

# 第 56 章

## LINQ to SQL

### 本章内容:

- 使用 LINQ to SQL 和 Visual Studio 2010
- 把 LINQ to SQL 对象映射到数据库实体上
- 脱离 O/R 设计器构建 LINQ to SQL 操作
- 使用 O/R 设计器和自定义对象
- 用 LINQ 查询 SQL Server 数据库
- 存储过程和 LINQ to SQL

在 C# 2010 中, 令人激动的新增功能之一是.NET LINQ(Language Integrated Query, 语言集成查询架构)架构。LINQ 主要在编程数据集成的基础上提供了一种轻型外观。这是一个很好的功能, 因为数据是最重要的。

每个应用程序都以某种方式处理数据, 无论这些数据来自内存(内存中的数据)、数据库、XML 文件、文本文件还是其他地方。许多开发人员发现很难把 C#中强类型化的、面向对象的数据移动到数据层中(其中对象是辅助类的成员)。在最好的情况下, 从一个环境转换到另一个环境充满了易于出错的操作, 且顶多是一个重组过程。

在 C#中, 用对象编程意味着利用一个强类型化的强大功能处理代码。很容易在名称空间中导航, 使用 Visual Studio IDE 中的调试器等。但是, 在访问数据时, 情况就大不相同了。

这是一个没有强类型化的环境, 调试很痛苦, 甚至是不可能的, 大部分时间都花在把字符串作为命令发送给数据库。开发人员还必须注意基础数据, 了解它的构造方式或者所有数据点的相互关系。

LINQ 为基础数据存储器提供了一个强类型化的界面。LINQ 为开发人员提供了在编码环境下工作的方式, 可以把基础数据作为对象访问, 使用 IDE、IntelliSense 甚至调试功能。

有了 LINQ, 我们创建的查询现在就变成.NET Framework 和其他环境中的重要成员。在对数据存储器执行查询时, 会很快发现它们现在正常工作且行为方式类似于系统中的类型。这说明, 现在可以使用任意兼容.NET 的语言并查询基础数据存储器, 这在以前是不可能的。



第 11 章概述了 LINQ。

图 56-1 显示了 LINQ 在查询数据中的作用。

在图 56-1 中，根据要在应用程序中处理的基础数据的不同，有不同类型的 LINQ 功能。下面列出了各种 LINQ 技术：

- LINQ to Objects
- LINQ to DataSets
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

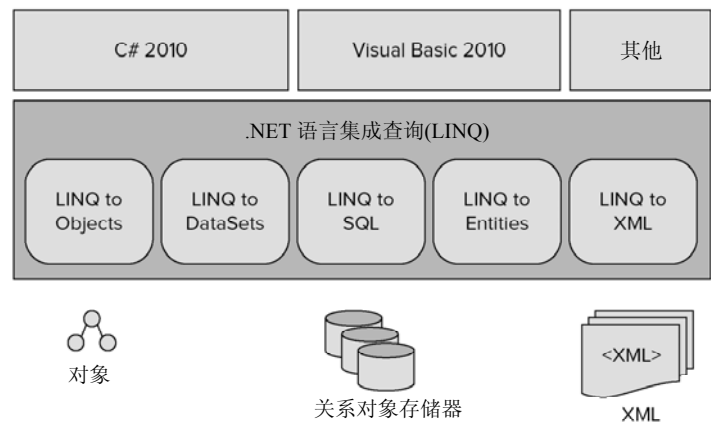


图 56-1

对于开发人员，类库提供了可以使用 LINQ 查询的对象，像查询任何其他数据存储器那样。其实，对象只不过是存储在内存中的数据。实际上，对象本身可能就在查询数据。此时应使用 LINQ to Objects。

LINQ to SQL(本章的重点)、LINQ to Entities 和 LINQ to DataSets 都提供了查询关系数据的方式。使用 LINQ 可以直接查询数据库，甚至查询数据库提供的存储过程。图 56-1 中的最后一项是使用 LINQ to XML 查询 XML(详见第 33 章)。LINQ 非常强大的原因是它对所查询的内容没有任何限制，因为查询非常类似。

### 56.1 LINQ to SQL 和 Visual Studio 2010

特殊的是 LINQ to SQL 是在 SQL Server 数据库上设置一个强类型化界面的方式。LINQ to SQL 提供的方法是当前可用于查询 SQL Server 最简单的方法。它不仅在数据库中查询单个表。例如，如果调用 Northwind 数据库的 Customers 表，提取该数据库的 Orders 表中某位顾客的特定订单，LINQ 就使用两个表之间的关系并代表用户进行查询。LINQ 会查询数据库，加载要在代码中使用的数据(也是强类型化的)。

注意，LINQ to SQL 不仅可以查询数据，还可以执行需要的 Insert/Update/Delete 语句。

也可以与整个过程交互操作，并定制所执行的操作，给 CRUD(Create/Read/Update/Delete)操作添加自己的业务逻辑。

Visual Studio 2010 非常重视 LINQ to SQL，因为这是一个功能丰富的用户界面，允许设计要使用的 LINQ to SQL 类。

下一节介绍如何创建第一个 LINQ to SQL 实例，从 Northwind 数据库的 Products 表中提取数据项。

56.1.1 调用 Products 表

作为使用 LINQ to SQL 的一个例子，本章首先调用 Northwind 数据库中的单个表，用这个表把一些结果显示在屏幕上。

首先创建一个控制台应用程序(使用 .NET Framework 4)，把 Northwind 数据库文件添加到这个项目中(Northwind.MDF)。



下面的例子使用 SQL Server Express Database 文件 Northwind.mdf。要获得这个数据库，请搜索 “Northwind and pubs Sample Database for SQL Server 2000”。这个链接在 <http://www.microsoft.com/downloads/details.aspx?FamilyId=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>。安装后，Northwind.mdf 文件位于 C:\SQL Server 2000 Sample Database 目录下。要把这个数据库添加到应用程序中，可以右击正在使用的解决方案，选择 Add Existing Item 命令。在打开的对话框中，找到刚才安装的 Northwind.mdf 文件。如果在获得使用该数据库的权限方面有问题，那么可以从 Visual Studio Server Explorer 中建立与该文件的数据连接，使自己成为该数据库的相应用户。Visual Studio 就会进行相应的改变，允许使用该数据库。

现在，在 Visual Studio 2010 中创建 .NET Framework 4 提供的许多应用程序类型时，默认已经有了使用 LINQ 的正确引用。在创建控制台应用程序时，代码中会包含如下 using 语句：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

可以看出，代码已经包含需要的 LINQ 引用。

56.1.2 添加 LINQ to SQL 类

下一步是添加一个 LINQ to SQL 类。使用 LINQ to SQL 时，一个主要优点是 Visual Studio 2010 会使该任务尽可能简单。Visual Studio 提供了一个与对象相关的映射设计器，称为 O/R 设计器，它允许可视化地设计数据库要映射的对象。

首先，右击解决方案，从弹出的菜单中选择 Add New Item 命令。在 Add New Item 对话框中，包含 LINQ to SQL Classes 选项，如图 56-2 所示。

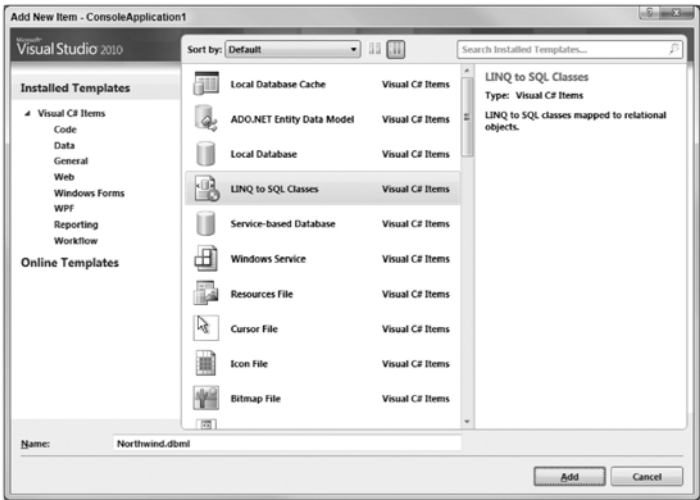


图 56-2

因为这个例子使用 Northwind 数据库，所以把文件命名为 Northwind.dbml。单击 Add 按钮，就会创建几个文件，图 56-3 显示了添加 Northwind.dbml 文件后的 Solution Explorer 窗口。

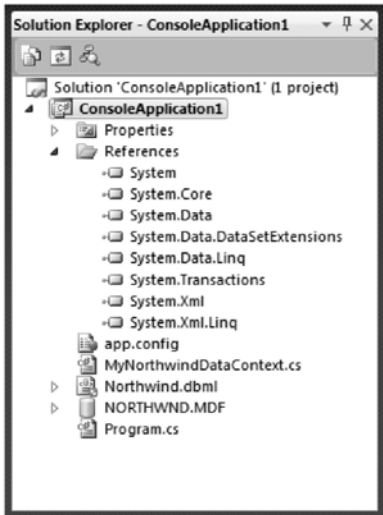


图 56-3

这个操作在项目中添加了许多内容。添加了 Northwind.dbml 文件，它包含两个组件。因为前面添加的 LINQ to SQL 类要使用 LINQ，所以也添加了下面的引用：System.Core、System.Data.DataSetExtensions、System.Data.Linq 和 System.XML.Linq。

56.1.3 O/R 设计器概述

在项目(Northwind.dbml 文件)中添加 LINQ to SQL 类时，还在 IDE 上添加了一个新功能：.dbml 文件的可视化表示。新的 O/R 设计器在 IDE 的文档窗口中显示为一个选项卡。图 56-4 显示了 O/R 设计器第一次启动时的视图。

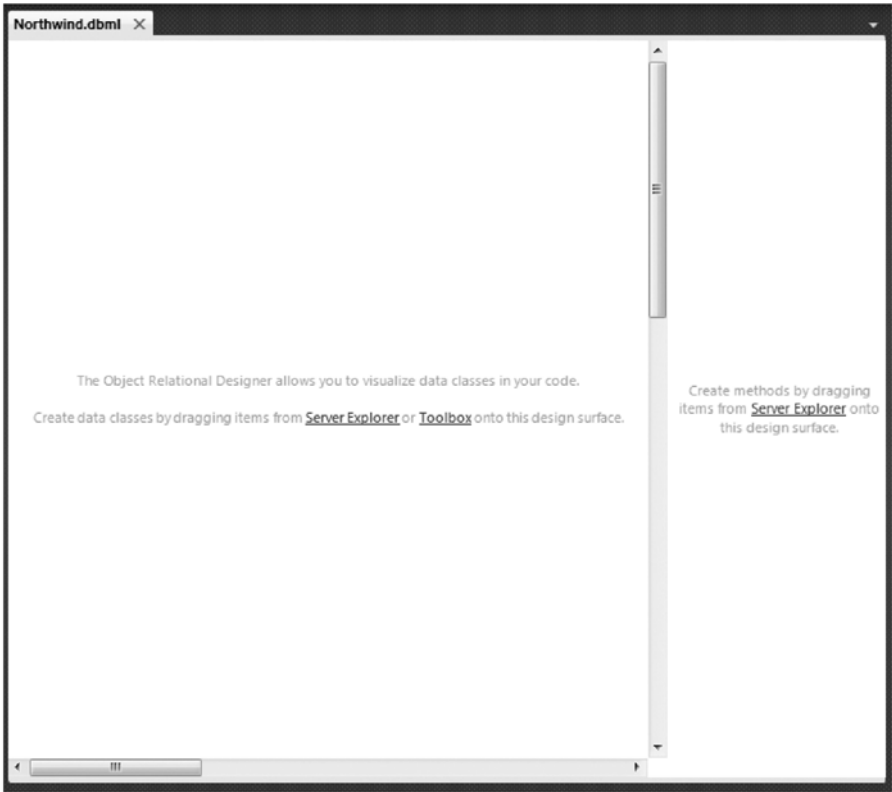


图 56-4

O/R 设计器由两部分组成。第一部分用于显示数据类，它可以是表、类、关联和继承。在这个设计界面上拖动这些项，可以显示所使用的对象的可视化表示。第二部分(右边)用于显示方法，这些方法映射到数据库中的存储过程上。

在 O/R 设计器中查看.dbml 文件时，Visual Studio 工具箱中还有一组 Object Relational Designer 控件。该工具箱如图 56-5 所示。

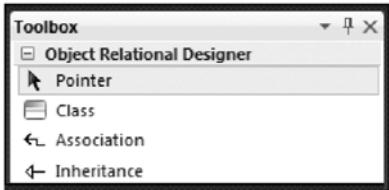


图 56-5

56.1.4 创建 Product 对象

这个例子要使用 Northwind 数据库中的 Products 表，这表示必须创建一个 Products 表，Products 表使用 LINQ to SQL 映射到这个表上。完成该任务只需在 Visual Studio 的 Server Explorer 对话框中打开包含在该数据库中的表对应的视图，把 Products 表拖放到 O/R 设计器的设计界面上。这个操作的结果如图 56-6 所示。

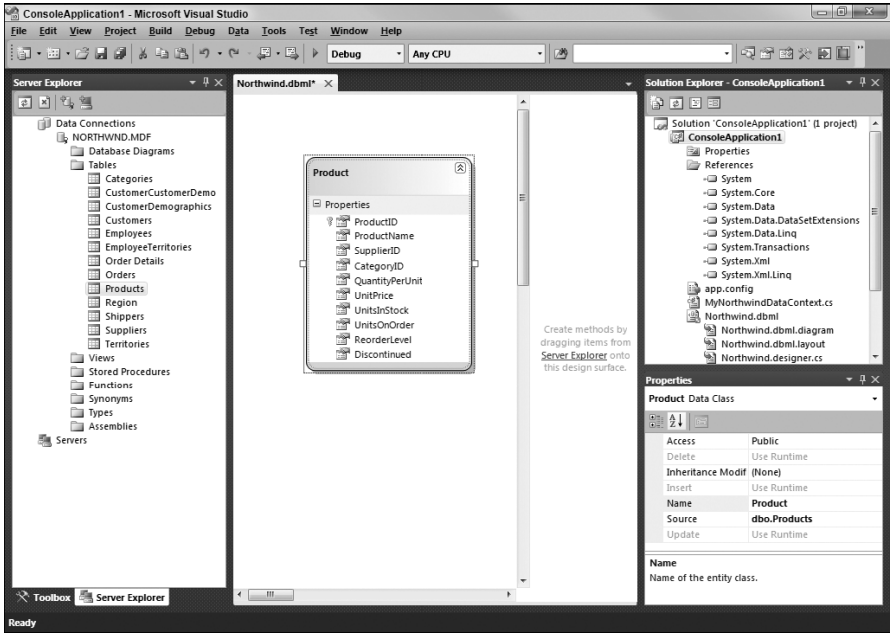


图 56-6

执行这个操作，会把一些代码添加到.dbml 文件的设计器文件中。这些类可以对 Products 表进行强类型化访问。为了演示这个访问，把注意力转向控制台应用程序的 Program.cs 文件上。下面显示了这个例子需要的代码：



可从  
 wrox.com  
 下载源代码

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = dc.Products;

            foreach (Product item in query)
            {
                Console.WriteLine("{0} | {1} | {2}",
                    item.ProductID, item.ProductName, item.UnitsInStock);
            }

            Console.ReadLine();
        }
    }
}

```

[代码下载 ConsoleApplication1.sln](#)

虽然这些代码并不多，但它查询 Northwind 数据库中的 Products 表，提取出要显示的数据。下面单步执行这段代码，从 Mian()方法的第一行开始：

```
NorthwindDataContext dc = new NorthwindDataContext();
```

NorthwindDataContext 对象是 DataContext 类型的一个对象。基本上可以把这个对象看作映射到 Connection 类型对象上的一个对象。这个对象可以使用连接字符串，并为任何需要的操作连接到数据库上。

下一行比较有趣：

```
var query = dc.Products;
```

这里使用了新的 var 关键字，它是一个隐式类型化的变量。如果不能确定输出类型，就可以使用 var 替代类型定义，再在编译期间设置类型。事实上，代码(dc.Products;) 返回一个 System.Data.Linq.Table<ConsoleApplication1.Product> 对象，这就是编译应用程序时设置的 var。因此，这表示也可以编写如下语句：

```
Table<Product>query = dc.Products;
```

这种方法实际上比较好，因为程序员以后查看应用程序的代码时，很容易理解其含义。使用 var 关键字还有一个隐含的优点：程序员可能会发现它有问题。要使用 Table<Product> (这基本上是 Product 对象的一个泛型列表)，就应引用 System.Data.Linq 名称空间。

赋予 query 对象的值是 Products 属性的值，它是 Table<Product> 类型。之后，下面的代码迭代 Table<Product> 中的 Product 对象集合：

```
foreach (Product item in query)
{
    Console.WriteLine("{0} | {1} | {2}",
        item.ProductID, item.ProductName, item.UnitsInStock);
}
```

在本例中，这个迭代从 Product 对象中提取出 ProductID、ProductName 和 UnitsInStock 属性，并把它们写到程序中。因为我们仅使用该表的几项，所以还可以使用 O/R 设计器中的选项删除从数据库中提取的、不感兴趣的列。程序的结果如下：

```
1 | Chai | 39
2 | Chang | 17
3 | Aniseed Syrup | 13
4 | Chef Anton's Cajun Seasoning | 53
5 | Chef Anton's Gumbo Mix | 0

** Results removed for space reasons **

73 | R d Kaviar | 101
74 | Longlife Tofu | 4
75 | Rh nbr u Klosterbier | 125
76 | Lakkalik ri | 57
77 | Original Frankfurter grüne Soße | 32
```

从这个例子可以看出，很容易使用 LINQ to SQL 查询 SQL Server 数据库。

## 56.2 对象如何映射到 LINQ 对象上

LINQ 的优点是提供了在代码(和 IntelliSense)中使用的强类型化对象，这些对象映射到已有的数

数据库对象上。另外，LINQ 不过是这些已有数据库对象上的一个瘦外观。表 56-1 列出了数据库对象和 LINQ 对象之间的映射。

表 56-1

| 数据库对象 | LINQ 对象     |
|-------|-------------|
| 数据库   | DataContext |
| 表     | 类和集合        |
| 视图    | 类和集合        |
| 列     | 属性          |
| 关系    | 嵌套的集合       |
| 存储过程  | 方法          |

左侧是正在处理的数据库。数据库是一个完整的实体——表、视图、触发器、存储过程构成数据库。在右侧 LINQ 对象中，有一个 DataContext 对象，它绑定到数据库上。为了与数据库进行必要的交互操作，该对象包含一个连接字符串，并管理所发生的所有事务；它还负责记录操作，并管理数据的输出。DataContext 对象通过数据库全面管理事务。

如前面的控制台应用程序例子所示，表转换为类。这表示如果有一个 Products 表，就有一个 Product 类。注意 LINQ 的名称是友好的，因为它把复数形式的表名改为单数形式，给要在代码中使用的类指定合适的名称。除了用作类的数据库表之外，把数据库视图也看作类。另一方面，把列看作属性，因此可以直接管理列的属性(名称和类型定义)。

关系是在各个对象之间映射的嵌套集合。因此可以定义映射到多个项上的关系。

还必须理解存储过程的映射。在代码中，存储过程实际上映射到 DataContext 实例的方法上。下一节将详细介绍 DataContext 实例和 LINQ 中的表对象。

在处理 LINQ to SQL 的体系结构时，注意其中有 3 层：应用层、LINQ to SQL 层和 SQL Server 数据库层。从前面的例子中可以看出，可以在应用程序的代码中创建强类型化的查询：

```
dc.Products;
```

这个查询会被 LINQ to SQL 层转换为 SQL 查询，之后提供给数据库：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],  
[t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],  
[t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],  
[t0].[Discontinued]  
FROM [dbo].[Products] AS [t0]
```

结果，LINQ to SQL 层通过这个查询从数据库中提取行，把返回的数据变成一个强类型化的对象集合，以便于使用。

### 56.2.1 DataContext 对象

同样，在使用 LINQ to SQL 时，DataContext 对象管理在所使用的数据库中发生的事务。使用 DataContext 对象可以执行许多操作。

在实例化这些对象时，需要几个可选参数，如下所示：

- 一个字符串，表示 SQL Server Express 数据库文件的位置或所使用的 SQL Server 的名称
- 连接字符串
- 另一个 DataContext 对象

前两个可选字符串参数也可以包含自己的数据库映射文件。实例化这个对象后，就可以在许多不同的操作中以编程方式使用它。

1. 使用 ExecuteQuery()方法

使用 DataContext 对象完成的一个简单任务是用 ExecuteQuery<T>()方法快速运行自己编写的命令。例如，如果要使用 ExecuteQuery<T>()方法提取 Products 表中的所有产品，那么代码应如下所示：



可从  
 wrox.com  
 下载源代码

```

using System;
using System.Collections.Generic;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            DataContext dc = new DataContext(@"Data Source=.\\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            IEnumerable<Product>myProducts =
                dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS", "");

            foreach (Product item in myProducts)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }

            Console.ReadLine();
        }
    }
}
    
```

代码下载 ConsoleApplication1.sln

在这个例子中，调用 ExecuteQuery<T>()方法时传递了一个查询字符串，并返回一个 Product 对象集合。在该方法调用中使用的查询是一条简单的 Select 语句，它不需要传递任何参数。由于在查询中没有传递任何参数，因此可以使用双引号作为方法调用的第二个必选参数，甚至也可以不提供这个参数。如果要在查询中替换任何值，那么可以按如下方式构建 ExecuteQuery<T>()方法调用：

```

IEnumerable <Product> myProducts =
    dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS WHERE UnitsInStock > {0}",
    50);
    
```

在这个例子中，{0}是一个占位符，用于替换要传入的参数值，ExecuteQuery<T>()方法的第二个参数是在替换过程中使用的参数。

## 2. 使用 Connection 属性

Connection 属性返回 System.Data.SqlClient.SqlConnection(由 DataContext 对象使用)的一个实例。如果需要与应用程序中使用的其他 ADO.NET 代码共享这个连接, 或者需要获得该连接提供的 SqlConnection 属性或方法, 就可以使用这个属性。例如, 获得连接字符串就很简单:

```
NorthwindDataContext dc = new NorthwindDataContext();

Console.WriteLine(dc.Connection.ConnectionString);
```

## 3. 使用 ADO.NET 事务

如果有一个可以使用的 ADO.NET 事务, 就可以使用 Transaction 属性把这个事务赋予 DataContext 对象实例。还可以通过 .NET 2.0 Framework 中的 TransactionScope 对象使用事务(需要引用 System.Transactions 程序集):



```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Transactions;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            using (TransactionScope myScope = new TransactionScope())
            {
                Product p1 = new Product() { ProductName = "Bill's Product" };
                dc.Products.InsertOnSubmit(p1);

                Product p2 = new Product() { ProductName = "Another Product" };
                dc.Products.InsertOnSubmit(p2);

                try
                {
                    dc.SubmitChanges();

                    Console.WriteLine(p1.ProductID);
                    Console.WriteLine(p2.ProductID);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }

                myScope.Complete();
            }

            Console.ReadLine();
        }
    }
}
```

}

代码段 ConsoleApplication1.sln

在这个例子中，使用了 TransactionScope 对象，如果数据库中的某个操作失败，就把所有操作滚到原始状态。

4. DataContext 对象的其他方法和属性

除了前面介绍的项之外，DataContext 对象还有许多其他可用的方法和属性。表 56-2 列出了 DataContext 对象的一些方法。

表 56-2

| 方 法            | 说 明                                    |
|----------------|----------------------------------------|
| CreateDatabase | 可以在服务器上创建数据库                           |
| DatabaseExists | 可以确定数据库是否存在，是否可以打开                     |
| DeleteDatabase | 删除相关联的数据库                              |
| ExecuteCommand | 可以给数据库传入要执行的命令                         |
| ExecuteQuery   | 可以给数据库直接传递查询                           |
| GetChangeSet   | DataContext 对象跟踪数据库中发生的变化，这个方法可以访问这些变化 |
| GetCommand     | 可以访问要执行的命令                             |
| GetTable       | 可以访问数据库中的表集合                           |
| Refresh        | 可以利用数据库中存储的数据刷新对象                      |
| SubmitChanges  | 在代码中建立的数据库中执行 CRUD 命令                  |
| Translate      | 把 IDataReader 转换为对象                    |

除了这些方法之外，DataContext 对象还包含一些属性，如表 56-3 所示。

表 56-3

| 属 性                    | 说 明                                                       |
|------------------------|-----------------------------------------------------------|
| ChangeConflicts        | 调用 SubmitChanges()方法时，提供一个导致并发冲突的对象集合                     |
| CommandTimeout         | 可以设置一个超时期限，允许在这个时间段内在数据库上运行命令。如果查询需要执行较长时间，就应把这个属性设置为较高的值 |
| Connection             | 可以使用客户端使用的 System.Data.SqlClient.SqlConnection 对象         |
| DeferredLoadingEnabled | 可以指定是否延迟加载一对多或一对一关系                                       |
| LoadOptions            | 可以指定或检索 DataLoadOptions 对象的值                              |
| Log                    | 可以指定在查询中使用的命令的输出位置                                        |
| Mapping                | 提供映射所基于的 MetaModel                                        |
| ObjectTrackingEnabled  | 指定是否跟踪数据库中对象的变化，以进行事务处理。如果正在处理只读数据库，就应把这个属性设置为 false      |
| Transaction            | 可以指定数据库中使用的本地事务                                           |

56.2.2 Table<TEntity>对象

Table<TEntity>对象表示在数据库中使用的表。例如，前面使用了 Product 类，它就是一个 Table<Product>实例。如本章所述，许多方法都可用于 Table<TEntity>对象。其中一些方法在表 56-4 中定义。

表 56-4

| 方 法                           | 说 明                                                                   |
|-------------------------------|-----------------------------------------------------------------------|
| Attach                        | 可以把一个实体关联到 DataContext 实例上                                            |
| AttachAll                     | 可以把一个实体集合关联到 DataContext 实例上                                          |
| DeleteAllOnSubmit<TSubEntity> | 可以把所有挂起的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作 |
| DeleteOnSubmit                | 可以把一个挂起的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作 |
| GetModifiedMembers            | 提供一个已修改的对象数组，可以访问它们的当前值和更改后的值                                         |
| GetNewBindingList             | 提供一个新列表，以绑定到数据存储器上                                                    |
| GetOriginalEntityState        | 提供一个对象的实例，且显示其初始状态                                                    |
| InsertAllOnSubmit<TSubEntity> | 可以把所有挂起的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作 |
| InsertOnSubmit                | 可以把一个挂起的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作 |

56.3 脱离 O/R 设计器工作

Visual Studio 2010 中的新 O/R 设计器很容易创建 LINQ to SQL 需要的所有对象，但基础架构允许从头开始完成所有任务。这将最大限度地控制实际发生的事件。

56.3.1 创建自己的自定义对象

为了完成与前面 Customer 表相同的任务，需要通过一个类提供 Customer 表。首先在项目 (Customer.cs)中创建一个新类，这个类的代码如下所示：



可从  
 wrox.com  
 下载源代码

```

using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
    }
}
    
```

```

        public string ContactName { get; set; }
        [Column]
        public string ContactTitle { get; set; }
        [Column]
        public string Address { get; set; }
        [Column]
        public string City { get; set; }
        [Column]
        public string Region { get; set; }
        [Column]
        public string PostalCode { get; set; }
        [Column]
        public string Country { get; set; }
        [Column]
        public string Phone { get; set; }
        [Column]
        public string Fax { get; set; }
    }
}

```

代码段 Customer.cs

这里, Customer.cs 文件定义要用于 LINQ to SQL 的 Customer 对象。这个类把 Table 属性赋予它, 用于表示表类。Table 类属性包含一个 Name 属性, 它定义在数据库中使用的表的名称, 这个表名要通过连接字符串中引用。使用 Table 属性还表示, 需要在代码中引用 System.Data.Linq.Mapping 名称空间。

除了 Table 属性之外, 类中定义的每个属性都使用 Column 属性。如前所述, SQL Server 数据库中的列映射到代码的属性上。

### 56.3.2 通过自定义对象和 LINQ 查询

只有具备 Customer 类, 才可以查询 Northwind 数据库中的 Customers 表。完成这一任务的代码通过同一控制台应用程序中的以下示例说明:

```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            DataContext dc = new DataContext(@"Data Source=.\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            dc.Log = Console.Out; // Used for outputting the SQL used

            Table<Customer>myCustomers = dc.GetTable<Customer>();

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",

```

```

        item.CompanyName, item.Country);
    }
    Console.ReadLine();
}
}
}

```

在这个例子中，使用默认的 DataContext 对象，把包含 SQL Server Express 数据库 Northwind 的连接字符串作为一个参数传递。再使用 GetTable<TEntity>()方法填充 Customer 类型的 Table 类。这个例子中的 GetTable<TEntity>()操作使用定制的 Customer 类：

```
dc.GetTable<Customer>();
```

LINQ to SQL 使用 DataContext 对象在 SQL Server 数据库上执行查询，把返回的行作为强类型化的 Customer 对象。这就允许迭代 Table 对象的集合中的每个 Customer 对象，获得需要的信息，如下面的 Console.WriteLine()语句所示：

```

foreach (Customer item in myCustomers)
{
    Console.WriteLine("{0} | {1}",
        item.CompanyName, item.Country);
}

```

运行这段代码，会在控制台应用程序中生成如下结果：

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [Customers] AS [t0]
--Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 4.0.21006.1

Alfreds Futterkiste | Germany
Ana Trujillo Emparedados y helados | Mexico
Antonio Moreno Taquería | Mexico
Around the Horn | UK
Berglunds snabbk  p | Sweden

// Output removed for clarity

Wartian Herkku | Finland
Wellington Importadora | Brazil
White Clover Markets | USA
Wilman Kala | Finland
Wolski Zajazd | Poland

```

56.3.3 通过查询限制所调用的列

注意，该查询检索在 Customer 类文件中指定的每一列。如果删除不需要的列，就可以得到一个 新的 Customer 类文件，如下所示：



可从  
wrox.com  
下载源代码

```

using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]

```

```
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set; }
    [Column]
    public string CompanyName { get; set; }
    [Column]
    public string Country { get; set; }
}
```

代码段 Customer.cs

这里删除了应用程序未使用的所有列。现在，如果运行控制台应用程序，并查看生成的 SQL 查询，结果如下：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]
```

可以看出，在对 Customers 表的查询中，仅使用在 Customer 类中定义的 3 列。

CustomerID 属性比较有趣，因为可以在 Column 属性中使用 IsPrimaryKey 设置，把该列指定为表的主键。这个设置接受一个布尔值，这里它设置为 true。

### 56.3.4 使用列名

列的另一个要点是在 Customer 类中定义的属性名必须与数据库中使用的名称相同。例如，如果把 CustomerID 属性名改为 MyCustomerID，尝试运行控制台应用程序时，就会得到如下异常：

```
System.Data.SqlClient.SqlException was unhandled
  Message="Invalid column name 'MyCustomerID'."
  Source=".Net SqlClient Data Provider"
  ErrorCode=-2146232060
  Class=16
  LineNumber=1
  Number=207
  Procedure=""
  Server="\\\\.\\pipe\\F5E22E37-1AF9-44\\tsql\\query"
```

为了改正这个错误，需要在前面创建的 Customer 自定义类中定义列的名称。为此，可以使用 Column 属性，如下所示：

```
[Column(IsPrimaryKey = true, Name = "CustomerID")]
public string MyCustomerID { get; set; }
```

与 Table 属性一样，Column 属性也包含一个 Name 属性，Name 属性指定列在 Customers 表中显示的名称。

这会生成如下查询：

```
SELECT [t0].[CustomerID] AS [MyCustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]
```

这也意味着，需要使用新的名称 MyCustomerID 引用该列，如 item.MyCustomerID。

### 56.3.5 创建自己的 DataContext 对象

使用普通的 DataContext 对象有时并不是最好的方法，而创建自己的 DataContext 类可以进行更多控制。为此，创建一个新类 MyNorthwindDataContext.cs，使这个类继承自 DataContext。该类最简单的形式如下：



```
using System.Data.Linq;

namespace ConsoleApplication1
{
    public class MyNorthwindDataContext: DataContext
    {
        public Table<Customer>Customers;

        public MyNorthwindDataContext()
            : base(@"Data Source=.\SQLEXPRESS;
                AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
                Integrated Security=True;User Instance=True")
        {
        }
    }
}
```

代码段 MyNorthwindDataContext.cs

这里，MyNorthwindDataContext 类继承自 DataContext，从前面创建的 Customer 类中提供 Table<Customer>对象的一个实例。构造函数是这个类的另一个要求。这个构造函数使用一个基本实例初始化对象的新实例，这个对象引用文件(这里是 SQL 数据库文件的一个连接)。

现在使用自己的 DataContext 对象，可以改变应用程序中的代码，如下所示：



```
using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            MyNorthwindDataContext dc = new MyNorthwindDataContext();
            Table<Customer> myCustomers = dc.Customers;

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                    item.CompanyName, item.Country);
            }

            Console.ReadLine();
        }
    }
}
```

代码下载 ConsoleApplication1.sln

创建 MyNorthwindDataContext 对象的实例后，就允许该类管理到数据库的连接。现在还可以通过 dc.Customers 语句直接访问 Customer 类。



注意，本章的例子都比较纯粹，因为它们都没有包含构建应用程序时通常包含的错误处理和日志功能，而只是阐述本章讨论的要点。

56.4 自定义对象和 O/R 设计器

除了在自己的.cs 文件中构建自定义对象再把该类关联到自己构建的 DataContext 对象上之外，还可以在 Visual Studio 2010 中使用 O/R 设计器构建自己的类文件。以这种方式使用 Visual Studio 时，Visual Studio 会创建相应的.cs 文件，O/R 设计器还提供了类文件和自己建立的所有关系的可视化表示。

在查看.dbml 文件的设计器视图时，注意工具箱中有 3 项：Class、Association 和 Inheritance。

对于这个例子，从工具箱中选择 Class 对象，把它拖放到设计界面上。这会显示泛型类的图像，如图 56-7 所示。

现在可以单击 Class1 名称，把这个类重命名为 Customer。右击该名称的旁边，从弹出的菜单中选择 Add | Property 命令，可以给类文件添加属性。对于这个例子，给 Customer 类添加 3 个属性——CustomerID、CompanyName 和 Country。如果突出显示 CustomerID 属性，那么可以在 Visual Studio 的 Properties 对话框中配置该属性，把 Primary Key 设置从 False 改为 True。也可以突出显示整个类，进入 Properties 对话框，把 Source 属性改为 Customers，因为这是 Customer 对象需要使用的表名。之后，类的可视化表示就如图 56-8 所示。



图 56-7

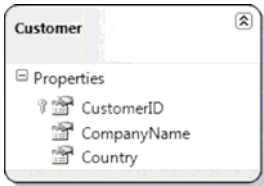


图 56-8

从图 56-8 可以看出，CustomerID 属性的名称旁边可能显示主键图标。此时单击 Northwind.dbml 文件旁边的加号，就会看到 3 个文件——Northwind.dbml.layout、Northwind.designer.cs 和 Northwind.dbml.diagram。Northwind.dbml.layout 文件是有助于 Visual Studio 在 O/R 设计器中进行可视化表示的 XML 文件。但最重要的文件是 Northwind.designer.cs，这是我们创建的 Customer 类文件。打开这个文件，可以查看 Visual Studio 创建的对象。

首先，会发现 Customer 类文件在页面的代码中：

```

[Table(Name="Customers")]
public partial class Customer: INotifyPropertyChanging,
                               INotifyPropertyChanged
{
    // Code removed for clarity
}
    
```

Customer 类是根据我们在设计器中提供的名称而指定的类名。这个类有一个 Table 属性，其值

是 Customers，因为这是该对象在连接到 Northwind 数据库上时需要使用的数据库的名称。  
在 Customer 类中，会发现前面定义的 3 个属性，这里只列出其中一个属性——CustomerID：



可从  
 wrox.com  
 下载源代码

```

[Column(Storage="_CustomerID", CanBeNull=false, IsPrimaryKey=true)]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
    }
}
    
```

代码段 Customer.cs

与前面示例在构建立类时的方式类似，所定义的属性使用 Column 特性和对这个特性可用的一些属性。例如，用 IsPrimaryKey 项设置主键。

除了 Customer 类之外，在创建的文件中还有一个继承自 DataContext 对象的类：

```

[System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext: System.Data.Linq.DataContext
{
    // Code removed for clarity
}
    
```

DataContext 对象(NorthwindDataContext)可以连接到 Northwind 数据库和 Customer 表上，与前面的例子一样。

使用 O/R 设计器可以使数据库对象类文件的创建更简单、直接。然而，同时，如果要完全控制，就可以自己编写所有代码，得到期望的结果。

56.5 查询数据库

如前所述，在应用程序的代码中有许多方式查询数据库。该查询最简单的形式如下：

```

Table<Product> query = dc.Products;
    
```

这条命令把整个 Products 表放在 query 对象实例中。

56.5.1 使用查询表达式

除了使用 dc.Products 把表直接提取出数据库之外，还可以在代码中直接使用强类型化的查询表

达式。下面的代码就是一个例子：



可从  
 wrox.com  
 下载源代码

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        select p;
            foreach (Product item in query)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }

            Console.ReadLine();
        }
    }
}

```

代码下载 ConsoleApplication1.sln

在这个例子中，用“from p in dc.Products select p;”的查询值填充 query 对象(同样是 Table<Product> 对象)。为了增强可读性，把这条命令放在两行上，根据个人爱好也可以放在一行上。

### 56.5.2 查询表达式

在代码中可以使用许多查询表达式，上面的例子仅是一条简单的 select 语句，它返回整个表。表 56-5 列出了其他一些查询表达式。

表 56-5

| 选 项       | 语 法                                                                                                                               |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------|
| Project   | select <expression>                                                                                                               |
| Filter    | where <expression>, distinct                                                                                                      |
| Test      | any(<expression>), all(<expression>)                                                                                              |
| Join      | <expression> join <expression> on <expression> equals <expression>                                                                |
| Group     | group by <expression>, into <expression>, <expression> group join<decision> on <expression> equals <expression> into <expression> |
| Aggregate | count([<expression>]), sum(<expression>), min(<expression>), max(<expression>), avg(<expression>)                                 |
| Patition  | skip [while] <expression>, take [while] <expression>                                                                              |
| Set       | union, intersect, except                                                                                                          |
| Order     | order by <expression>, <expression> [ascending   descending]                                                                      |

56.5.3 使用表达式筛选

除了直接查询整个表之外，还可以使用 `where` 和 `distinct` 选项筛选数据项。下面的例子就查询 `Products` 表中指定类型的记录：

```
var query = from p in dc.Products
             where p.ProductName.StartsWith("L")
             select p;
```

在该示例中，这个查询从 `Products` 表中选择所有以字母 `L` 开头的记录。这通过 `where p.ProductName.StartsWith("L")` 表达式实现。`ProductName` 属性有许多选择方法，可以细调需要的筛选条件。这个操作生成如下结果：

```
65 | Louisiana Fiery Hot Pepper Sauce
66 | Louisiana Hot Spiced Okra
67 | Laughing Lumberjack Lager
74 | Longlife Tofu
76 | Lakkalikööri
```

根据需要还可以给列表添加任意多个表达式。例如，下面的例子给查询添加了两条 `where` 语句：

```
var query = from p in dc.Products
             where p.ProductName.StartsWith("L")
             where p.ProductName.EndsWith("i")
             select p;
```

其中一个筛选表达式查找产品名称以字母 `L` 开头的项，第二个表达式确保还应用了第二个条件，即对应项必须以字母 `i` 结尾。这会生成如下结果：

```
76 | Lakkalik      ri
```

56.5.4 执行连接

除了使用一个表之外，还可以使用多个表，用查询执行连接。如果把 `Customers` 表和 `Orders` 表都拖放到 `Northwind.dbml` 设计界面上，就会得到如图 56-9 所示的结果：

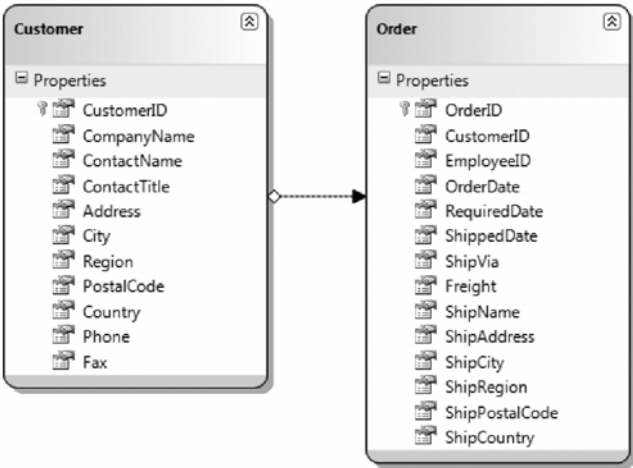


图 56-9

从图 56-9 可以看出，把这些元素拖放到设计界面上后，Visual Studio 就知道这些项之间存在一个关系，并在代码中创建这个关系，用黑色箭头表示它。

现在，就可以在查询中通过 join 语句使用这两个表，如下面的例子所示：



可从  
 wrox.com  
 下载源代码

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            dc.Log = Console.Out;

            var query = from c in dc.Customers
                        join o in dc.Orders on c.CustomerID equals o.CustomerID
                        orderby c.CustomerID
                        select new { c.CustomerID, c.CompanyName,
                                   c.Country, o.OrderID, o.OrderDate };

            foreach (var item in query)
            {
                Console.WriteLine(item.CustomerID + " | " + item.CompanyName
                                   + " | " + item.Country + " | " + item.OrderID
                                   + " | " + item.OrderDate);
            }

            Console.ReadLine();
        }
    }
}

```

代码段 ConsoleApplication1.sln

这个例子从 Customers 表中提取数据，并连接 Orders 表中与 CustomerID 列匹配的记录。这通过 join 语句实现：

```
join o in dc.Orders on c.CustomerID equals o.CustomerID
```

接着用 select new 语句创建一个新对象，这个新对象包含 Customers 表中的 CustomerID、CompanyName、Country 列，以及 Orders 表中的 OrderID 和 OrderDate 列。

在迭代这个新对象的集合时，有趣的是 foreach 语句还使用 var 关键字，因为该类型在此刻未知：

```

foreach (var item in query)
{
    Console.WriteLine(item.CustomerID + " | " + item.CompanyName
                       + " | " + item.Country + " | " + item.OrderID
                       + " | " + item.OrderDate);
}

```

无论如何，item 对象可以访问我们指定的所有属性。运行这个示例，会得到类似于如下部分结果的输出：

|       |  |             |  |         |  |       |  |                       |
|-------|--|-------------|--|---------|--|-------|--|-----------------------|
| WILMK |  | Wilman Kala |  | Finland |  | 10695 |  | 10/7/1997 12:00:00 AM |
| WILMK |  | Wilman Kala |  | Finland |  | 10615 |  | 7/30/1997 12:00:00 AM |
| WILMK |  | Wilman Kala |  | Finland |  | 10673 |  | 9/18/1997 12:00:00 AM |
| WILMK |  | Wilman Kala |  | Finland |  | 11005 |  | 4/7/1998 12:00:00 AM  |
| WILMK |  | Wilman Kala |  | Finland |  | 10879 |  | 2/10/1998 12:00:00 AM |
| WILMK |  | Wilman Kala |  | Finland |  | 10873 |  | 2/6/1998 12:00:00 AM  |
| WILMK |  | Wilman Kala |  | Finland |  | 10910 |  | 2/26/1998 12:00:00 AM |

56.5.5 分组项

也可以通过查询对项分组。在 Northwind.dbml 示例中，把 Categories 表拖放到设计界面上，会发现这个表和 Products 表之间存在一个关系。下面的例子说明了如何按类别对产品分组：



可从  
 wrox.com  
 下载源代码

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        orderby p.Category.CategoryName ascending
                        group p by p.Category.CategoryName into g
                        select new { Category = g.Key, Products = g};

            foreach (var item in query)
            {
                Console.WriteLine(item.Category);

                foreach (var innerItem in item.Products)
                {
                    Console.WriteLine(" " + innerItem.ProductName);
                }

                Console.WriteLine();
            }

            Console.ReadLine();
        }
    }
}

```

代码下载 ConsoleApplication1.sln

这个例子创建了一个新对象，它是一组类别，再把整个 Product 表打包到这个新表(g)中。在此之前，类别使用 orderby 语句按名称排序，因为所提供的订单按升序排列(另一个选项是降序)。输出是 Category(通过 Key 属性传递)和 Product 实例。foreach 语句的迭代对类别迭代一次，给在类别中找到的每种产品迭代一次。

这个程序的部分输出结果如下：

Beverages
 Chai

```

Chang
Guaran á Fant á stica
Sasquatch Ale
Steeleye Stout
Côte de Blaye
Chartreuse verte
Ipoh Coffee
Laughing Lumberjack Lager
Outback Lager
Rhönbräu Klosterbier
Lakkalikööri

Condiments
Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
Grandma's Boysenberry Spread
Northwoods Cranberry Sauce
Genen Shouyu
Gula Malacca
Sirop d'érable
Veggie-spread
Louisiana Fiery Hot Pepper Sauce
Louisiana Hot Spiced Okra
Original Frankfurter grüne Soße
    
```

除了本章介绍的命令和表达式之外，还有许多其他的命令和表达式。

## 56.6 存储过程

前面都是直接查询表，使 LINQ 为操作创建相应的 SQL 语句。在使用那些大量使用存储过程的已有数据库和遵循使用存储过程的最佳实践的数据库时，LINQ 也是一个可用的选项。

LINQ to SQL 把存储过程看作方法调用。如图 56-4 所示，O/R 设计器可以把表拖放到其设计界面上，之后就可以通过编程方式使用表。在 O/R 设计器的右侧，有一个区域可以拖放存储过程。

拖放到 O/R 设计器这个部分的任何存储过程都变成可以在 DataContext 对象中可用的方法。例如，把 TenMostExpensiveProducts 存储过程拖放到 O/R 设计器的这个部分上。

下面的例子说明了如何在 Northwind 数据库中调用这个存储过程：



可从  
wrox.com  
下载源代码

```

using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
        }
    }
}
    
```

```
        ISingleResult<Ten_Most_Expensive_ProductsResult> result =  
            dc.Ten_Most_Expensive_Products();  
  
        foreach (Ten_Most_Expensive_ProductsResult item in result)  
        {  
            Console.WriteLine(item.TenMostExpensiveProducts + " | " +  
                item.UnitPrice);  
        }  
  
        Console.ReadLine();  
    }  
}
```

---

代码下载 [ConsoleApplication1.sln](#)

从这个例子中可以看出，存储过程输出的行被收集到 `ISingleResult<Ten_Most_Expensive_ProductsResult>` 对象中。之后，迭代这个对象，这与余下的操作一样简单。

从这个例子可以看出，调用存储过程是很简单的过程。

## 56.7 小结

.NET Framework 4 版本的一个激动人心的功能是该平台提供的 LINQ 功能。本章介绍了 LINQ to SQL 的使用和查询 SQL Server 数据库时可用的一些选项。

使用 LINQ to SQL 可以通过一组强类型化的操作对数据库执行 CRUD 操作。也可以使用已有的访问功能，不管是与 ADO.NET 交互，还是使用存储过程。