只要使用 C＃的“&&”运算符创建一个复杂的布尔语句即可。

17.4.2 重访匿名类型

C# 3.0 现在支持创建匿名类型。这个架构允许在运行时定义一个对象的整个结构（或构形），而不用预定义一个强类型的类（或结构）。在上面的例子中，学生类只不过代表了 5 个数据点（属性）的集合而已，没有由其他类成员（方法、事件、覆盖的虚拟成员等）来代表什么特别的功能。

据此，进一步简化当前的例子，去掉整个 StudentBaseInfo 类定义，而选择使用一个匿名类型，且不必使用强类型的 List<>，只用一个隐形数组就可以了，从而可以将 Page_Load 更

新为如代码 17-13 所示。

代码 17-13　创建一个匿名类

```
...
using System.Collections.Generic;
public partial class UseGeneric : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //使用对象初始化器构造一个匿名类型
        var students = new[ ]{
        new StudentBaseInfo(200801,"Student01","Student01@web.com"),
        new StudentBaseInfo(200802, "Student02", "Student02@web.com"),
        new StudentBaseInfo(200803, "Student03", "Student03@web.com"),
        new StudentBaseInfo(200804, "Student04", "Student04@web.com"),
        new StudentBaseInfo(200805, "Student05", "Student05@web.com"),
        new StudentBaseInfo(200806, "Student05", "Student06@web.com")
         };
           //建立查询表达式
        var LargeID = from s in students
                      where s.ID >200803
                      select s;
          foreach (var s in LargeID)
          //输出查询结果
          Response.Write(s.Name+"</br>");
    }
}
```

测试查询，输出结果如图 17-8 所示。尽管输出是完全一样的，但上述代码表示成了高度函数式语法的形式。

图 17-8　匿名类输出结果

17.5　LINQ 查询非泛型集合

LINQ 查询运算符的设计用于实现 IEnumerable<T>接口的类型，无论是直接还是间接的都可以通过扩展的方法实现。鉴于 System.Array 已经具备了这些所需的基础结构，System.Collections 中传统的非泛型容器类却没有这些结构，开发者可以用泛型 Sequernce.OfType<T>( )方法来对包含在这些非泛型集合里的数据进行迭代操作。

OfType<T>( )方法是 Sequence 中少数几个并不扩展的泛型类型成员中的一个。当对一个实现了 IEnumerable 接口的非泛型性的容器类调用这个成员方法时，只需指定容器中项的类型就可以提取一个兼容于 IEnumerable<T>的对象。添加一个 UseIEnumerable.aspx 窗体，使用 LINQ 查询非泛型集合，如代码 17-14 所示。

代码 17-14　使用 LINQ 查询非泛型集合

```
...
using System.Data.SqlTypes;
using System.Collections.Generic;
public partial class UseIEnumerable : System.Web.UI.Page
```

```
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //这是个非泛型的学生的集合
        ArrayList students = new ArrayList( );
        //向集合中添加对象
        students.Add(new StudentBaseInfo(200801, "Student01", "Student01@web.com"));
        students.Add(new StudentBaseInfo(200802, "Student02", "Student02@web.com"));
        students.Add(new StudentBaseInfo(200803, "Student03", "Student03@web.com"));
        students.Add(new StudentBaseInfo(200804, "Student04", "Student04@web.com"));
        students.Add(new StudentBaseInfo(200805, "Student05", "Student05@web.com"));
        students.Add(new StudentBaseInfo(200806, "Student05", "Student06@web.com"));
        //把 ArrayList 转换成一个兼容 IEnumerable<T>的类型。
        IEnumerable<StudentBaseInfo> student = students.OfType<StudentBaseInfo>( );
        //建立查询表达式
        var LargeID = from s in student
                      where s.ID > 200803
                      select s;
        foreach (var s in LargeID)
            //输出查询结果
            Response.Write(s.Name + "</br>");
    }
}
```

测试查询，输出结果如图 17-9 所示。非泛型类型可包含任何类型的项，因为这些容器类成员的原型是接受 System.Objects 的。以 ArrayList 为例，假定一个 ArrayList 内包含好几项，只有一些是数字。如果要得到只含数字类型的数据的子集，可以使用 OfType<T>( )，因为它会过滤出那些类型不同于迭代操作中所指定类型的元素。

```
//从 ArrayList 提取整数
ArrayList students1 = new ArrayList( );
        students1.AddRange(new object[ ]{10,20,30,false,new StudentBaseInfo( ),"string data"});
        IEnumerable<int> myints=students1.OfType<int>( );
        foreach(int i in myints)
            Response.Write("整数为："+i);
```

测试查询，输出结果如图 17-10 所示。

图 17-9　查询非泛型集合

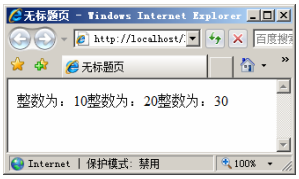

图 17-10　查询结果

17.6　查询运算符的内部表示

通过以上小节的学习，读者可以建立 LINQ 查询表达式，隐形局部变量、匿名类型都可

以应用，对使用查询运算符（如 from、in、where、orderby 和 select）来建立查询表达式的概念也有了初步的了解。C#编译器实际上是把这些标记翻译成了对 System.Query.Sequence 类型各种方法的调用。

注意：实际上，Sequence 的许多方法的原型都是把代理（delegate）作为参数。特别是很多方法都要求一个定义在 System.Query 命名空间的类型为 Func<>的泛型代理作为参数，如下所示。

```
// Sequence.where<T>( )方法的重载版本
Public static IEnumerable<T>Where<T>(IEnumerable<T>source,Func<T,bool>predicate);
Public static IEnumerable<T>Where<T>(IEnumerable<T>source,Func<T,int,bool>predicate)
```

这里有个代理的概念，顾名思义，代表一个接受一串参数和一个返回值的给定函数的模式。假如用 Visual Studio 2005 对象浏览器检查这个类型的话，读者会注意到 Func<>代理可以接受 0 到 4 个输入参数（在这里，分别命名为 A0、A1、A2 和 A3）和一个用 T 来表示的返回类型，如下所示。

```
//Func<>的各种格式
Public delegate T Func<A0,A1,A2,A3,T>(A0 arg( ),A1 arg1,A2 arg2,A3 arg3)
Public delegate T Func <A0,A1,A2,T>(A0 arg( ),A1 arg1,A2 arg2)
Public delegate T Func <A0,A1,T>(A0 arg( ),A1 arg1)
Public delegate T Func <A0>(A0 arg( ))
Public delegate T Func <T>( )
```

鉴于 System.Query.Sequence 的许多成员都要求一个代理作为输入，在调用它们时，读者可以手工创建一个新的代理类型，编写所需的目标方法，并使用 C#的匿名方法，或者也可以定义一个合适的 Lambda 表达式（C# 3.0 提供的新选项）。不管用什么方法，最终结果是完全一样的。

使用查询运算符来建立一个 LINQ 查询表达式是最简单的方法。下面来分析一下这些可行的方法，来看一下 C#查询运算符和内在的 Sequence 类型之间的联系。

17.6.1　用查询运算符建立查询表达式

首先添加一个名为“UseSequence.aspx”的窗体，用查询运算符建立查询表达式，如代码 17-15 所示。

代码 17-15　建立查询表达式

```
┆
using System.Collections.Generic;
public partial class UseSequence : System.Web.UI.Page
{
        protected void Page_Load(object sender, EventArgs e)
    {
        //创建一个数组
        string[ ] cv = { "北京", "呼和浩特", "上海", "乌鲁木齐", "天津", "阿克苏" };
        //使用查询运算符建立查询表达式
        var su = from g in cv
                 where g.Length > 3
                 orderby g
                 select g;
```

```
        //输出查询结果
        foreach(var g in su)
            Response.Write(g+"</br>");
    }
}
```

测试查询，输出结果如图 17-11 所示。

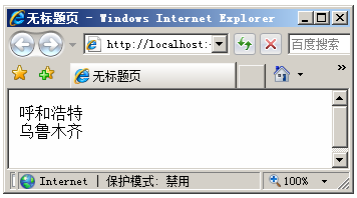

图 17-11　查询结果

使用 C#查询运算符来建立查询表达式的明显好处是，既看不到 Func<>代理，也不需要考虑对 Sequence 类型的调用，因为做此类翻译是 C#编译器的任务。显而易见，使用各种查询运算符（from、in、where、orderby 等）来建立 LINQ 表达式，是最常见和最直截了当的方式。

17.6.2　使用 Sequence 类型和 Lambda 表达式来建立查询表达式

下面使用的是 LINQ 查询运算符，只不过是调用由 Sequence 类型定义各种扩展方法的速记版本而已。在 UseSequence.aspx 中，使用 Sequence 类型和 Lambda 表达式来建立查询表达式，如代码 17-16 所示。

代码 17-16　使用 Sequence 类型定义扩展方法

```
using System.Collections.Generic;
public partial class UseSequence : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

        //用通过 Sequence 类型赋予 Array 的扩展方法建立查询表达式
        var su = cv.Where(ga => cv[0].Length > 3).OrderBy(ga => cv).Select(ga => cv);
        foreach(var ga in su)
            Response.Write(ga+"</br>");
    }
}
```

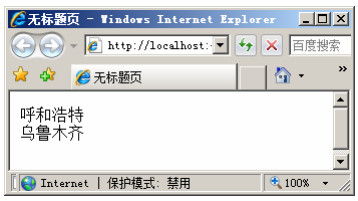

图 17-12　输出结果

测试查询，输出结果如图 17-12 所示。

这里对字符串数组对象调用了泛型 Where<T>( )方法，它是由 Sequence 定义赋予 Array 类型的扩展方法。Sequence.Where<T>( )方法利用了 System.Query.Func<A0,T>代理类型。这个代理的第一个类型参数表示需要处理的、与 IEnumerable<T>兼容的数据，这里是一个字符串数组。第二个类型参数表示将要处理所提到的数据的方法。由于笔者选择了 Lambda 表达式，而不是直接生成一个 Func<T>实例或编造一个匿名方法，所以指定 cv 参数

将由 cv.Length>3 语句来处理。该语句产生一个布尔返回类型。

Where<T>( )方法的返回值被转化成了强类型的 IEnumerable<T>，但在后台操作的是 OrderedSequence 类型。这个结果对象调用了泛型 OrderBy<T,K>( )方法，该方法也需要一个 Func<T,K>代理。最后从指定的 Lambda 表达式的结果选择每个元素，借此再次使用了 Func<T,K>代理。

可以看出，与使用 C#查询运算符相比，直接使用 Sequence 类型的方法来建立一个 LINQ 查询表达式比较麻烦。此外，Sequence 的方法要求代理作为参数，需要经常编写 Lambda 表达式来提供将由底层代理目标处理的输入数据。

17.6.3　使用 Sequence 类型和匿名方法来建立查询表达式

鉴于 C# 3.0 Lambda 表达式只是用于操作匿名方法的简化符号，在 UseSequence.aspx 中，使用 Sequence 类型和匿名方法来建立查询表达式，如代码 17-17 所示。

代码 17-17　使用 Sequence 类型和匿名方法建立查询表达式

```
using System.Collections.Generic;
public partial class UseSequence : System.Web.UI.Page
{
        protected void Page_Load(object sender, EventArgs e)
    {
        //建立一个数组
        string[ ] cv = { "北京", "呼和浩特", "上海", "乌鲁木齐", "天津", "阿克苏" };
        //使用匿名方法建立所需的 Func<>的代理
        Func<string, bool> searchFilter = delegate(string game) { return game.Length > 3; };
        Func<string, string> itPro = delegate(string s) { return s; };
        //把代理传递给 Sequence 的方法
          var sb=cv.Where(searchFilter).OrderBy(itPro).Select(itPro);
        //输出结果
          foreach(var game in sb)
              Response.Write(game+"</br>");
    }
}
```

测试查询，输出结果如图 17-13 所示。

读者可以看到，这个查询表达式的建立更加麻烦，因为手工编写了 Sequence 类型的 Where( )方法、OrderBy( )方法和 Select( )方法用的 Func<>代理。但从好的方面来看，匿名方法确实把所有的处理都包含在单一方法定义中了，然而这个方法在功能上与前面的两种方法完全等同。

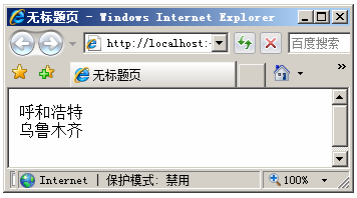

图 17-13　查询结果

17.6.4　使用 Sequence 类型和原始代码建立查询表达式

如果要使用非常繁琐的方式来建立一个查询表达式，可以避免使用 Lambda 或匿名方法

句法，而是直接创建每个 Func<>类型的代理目标。在 UseSequence.aspx 中，使用 Sequence 类型和原始代码建立查询表达式如代码 17-18 所示。

代码 17-18　使用 Sequence 类型和原始代码建立查询表达式

```
using System.Collections.Generic;
public partial class UseSequence : System.Web.UI.Page
{
        protected void Page_Load(object sender, EventArgs e)
    {
            //建立一个数组
            string[ ] cv={ "北京", "呼和浩特", "上海", "乌鲁木齐", "天津", "阿克苏" };
            //使用匿名方法建立所需的 Func<>的代理
            Func<string, bool> searchFilter = new Func<string, bool>(Fiter);
            Func<string, string> itPro = new Func<string, string>(ProcessItem);
            //把代理传递给  Sequence 的方法
            var sb = cv.Where(searchFilter).OrderBy(itPro).Select(itPro);
            //输出结果
               foreach(var game in sb)
                     Response.Write(game + "</br>");
            }
    //代理目标
       public static bool Fiter(string s) { return s.Length > 3; }
       public static string ProcessItem(string s) { return s; }
}
```

测试结果和上面输出相同。综上所述，笔者使用了不同方式实现查询，它们的输出都是一样的。下面列出了关于 LINQ 查询表达式在底层如何表示的要点。

- 查询表达式是通过各种查询运算符建立的。
- 查询运算符仅仅是调用由 System.Query.Sequence 定义的扩展方法的简化符号。
- Sequence 的许多方法要使用代理（特别是 Func<>）作为参数。
- 在 C# 3.0 中，任何要求代理参数的方法都可以传递给一个 Lambda 表达式。
- Lambda 表达式是伪装的匿名方法。
- 匿名方法是对指派一个原始代理，然后手工建立一个代理目标方法的简化符号。

注意：这也许深入了一点，但笔者希望这个讨论能帮助读者理解运算符背后的工作原理。

17.7　研究 LINQ 查询运算符

LINQ 现在定义了很多查询运算符，在这个章节里笔者将介绍其中的几个。但需要知道的是，在安装 LINQ 时，读者会收到一个非常有用的名符其实的 Standard Query Operators.doc（标准查询运算符文档）白皮书。默认情况下它位于“C:\Program File\LINQ Preview\Docs”目录下，这个文档讨论了每个运算符的细节。读者可以详细查阅这个参考文献，如表 17-3 所示只列出了其中的几个运算符。

表 17-3　各种 LINQ 查询运算符

查询运算符	含义
from，in	用于定义任何 LINQ 表达式的主干
Where	用于定义从一个容器里取出那些项的限制条件
Select	用于从容器中选择一个序列
Join，in，on，equals，into	基于指定的键来做关联操作。注意，这些“关联”不必与关系数据库的数据有什么关系
Orderby，ascending，descending	允许结果子集按升序或降序排序
Groupby，into	用待定的值来对数据分组后得到一个子集

除了表 17-3 中列出的部分运算符外，Sequence 类型还提供了一套没有直接查询运算符的简化符号，是以扩展方法呈现的方法，可以调用这些泛型方法以各种方式来对一个结果集进行转换（Reverse<>( )、ToArray<>( )、ToList<>( )等）。其中的一些方法用于从一个结果集提取单例，也有一些方法是对结果集进行集操作的（Distinct<>( )、Union<>( )、Intersect<>( )等），还有一些方法是对结果结合进行聚合操作的（Count<>( )、Sum<>( )、Min<>( )、Max<>( )等）。

使用这些查询运算符以及 System.Query.Sequence 类型的其他辅助成员，读者可以构建非常有表达力的查询表达式。

17.8　构建 LINQ 查询表达式

为了帮助读者进一步熟悉查询表达式的句法，下面这个例子还是使用前面创建的 StudentBaseInfo 类。这些方法显示了由各种查询运算符操作后得到的结果集。

17.8.1　基本的选择语法

因为 LINQ 查询表达式是在编译时校验的，所以记住这些运算符的次序是非常重要的。简单地说，每个 LINQ 查询表达式都是使用 from、in 和 select 运算符来建立的，如下所示。

```
var result=from item in container select item; //LINQ 查询表达式
```

在此，查询表达式不过是从容器里挑选出每个项而已，类似 Select 的 SQL 语句，如代码 17-19 所示。

代码 17-19　查询表达式中所有的对象

```
┆
using System.Data.Linq;
public partial class UseLinq : System.Web.UI.Page
{
    public void BasicSelect(StudentBaseInfo[ ] students)
```

```
    {
        //得到所有的对象
        var allstu = from s in students select s;
        foreach (var s in allstu)
            //输出查询结果
            Response.Write(s.ID+","+s.Name+","+s.Email+"</br>");
      }
    protected void Page_Load(object sender, EventArgs e)
    {
        //构建数据源
        StudentBaseInfo [ ] students = new[ ]{
        new StudentBaseInfo(200801, "Student01", "Student01@web.com"),
        new StudentBaseInfo(200802, "Student02", "Student02@web.com"),
        new StudentBaseInfo(200803, "Student03", "Student03@web.com"),
        new StudentBaseInfo(200804, "Student04", "Student04@web.com"),
        new StudentBaseInfo(200805, "Student05", "Student05@web.com"),
        new StudentBaseInfo(200806, "Student05", "Student06@web.com")
         };
          //调用输出方法
        BasicSelect(students);
    }
}
```

测试查询，输出结果如图 17-14 所示。

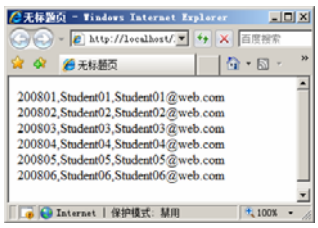

图 17-14　浏览显示对象

其实这个查询表达式并不是很有用，因为子集与传入参数的数据是一模一样的。如果读者愿意的话可以使用传入的参数，通过使用下面的选择语法来只提取每个学生对象的 name 值。

```
//只输出学生的名字
var names = from c in students select c.Name;
            foreach (var n in names)
                Response.Write(n);
```

此例中，names 只是一个实现了 IEnumerable<string>的内部类型，因为对每个学生对象只选择了其 name 属性的值。再强调一下，通过 var 关键词使用隐型类型，那么编码任务就被简化了。

17.8.2　获取数据子集

为从数据容器里得到特定的子集，可以使用 where 运算符，其语法格式如下所示。

```
var result=from item in container where Boolean expression select item;
```

注意：where 运算符预期的一个运算结果是布尔值的表达式。

可以在 UseLinq.aspx 中编写下面的代码。

```
//这里只得到学生 Student3
var stu = from s in students where s.Name == "Student03" select s;
            //输出结果
            foreach(var s in stu)
                Response.Write(s.Name);
```

测试查询，输出结果如图 17-15 所示。

可见，当构建一个 Where 字句的时候，允许利用任意合法的 C#运算符来构建复杂的表

达式。例如，下面只提取那些年龄超过 20 岁姓名为 Student03 的查询。

图 17-15　获取所选对象

```
//这里得到学生姓名为 Student3 和年龄大于 20 岁
var stu = from s in students
where s.Name == "Student03"&&s.Age>20
select s;
            //输出结果
            foreach(var s in stu)
                Response.Write(s.Name);
```

从现有的数据源投影出新的数据形式也是可能的。假如想从传入的 StudentBaseInfo[ ]参数中得到一个只有别名和年龄的结果集，可以定义一个 select 语句，通过 C# 3.0 匿名类型动态地形成新的类型。

17.8.3　逆转结果集的顺序

使用 Reverse 操作，能够将序列中的元素的顺序进行反转。Enumerable 类的 Reverse()方法的原型如下。

```
Public static IEnumerable<TSource>Reverse<TSourse>(this IEnumerable<TSource>source)
```

可以在 UseLinq.aspx 中编写下面的代码，如代码 17-20 所示。

代码 17-20　使用 Reverse 操作逆转结果集的顺序

```
public void ReverseSelect(StudentBaseInfo[ ] students)
    {
        //逆序输出所有对象
        var stu = (from s in students select s).Reverse<StudentBaseInfo>();
        foreach (var s in stu)
            Response.Write(s.Name + "</br>");
    }
```

测试查询，输出结果如图 17-16 所示。

图 17-16　浏览逆序输出效果

17.8.4　对表达式进行排序

一个查询表达式可以接受一个 orderby 运算符，来对子集的项排序。在默认情况下，按升序排序，因此对字符串的排序是按字母表来排序的，对数值数据排序是从小到大来排的，依此类推。假如读者想按降序来查看结果集，只要包含一个 descending 运算符就可以了，如代码 17-21 所示。

代码 17-21　查询表达式的排序

```
public void OrderedResults(StudentBaseInfo[ ] students)
    {
        //对所有学生按姓名排序
        var sb = from s in students
                 orderby s.Name
                 select s;
        //输出排序结果
```

```
        foreach (var s in sb)
            Response.Write(s.ID+" ,"+s.Name+" ," +s.Email+"</br>");
    }
```

测试查询，输出结果如图 17-17 所示。

可见升序是默认的，也可以利用 ascending 运算符来明确意图，如下所示。

```
var sb = from s in students
                    orderby s.Name descending
                    select s;
```

测试查询，输出结果如图 17-18 所示。

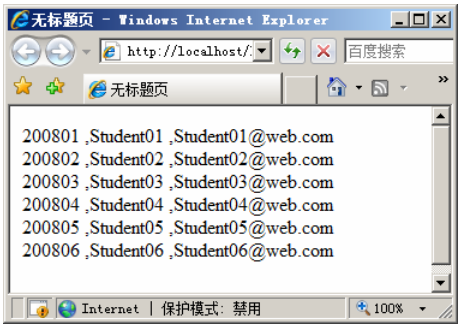

图 17-17　浏览默认升序效果

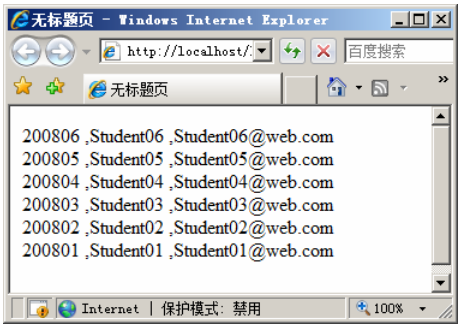

图 17-18　按名字降序排列效果

有了以上例子，现在就可以总结查询表达式的基本语法格式了，格式如下。

```
var result=from item in container orderby value
      ascending/descending select item;
```

假如读者想对一个子集进行排序，可以参考以下的查询格式。

```
var result=from item in container
            where Boolean expression
            orderby value ascending/descending
            select item;
```

17.9　使用 LINQ 到 SQL 来查询关系数据库

LINQ 到 SQL，就是把 LINQ 查询表达式用在关系数据库中。除了查询本身外，LINQ 到 SQL 还提供了很多位于 System.Data.DLinq.dll 程序集的类型，便于代码库与物理库的数据库引擎之间的数据交互。

LINQ 到 SQL 的主要目的，是在关系数据库和它们进行交互的编程逻辑间提供一致性。例如，不用一段字符串而使用强类型的 LINQ 查询来表示数据库查询。同样，不把关系数据当做记录流来处理，而使用标准的面向对象技术来与数据交互。因为 LINQ 到 SQL 允许读者把数据访问直接集成在 C#代码库中，所以手工创建很多自定义类和数据访问库来隐藏 ADO.NET 编码的需求就极大地降低了。

当用 LINQ 到 SQL 编程时，不会看到诸如 SqlConnection、SqlCommand 和 SqlDataAdapter 等常见的 ADO.NET 类型的对象，或是 System.Transactions 命名空间的成员。通过使用 LINQ

查询表达式和其定义的实体类、DataContext 类型，读者可以进行所有预期的数据库的创建、获取、更新和删除操作，以及定义事务性上下文，创建新的数据库实体（或整个数据库）、调用存储过程和其他以数据库为中心的活动等。

此外，LINQ 到 SQL 中的类型（诸如 DataContext）是可以与标准 ADO.NET 数据库类型集成的。例如，一个重载的 DataContext 构造器接受一个与 IDbConnection 兼容的对象作为输入，这是所有 ADO.NET 连接对象都支持的接口。通过这样的方式，现有的 ADO.NET 数据访问库就可以与 C# 3.0 LINQ 查询表达式集成起来，反之亦然。事实上就微软而言，LINQ 到 SQL 只是 ADO.NET 大家庭的一个新成员而已。

17.9.1　LINQ to SQL 概述

LINQ to SQL 是将对象关系映射到.NET 框架中的一种实现。它可以将关系数据库（如 SQL Server 数据库）映射为.NET Framework 中的一些类，然后开发人员就可以使用 LINQ to SQL 对数据库中的数据进行查询、修改、插入、删除等操作。总之，通过使用 LINQ to SQL，开发人员可以使用 LINQ 技术访问基于关系的数据库（如 SQL Server 数据库），就如访问内存中的集合一样。

注意: 为了更加具体地说明 LINQ to SQL 的功能，以下使用 SQL Server 数据库介绍 LINQ to SQL 的功能。

LINQ to SQL 最重要的一个功能，就是为 SQL Server 数据库创建一个对象模型（由基于.NET 框架的类组成），并将该对象模型映射到 SQL Server 数据库中相应的对象（如表、列、外键关系、存储过程、函数等）。其中，LINQ to SQL 类映射到 SQL Server 数据库中的表，这些 LINQ to SQL 类被称为“实体类”。实体类中的属性或字段映射到 SQL Server 数据库中表的列。实体类之间的关系映射到 SQL Server 数据库中的外键关系。LINQ to SQL 类中的方法映射为 SQL Server 数据库中的存储过程或函数。LINQ to SQL 对象模型和 SQL Server 数据库中对象的映射关系如表 17-4 所示。

表 17-4　LINQ to SQL 对象模型和 SQL Server 数据库中对象的映射关系

LINQ to SQL 对象模型的基本元素	SQL Server 数据库中的对象
实体类	表
属性（或字段）	列
关联	外键关系
方法	存储过程或方法

1．实体类和数据库表

实体类和.NET Framework 中的其他类比较相似，但是实体类使用 TableAttribute 属性来描述其与数据库中表的关联关系。代码 17-22 创建了一个名称为 SutdentInfo 的实体类，该实体类映射到 LinqDB 数据库中的 StudentInfo 表。

代码 17-22　实体类和数据库表

```
[Table(Name="StudentInfo")]
public class StudentInfo
{
        public int ID;
        public string name;
         ┆
}
```

2．属性（或字段）和数据库表中的列

如果一个实体类映射到数据库中的表，那么它的属性或字段可以被映射到数据库表中的列。其中，它们之间的映射关系使用 ColumnAttribute 属性表示。代码 17-23 创建了一个名称为 StudentInfo 的实体类，该实体类映射到 LinqDB 数据库中的 StudentInfo 表。ID 属性映射到 StudentInfo 表的 ID 列，并且 ID 列为 StudentInfo 表的主键。name 属性映射到 StudentInfo 表中的 name 列。

代码 17-23　Username 属性映射到 StudentInfo 表中的 Username 列

```
[Table(Name=”StudentInfo”)]
public class StudentInfo
{
        public int ID;
        public string name;
        //映射 ID 列
        [Column(IsPrimaryKey=true)]
        Public in ID
        {
                get{return iD;}
                set{iD=value;}
}
///映射 Studentname 列
[Column]
Public string name
{
        get{return name;}
         set{name=value;}
}
……
}
```

3．关系和数据库表的外键关系

在 LINQ to SQL 中，LinqDB 数据库中的外键关联通过 AssociationAttribute 属性表示。LinqDB 数据库中的 StudentInfo 和 UserRole 表之间就存在外键关系（UserRole 表的 UserID 列应用 StudentInfo 表的 ID 列作为外键）。代码 17-24 在 StudentInfo 类中创建了一个名称为 UserRole 的关联，该关联映射到 StudentInfo 和 UserRole 表的外键关系（StudentInfo_UserRole）。

代码 17-24　UserRole 关联映射到 StudentInfo 和 UserRole 表的外键关系

```
[Table(Name="StudentInfo")]
public class StudentInfo
```

```
{
      ┆
      private EntitySet<UserRole>_UserRole;
      [Association(Name="StudentInfo_UserRole",Storage="_UserRole",OtherKey="UserID")]
      Public EntitySet<UserRole>UserRole
      {
            get{return this._UserRole;}
            set{this._UserRole.Assign(value)}
}
 ┆
}
```

4．方法和存储过程或函数

LINQ to SQL 使用 FunctionAttribute 属性可以将类中的方法映射到数据库中的存储过程或函数，并使用 ParameterAttribute 属性来描述方法的参数和存储过程或函数中参数的关系。其中，如果类中的方法映射到存储过程，则 IsComposable 属性的值为 false。如果类中的方法映射到函数，则 IsComposable 属性的值为 true。

下面的实例代码将 Pr_GetUsersAndRoles( )方法映射（由 FunctionAttribute 属性指定映射关系）到 LinqDB 数据库中的名称为“Pr_GetUserAndRoles”的存储过程。其中，该存储过程不携带任何参数。

```
[Function(Name=”dbo.Pr_GetUsersAndRoles”)]
Public iSingleResult<Pr_GetUsersAndRolesResult>Pr_GetUsersAndRoles( )
{
      IExecuteResult
result=this.ExecuteMethodCall(this,((MethodInfo)(MethodInfo.GetCurrentMethod( ))));
      return((ISingleResult<Pr_GetUsersAndRolesResult>)(result.ReturnValue));
}
```

17.9.2　使用 LINQ to SQL 可以执行的操作

LINQ to SQL 支持作为 SQL 开发人员所期望的所有关键功能。读者可以查询表中的信息、在表中插入信息，以及更新和删除表中的信息。在下面的示例中，笔者以微软提供的数据库——Northwind 数据库为例，介绍如何使用 LINQ to SQL 实现数据库的选择、插入、更新和删除操作。

1．选择操作

只需要编写一个 LINQ 查询，然后执行该查询以检索结果，即可实现选择。LINQ to SQL 本身会将所有必要操作，转换为读者熟悉的 SQL 操作。代码 17-25 是在 Northwind 数据库的 Customers 表中，检索并显示来自伦敦的客户公司名称。

代码 17-25　检索来自伦敦的客户公司名称

```
// Northwnd 继承自 DataContext.
Northwnd nw = new Northwnd(@"northwnd.mdf");
var Query =
      from cust in nw.Customers
      where cust.City == "London"
```

```
    select cust.CompanyName;              // LINQ 查询语句
foreach (var customer in Query)           //遍历查询结果
{
   Response.Write(customer);
}
```

2．插入操作

若要执行 SQL Insert，首先创建一个新对象，然后添加到对象集合中，最后对 DataContext 调用 SubmitChanges 即可。代码 17-26 通过使用 InsertOnSubmit，向 Customers 表添加了一位新客户。

代码 17-26　插入操作

```
// Northwnd 继承自 DataContext.
Northwnd nw = new Northwnd(@"northwnd.mdf");
Customer mycust = new Customer( );                    //创建新用户对象
mycust.CompanyName = "SomeCompany";
mycust.City = "London";
mycust.CustomerID = "98128";
mycust.PostalCode = "55555";
mycust.Phone = "5517-5517-5555";
nw.Customers.InsertOnSubmit(mycust);
//提交操作
nw.SubmitChanges( );
```

3．更新操作

若要更新某一数据库项，首先要检索该项，然后直接在对象模型中编辑它。在修改了该对象之后，对 DataContext 调用 SubmitChanges 方法就能更新数据库。代码 17-27 检索来自伦敦的所有客户，然后将其所在的城市名称从 London 改为 New London，最后调用 SubmitChanges 方法将所做的更改发送至数据库。

代码 17-27　更新操作

```
Northwnd nw = new Northwnd(@"northwnd.mdf");
var Query =
    from cust in nw.Customers
    where cust.City.Contains("London")
    select cust                    ;//LINQ 查询语句
foreach (var customer in Query)
{
    if (customer.City == "London")
    {
        customer.City = " New London";
    }
}
nw.SubmitChanges( );
```

4．删除操作

若要删除某一项数据，需要从其所属集合中先移除该项，然后对 DataContext 调用 SubmitChanges 方法，提交所做的更改。

注意：LINQ to SQL 无法识别级联删除操作。

代码 17-28 从数据库中检索 CustomerID 为 12345 的客户，然后在确认检索到客户行之后，调用 DeleteOnSubmit 方法将该对象从集合中移除，最后调用 SubmitChanges 方法将删除内容转发至数据库。

代码 17-28　删除操作

```
Northwnd nw = new Northwnd(@"northwnd.mdf");
var mycust =
      from cust in nw.Customers
      where cust.CustomerID == "12345"
      select cust;              //先查询存在再删除
      if (mycust.Count( ) > 0)
{
      nw.Customers.DeleteOnSubmit(mycust.First( ));
      nw.SubmitChanges( );
}
```

17.9.3　使用 Visual Studio 2008 创建 DBML 文件

本节介绍使用 Visual Studio 2008 为 SQL Server 数据库 LinqDB 创建 DBML 文件的方法，并为该数据库中的表（如 StudentInfo、Role、UserRole、Category 等）创建实体类，为每一个表的列创建相应的属性，如 LinqDB 数据库中的 StudentInfo 表将和 LinqDB.dbml 文件中的 StudentInfo 类进行映射，StudentInfo 表的列和 StudentInfo 表的属性进行映射。操作步骤如下。

（1）右击“解决方案资源关系器”面板中的 App_Code 节点，在弹出的快捷菜单中单击“添加新项”命令。在打开的“添加新项”窗口中，选择“LINQ to SQL 类”选项，并在“名称”文本框中输入“LinqDB.dbml”，如图 17-19 所示。

（2）打开“服务资源管理器”面板，在其中选择 LinqDB 数据库的各个表，并直接拖动到 LinqDB.dbml 文件的视图面板中。选择 LinqDB 数据库的 Category 表的操作如图 17-20 所示。

图 17-19　“添加新项”窗口

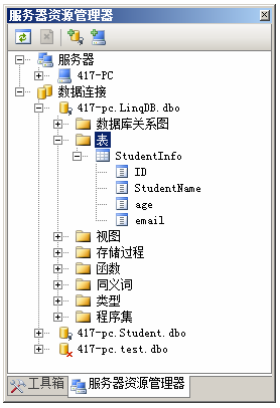

图 17-20　选择 LinqDB 数据库的表

（3）将 LinqDB 数据库中所有的表和存储过程，都直接拖动到 LinqDB.dbml 文件的视图中。此时，LinqDB.dbml 文件的视图如图 17-21 所示。单击 Visual Studio 2008 的工具按钮“保

存”，来保存上述操作的修改，即可创建 LinqDB.dbml 文件。

（4）在“解决方案资源管理器”面板中查看 LinqDB.dbml 文件，如图 17-22 所示。

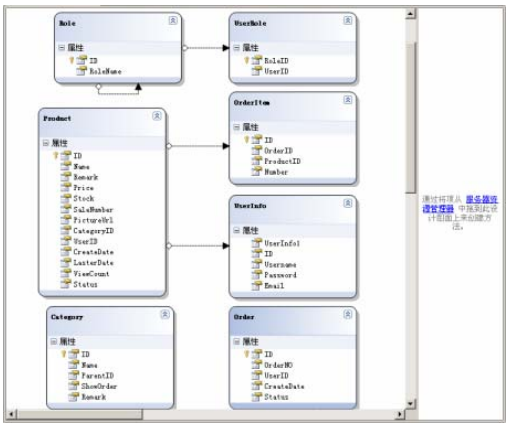

图 17-21　LinqDB 数据库的 LinqDB.dbml 文件

图 17-22　LinqDB.dbml 文件

（5）其中，LinqDB.dbml.layout 文件保存 LinqDB.dbml 文件的布局。该文件为一个 XML 文件，如代码 17-29 所示。

代码 17-29　LinqDB.dbml 文件

```
<?xml version="1.0" encoding="utf-8"?>
<ordesignerObjectsDiagram dslVersion="1.0.0.0" absoluteBounds="0, 0, 11, 8.5" name="LinqDB">
  <DataContextMoniker Name="/LinqDBDataContext" />
  <nestedChildShapes>
    <classShape Id="15b2fd51-126e-4232-9b81-09bf651cdad6" absoluteBounds="6.75, 3.75, 2,
1.6169661458333327">
      <DataClassMoniker Name="/LinqDBDataContext/Category" />
      <nestedChildShapes>
        <elementListCompartment Id="217f97517-872a-4de17-a624-27d9448b120a" absoluteBounds=
"6.765, 4.21, 1.9700000000000002, 1.0569661458333333" name="DataPropertiesCompartment"
titleTextColor="Black" itemTextColor="Black" />
      </nestedChildShapes>
    </classShape>
    ┆
  </nestedChildShapes>
</ordesignerObjectsDiagram>
```

17.10　使用 SqlMetail.exe 生成实体类

在 17.9.3 节中单击“保存”按钮保存 LinqDB.dbml 文件时，Visual Studio 2008 集成开发环境将调用代码生成工具 SqlMetail.exe，创建 LinqDB.dbml 文件的代码，即创建 LinqDB.designer.cs 文件。代码生成工具 SqlMetail.exe 可以为 LINQ to SQL 组件生成代码和映射，主要包括以下 3 个操作。

- ❑　从数据库生成源代码和映射文件。

- 从数据库生成中间数据库标记语言文件（后缀名为.dbml）。
- 从 dbml 生成代码和映射文件。

SqlMetail.exe 在生成 DBML 文件的代码和映射文件时，包括以下 3 个步骤。

（1）DBML 提取器先提取数据库的架构信息，然后把这些信息重新组合到 DBML 文件中。

（2）DBML 验证程序检查 DBML 文件是否存在错误。

（3）如果步骤（2）中未检查到验证错误，则将该文件传递到代码生成器。代码生成器也是一个 LINQ to SQL 组件，它将 DBML 文件转换为 C#或 XML 映射文件。

注意：代码生成工具 SQLMetal.exe 文件位于“C:\ProgramFiles\Microsoft SDKs\Windows\vn.nn\bin”目录下。

17.11　使用 LINQ to XML 操作 XML 文件

本节主要介绍使用 LINQ to XML 操作 XML 文件的方法，如对 XML 文件的添加、查询、更新和删除等操作。以某一 XML 文件为例，使用 XLINQ（LINQ to XML）对 XML 文件进行添加、查询、更新和删除等操作。

示例的 XML 文件为 Sample.xml，如代码 17-30 所示。

代码 17-30　XML 文件：Sample.xml

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <person name="章小盟" age="27" salary="33" />
</root>
```

在 Sample.aspx 设计视图中添加 3 个 TextBox 控件、1 个 Button 控件和 GridView 控件、1 个 CommandField 控件和 1 个 DetailsView 控件，自动生成的代码如代码 17-31 所示。

代码 17-31　使用 LINQ to XML 操作 XML 文件

```
姓名：<asp:TextBox ID="txtName" runat="server"></asp:TextBox>
             年龄：<asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
             薪水：<asp:TextBox ID="txtSalary" runat="server"></asp:TextBox>
          <asp:Button ID="btnAdd" runat="server" Text="添加" OnClick="btnAdd_Click" />
    </p>                                    <!-GridView 控件-->
    <asp:GridView ID="gvPerson" runat="server" DataKeyNames="name" OnSelectedIndexChanged=
"gvPerson_SelectedIndexChanged"
          OnRowDeleting="gvPerson_RowDeleting"
OnRowCancelingEdit="gvPerson_RowCancelingEdit"
          OnRowEditing="gvPerson_RowEditing" OnRowUpdating="gvPerson_RowUpdating">
          <Columns>
             <asp:CommandField ShowSelectButton="True" ShowEditButton="True" ShowDeleteButton=
"True">
               </asp:CommandField>
          </Columns>
    </asp:GridView>
    <br />
```

```
    <asp:DetailsView ID="dvPerson" runat="server" DataKeyNames="name">
    </asp:DetailsView>                    <!-- DetailsView 控件-->
```

后台的功能代码如代码 17-32 所示，主要实现了在 XML 添加数据的 btnAdd_Click 事件，然后用来显示 XML 内容的 BindPerson 方法，还实现了 XML 数据的删除和更新。

代码 17-32　使用 LINQ to XML 操作 XML 文件：Sample.aspx.cs

```
...
using System.Xml.Linq;
public partial class Sample : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //判断页面是否首次加载
        if (!Page.IsPostBack)
        {
            BindPerson( );//执行 BindPrerson( )方法
        }
    }
    private void BindPerson( )
    {
        //加载指定的 xml 文件
        XDocument xml = XDocument.Load(Server.MapPath("Sample.xml"));
        //使用查询语法获取 Person 集合
        var persons = from p in xml.Root.Elements("person")
                      select new
                      {
                          name = p.Attribute("name").Value,
                          age = p.Attribute("age").Value,
                          salary = p.Attribute("salary").Value
                      };
        //绑定数据源
        gvPerson.DataSource = persons;
        gvPerson.DataBind( );
    }
    protected void btnAdd_Click(object sender, EventArgs e)
    {
        //加载指定的 xml 文件
        XDocument xml = XDocument.Load(Server.MapPath("Sample.xml"));
        //创建需要新增的 XElement 对象
        XElement person = new XElement(
            "person",
            new XAttribute("name", txtName.Text),
            new XAttribute("age", txtAge.Text),
            new XAttribute("salary", txtSalary.Text));
        //添加需要新增的 XElement 对象
        xml.Root.Add(person);
        //保存 xml
        xml.Save(Server.MapPath("Sample.xml"));
        gvPerson.EditIndex = -1;
        BindPerson( );
    }
    protected void gvPerson_SelectedIndexChanged(object sender, EventArgs e)
    {
        //加载指定的 xml 文件
```

```
        XDocument xml = XDocument.Load(Server.MapPath("Sample.xml"));
        //使用查询语法获取指定的 Person 集合
        var persons = from p in xml.Root.Elements("person")
                      where p.Attribute("name").Value == gvPerson.SelectedValue.ToString()
                      select new
                      {
                          name = p.Attribute("name").Value,
                          age = p.Attribute("age").Value,
                          salary = p.Attribute("salary").Value
                      };
         //绑定数据源
        dvPerson.DataSource = persons;
        dvPerson.DataBind( );
    }
    protected void gvPerson_RowDeleting(object sender, GridViewDeleteEventArgs e)
    {
        // 加载指定的 xml 文件
        XDocument xml = XDocument.Load(Server.MapPath("Sample.xml"));
        //使用查询语法获取指定的 Person 集合
        var persons = from p in xml.Root.Elements("person")
                      where p.Attribute("name").Value== gvPerson.DataKeys[e.RowIndex].
Value.ToString( )
                      select p;
        //删除指定的 XElement 对象
        persons.Remove( );
        //保存 xml
        xml.Save(Server.MapPath("Sample.xml"));
        gvPerson.EditIndex = -1;
        BindPerson( );//执行 BindPrerson( )方法
    }
    protected void gvPerson_RowUpdating(object sender, GridViewUpdateEventArgs e)
    {
        //加载指定的 xml 文件
        XDocument xml = XDocument.Load(Server.MapPath("Sample.xml"));
        //使用查询语法获取指定的 Person 集合
        var persons = from p in xml.Root.Elements("person")
                      where p.Attribute("name").Value == gvPerson.DataKeys[e.RowIndex].
Value.ToString( )
                      select p;
       //更新指定的 XElement 对象
        foreach (XElement xe in persons)
        {
            xe.SetAttributeValue("age",
((TextBox)gvPerson.Rows[e.RowIndex].Cells[2].Controls[0]).Text);
            xe.SetAttributeValue("salary",
((TextBox)gvPerson.Rows[e.RowIndex].Cells[3].Controls[0]).Text);
        }
        //保存 xml
        xml.Save(Server.MapPath("Sample.xml"));
        gvPerson.EditIndex = -1;
        BindPerson();
    }
    protected void gvPerson_RowEditing(object sender, GridViewEditEventArgs e)
    {
        gvPerson.EditIndex = e.NewEditIndex;
        BindPerson( );//执行 BindPrerson( )方法
    }
```

```
    protected void gvPerson_RowCancelingEdit(object sender, GridViewCancelEditEventArgs e)
    {
        gvPerson.EditIndex = -1;
        BindPerson( );//执行 BindPrerson( )方法
    }
}
```

最终的效果如图 17-23 所示。

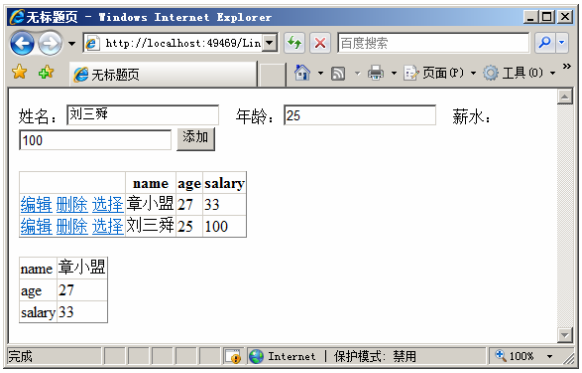

图 17-23　操作 XML 的效果

17.12　小结

LINQ 是一系列相关的技术，用于提供一个单一的、对称的方式来与各种形式的数据交互。就像本章所解释的那样，LINQ 可以与任何实现了 IEnumerable<T>接口的类型交互。这些类型包括简单数组，以及泛型的和非泛型数据集合。LINQ to SQL 是应用于关系数据库的相同技术，而 LINQ to XML 则允许使用函数式的 LINQ 语法来操作 XML 文档。