

56

LINQ to SQL

В ЭТОЙ ГЛАВЕ...

- Работа с LINQ to SQL с использованием Visual Studio 2010
- Отображение объектов LINQ to SQL на сущности базы данных
- Построение операций LINQ to SQL без применения O/R Designer
- Использование O/R Designer со специальными объектами
- Запрос базы данных SQL Server с использованием LINQ
- Хранимые процедуры и LINQ to SQL

Скорее всего, вы согласитесь с утверждением, что язык интегрированных запросов .NET (Language Integrated Query Framework — LINQ) в C# 2010 — это одно из самых впечатляющих средств, предоставляемых языком. По сути, LINQ представляет собой легкий фасад для программной интеграции данных. Его важность определяется тем, что в центр внимания поставлены данные.

Почти каждому приложению приходится в той или иной форме взаимодействовать с данными, независимо от того, поступают они из памяти, базы данных, XML-файлов, текстовых файлов или откуда-то еще. Многие разработчики сталкиваются с трудностями при переходе из строго типизированного объектно-ориентированного мира C# к уровню данных, где объекты являются “гражданами второго сорта”. Трансформация одного мира в другой в лучшем случае не работает вообще, а в худшем — оказывается полна чреватых ошибками действий.

В C# программирование с объектами означает замечательную строго типизированную возможность работы с кодом. Вы можете очень легко выполнять навигацию по пространствам имен, работать с отладчиком в IDE-среде Visual Studio и т.п. Однако, когда нужно обращаться к данным, вы обнаружите, что ситуация кардинально меняется.

Вы попадаете в мир, не являющийся строго типизированным, где отладка трудна или вообще невозможна, и большую часть времени приходится посылать команды в базу данных в виде строк. Как разработчик, вы также должны иметь представление о лежащих в основе данных, о том, как они структурированы, и как построены их отношения между собой.

LINQ предлагает разработчикам возможность оставаться в среде кодирования, которую они используют, при этом имея доступ к данным как к объектам, с которыми можно работать в IDE-среде, применяя IntelliSense, и даже выполнять отладку.

Благодаря LINQ, создаваемые запросы становятся “полноправными гражданами” .NET Framework, наряду со всем прочим, к чему вы привыкли. Когда вы начинаете работать с запросами к хранилищу данных, то быстро понимаете, что теперь они работают и ведут себя так, как будто являются типами в системе. Это значит, что вы можете теперь использовать любой .NET-совместимый язык и запрашивать лежащие в основе данные, как не могли никогда ранее.



Введение в LINQ дается в главе 11.

На рис. 56.1 показано место LINQ в рамках запроса к данным.

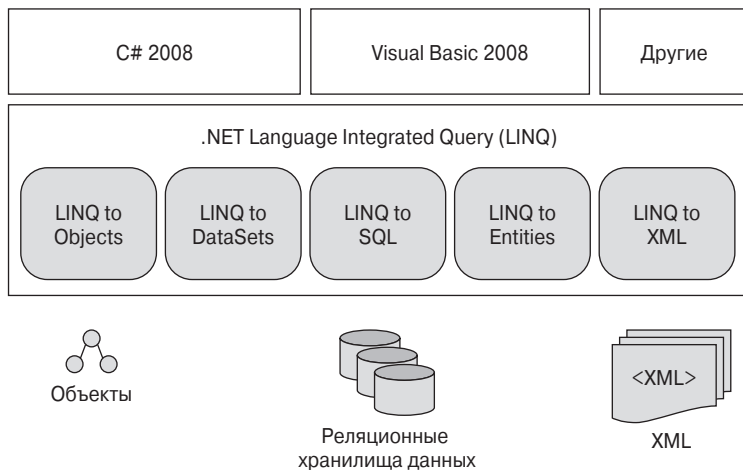


Рис. 56.1. LINQ и запросы к данным

Взглянув на этот рисунок, вы можете видеть, что существуют разные типы средств LINQ, в зависимости от лежащих в основе данных, с которыми будет проводиться работа в приложении. Существуют следующие технологии LINQ:

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

Как разработчик, вы получаете в свое распоряжение библиотеки классов, предоставляющие объекты, которые с помощью LINQ могут быть запрошены подобно любому другому источнику данных. Объекты — это не что иное, как данные, хранящиеся в памяти. Фактически сами объекты могут запрашивать данные. И здесь в игру вступает LINQ to Objects.

LINQ to SQL (тема настоящей главы), LINQ to Entities и LINQ to DataSets предоставляют средства для запроса реляционных данных. Используя LINQ, можно непосредственно запрашивать базу данных или даже запрашивать данные через хранимые процедуры базы данных. Последний элемент диаграммы означает возможность запросов к XML с использованием LINQ to XML (эта тема раскрыта в главе 33). Столь впечатляющим делает LINQ слабое влияние формы исходных данных на внешний вид запросов. Запросы к разным источникам данных очень похожи.

LINQ to SQL и Visual Studio 2010

В частности, LINQ to SQL является средством построения строго типизированного интерфейса к базе данных SQL Server. Попробовав, вы наверняка согласитесь с тем, что LINQ to SQL предлагает простейший подход к запросу данных SQL Server из всех доступных на сегодняшний день. Речь идет не просто о запросе единственной таблицы внутри базы данных, но, например, если вы возьмете таблицу Customers базы данных Northwind и захотите извлечь заказы определенного клиента из таблицы Orders той же базы, то LINQ использует отношения между таблицами, чтобы построить за вас нужный запрос. LINQ запросит базу и загрузит данные в виде, готовом для обработки в коде (опять-таки, строго типизировано).

Важно помнить, что LINQ to SQL — это не только средство запроса данных, но также и средство выполнения нужных операторов Insert/Update/Delete.

Вы можете также взаимодействовать с полным процессом и настраивать выполняемые операции, чтобы добавить собственную бизнес-логику к любой из операций CRUD (Create/Read/Update/Delete — создание/чтение/обновление/удаление).

Среда Visual Studio 2010 тесно взаимодействует с LINQ to SQL — в том смысле, что вы найдете здесь развитый интерфейс пользователя, позволяющий проектировать рабочие классы LINQ to SQL.

В следующем разделе внимание сосредоточено на демонстрации настройки первого экземпляра LINQ to SQL и извлечении элементов из таблицы Products базы данных Northwind.

Обращение к таблице Products

Для примера использования LINQ to SQL этот раздел начинается с обращения к одной таблице базы данных Northwind и использования ее для вывода некоторых результатов на экран.

Для начала создадим консольное приложение (с использованием .NET Framework 4) и добавим файл базы данных Northwind к этому проекту (Northwind.MDF).



В следующем примере используется файл базы данных SQL Server Express Database по имени Northwind.MDF. Чтобы получить эту базу данных, поищите в Интернете по запросу “Northwind and pubs Sample Databases for SQL Server 2000”. Вы можете найти эту ссылку по адресу <http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>. После установки вы найдете файл Northwind.MDF в каталоге вроде C:\SQL Server 2000 Sample Databases. Чтобы добавить эту базу данных в приложение, щелкните правой кнопкой мыши на решении, с которым работаете, и выберите в контекстном меню пункт Add Existing Item (Добавить существующий элемент). В открывшемся диалоговом окне перейдите к местоположению только что установленно-го файла Northwind.MDF. Если возникают проблемы с получением полномочий, необходимых для работы с базой данных, откройте соединение с файлом базы данных в окне Server Explorer внутри Visual Studio, после чего вы получите приглашение зарегистрировать соответствующего пользователя базы данных. Visual Studio проведет соответствующие изменения, чтобы это произошло.

По умолчанию при создании многих типов приложений .NET Framework 4 в среде Visual Studio 2010 вы обнаружите, что у вас уже есть соответствующая ссылка для работы с LINQ. При создании консольного приложения вы получите следующие операторы using в коде:

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

Несложно заметить, что необходимая ссылка LINQ уже на месте.

Добавление класса LINQ to SQL

Следующий шаг состоит в добавлении класса LINQ to SQL. При работе с LINQ to SQL одним из важнейших преимуществ, которые вы обнаружите, будет то, что Visual Studio 2010 берет на себя значительную часть работы, существенно облегчая вашу задачу. В Visual Studio предлагается визуальный конструктор объектно-реляционного отображения, называемый O/R Designer, который позволяет визуально проектировать объект для отображения базы данных.

Чтобы приступить к решению этой задачи, щелкните правой кнопкой мыши на узле решения и выберите в контекстном меню пункт Add New Item (Добавить новый элемент). Среди элементов, доступных в диалоговом окне Add New Item, вы найдете LINQ to SQL Classes (Классы LINQ to SQL), как показано на рис. 56.2.

Поскольку в этом примере используется база данных Northwind, то именем файла является Northwind.dbml. Щелкните на кнопке Add (Добавить) и вы увидите, что в результате будет создана пара файлов. На рис. 56.3 показан Solution Explorer после добавления файла Northwind.dbml.

Это действие добавит к проекту несколько элементов. Будет добавлен файл Northwind.dbml, содержащий два компонента. Поскольку добавленный класс LINQ to SQL работает с LINQ, в код также будут помещены следующие ссылки: System.Core, System.Data, DataSetExtensions, System.Data.Linq и System.Xml.Linq.

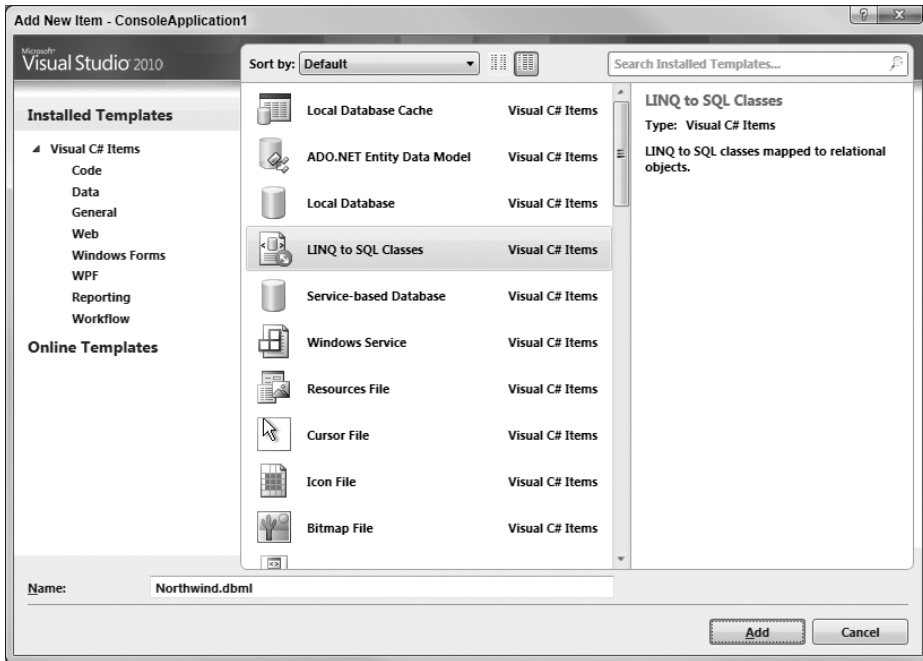


Рис. 56.2. Добавление элемента LINQ to SQL Classes

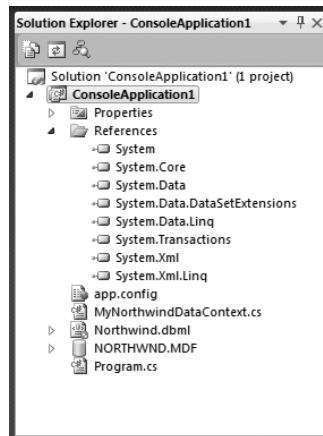


Рис. 56.3. Окно Solution Explorer после добавления файла Northwind.dbml

Введение в O/R Designer

Другое значительное пополнение IDE-среды, которое отображается после добавления к проекту класса LINQ to SQL (файл Northwind.dbml) — это визуальное представление файла .dbml. Новый инструмент O/R Designer появится в виде вкладки внутри окна документа непосредственно в IDE-среде. На рис. 56.4 показано представление O/R Designer при его первом появлении.

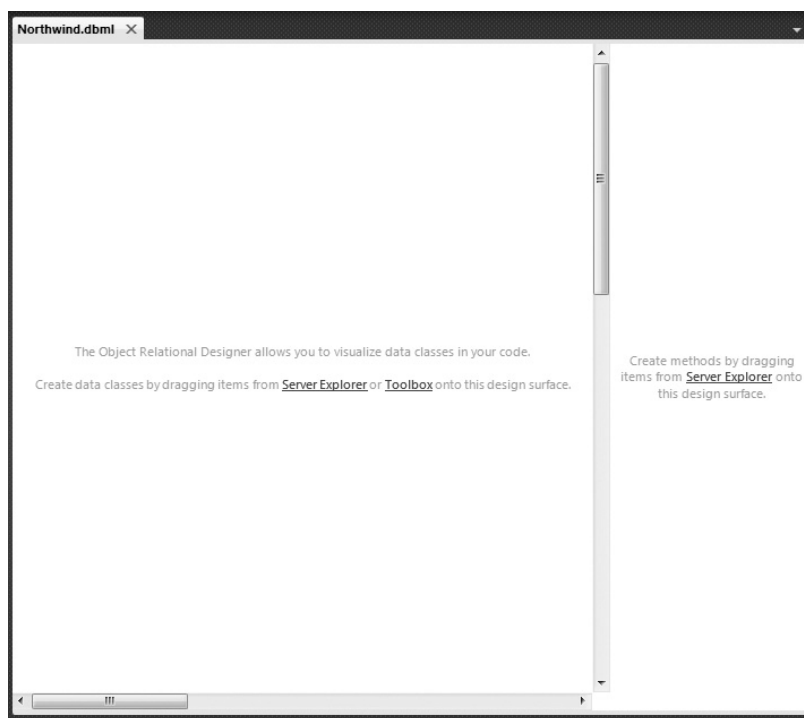


Рис. 56.4. Представление O/R Designer

O/R Designer состоит из двух частей. Первая часть предназначена для классов данных, которые могут таблицами, классами, ассоциациями и наследованиями. Перетаскивание этих элементов на поверхность визуального конструктора даст визуальное представление объекта, с которым можно работать. Вторая часть (справа) предназначена для методов, которые отображаются на хранимые процедуры внутри базы данных.

При просмотре файла .dbml внутри O/R Designer также открывается доступ к набору инструментов Object Relational Designer в панели инструментов Visual Studio. Эта панель инструментов представлена на рис. 56.5.

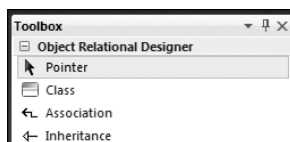


Рис. 56.5. Панель инструментов O/R Designer

Создание объекта Product

В этом примере понадобится работать с таблицей Products из базы данных Northwind, а это означает, что придется создать таблицу Products, которая использует LINQ to SQL для отображения на эту таблицу данных. Решение этой задачи заключается в открытии представления таблиц, содержащихся в базе данных, в диалоговом окне Server Explorer внутри Visual Studio и перетаскивании на поверхность проектирования O/R Designer таблицы Products. Результат этого действия можно видеть на рис. 56.6.

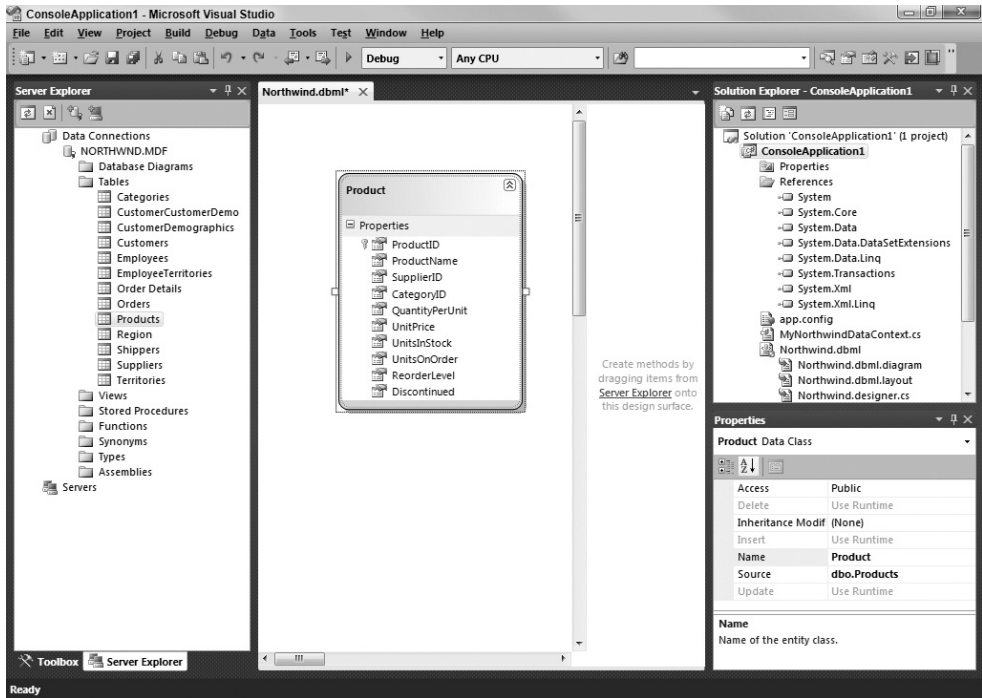


Рис. 56.6. Результат перетаскивания таблицы Products на поверхность проектирования O/R Designer

После выполнения этого действия в файлы визуального конструктора, сопровождающие файл .dbml, добавляется много кода. Эти классы обеспечат строго типизированный доступ к таблице Products. Для демонстрации этого обратите внимание на файл консольного приложения Program.cs. Ниже приведен код, который понадобится для этого примера.

```

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();
        }
    }
}

```

Фрагмент кода *ConsoleApplication1.sln*

Этот фрагмент кода содержит не так уж много строк, но он запрашивает таблицу `Products` внутри базы данных `Northwind` и извлекает из нее данные для отображения. Важно пройтись по этому коду, начиная с первой строки метода `Main()`:

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

`NorthwindDataContext` — это объект типа `DataContext`. По сути, его можно представлять как нечто, отображающееся на объект типа `Connection`. Этот объект работает со строкой соединения и подключается к базе данных для любых необходимых операций.

Следующая строка довольно интересна:

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

Здесь используется новое ключевое слово `var`, которое представляет неявно типизированную переменную. Если вы не уверены в выходном типе, можете использовать `var` вместо определения типа, и тип будет установлен во время компиляции. В действительности код `dc.Products` вернет объект `System.Data.Linq.Table<ConsoleApplication1.Product>`, и именно этот тип будет подставлен вместо `var` во время компиляции. Таким образом, это означает, что с тем же успехом можно было бы записать этот оператор следующим образом:

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

Этот подход на самом деле более предпочтителен, потому что программистам, которые впоследствии будут изучать код вашего приложения, легче будет понять, что здесь происходит. С применением ключевого слова `var` связаны скрытые аспекты, которые могут оказаться проблематичными для программистов. Чтобы использовать тип `Table<Product>`, который просто представляет обобщенный список объектов `Product`, потребуется указать ссылку на пространство имен `System.Data.Linq`.

Значение, присвоенное объекту `query` — это значение свойства `Products`, имеющее тип `Table<Product>`. Отсюда следующая часть кода выполняет итерацию по коллекции объектов `Product`, находящуюся в `Table<Product>`:

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

Итерация в этом случае извлекает свойства `ProductID`, `ProductName` и `UnitsInStock` объекта `Product` и выводит их на консоль. Поскольку вы используете только несколько элементов таблицы, есть возможность в `O/R Designer` удалить столбцы, которые не интересуют. Вывод, полученный от этой программы, представлен ниже:

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

** Часть результатов опущена **

73 | Rödkaviar | 101
74 | Longlife Tofu | 4
75 | Rhönbräu Klosterbier | 125
76 | Lakkaikööri | 57
77 | Original Frankfurter grüne Soße | 32
```

В этом примере видно, насколько легко запрашивать информацию из базы данных `SQL Server` с использованием `LINQ to SQL`.

Как объекты базы данных отображаются на объекты LINQ to SQL

Замечательное свойство LINQ состоит в том, что предоставляет в ваше распоряжение строго типизированные объекты для использования в коде (с помощью IntelliSense), и эти объекты отображаются на объекты базы данных. При этом напомним, что LINQ — это не более чем тонкий фасад для существующих объектов базы данных. В табл. 56.1 показано отображение, существующее между объектами базы данных и объектами LINQ.

Таблица 56.1. Отображение, существующее между объектами базы данных и объектами LINQ

Объект базы данных	Объект LINQ
База данных	<code>DataContext</code>
Таблица	Класс и коллекция
Представление	Класс и коллекция
Столбец	Свойство
Отношение	Вложенная коллекция
Хранимая процедура	Метод

Левая колонка относится к базе данных. База данных — это полноценная сущность: таблицы, представления, триггеры, хранимые процедуры, т.е. все, что составляет базу данных. На стороне LINQ для ее отображения имеется объект `DataContext`. Объект `DataContext` привязан к базе данных. Для организации взаимодействия с ней в нем имеется строка соединения, он управляет всеми транзакциями, заботится о протоколировании и управляет выводом данных. Объект `DataContext` полностью управляет транзакциями в базе данных.

Таблицы, как вы видели в примере, преобразуются в классы. Это значит, что если имеется таблица `Products`, то есть и класс `Product`. Вы заметите, что LINQ имеет дружественную систему именования, заменяя имя таблицы во множественном числе на единственное число, чтобы получить правильное имя класса. То же касается и представлений базы данных. Столбцы, с другой стороны, трактуются как свойства. Это дает возможность управлять атрибутами (именами и определениями типа) столбца напрямую.

Отношения — это вложенные коллекции, которые устанавливают отображение между различными объектами. Они предоставляют возможность определять отношения, отображенные на множественные элементы.

Важно также понимать особенности отображения хранимых процедур. Они в действительности отображаются на методы внутри кода из экземпляра `DataContext`. В следующем разделе более подробно рассматривается объект `DataContext` и объекты таблиц внутри LINQ.

Имея дело с архитектурой LINQ to SQL, вы заметите, что она состоит из трех уровней: собственно приложение, уровень LINQ to SQL и база данных SQL Server. Как было показано в предыдущих примерах, в коде приложения можно создавать строго типизированный запрос:

```
dc.Products;
```

Он, в свою очередь, транслируется в запрос SQL на уровне LINQ to SQL, который затем применяется к базе данных от вашего имени:

```
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]
```

При возврате уровень LINQ to SQL принимает строки, поступившие от базы данных по этому запросу, и возвращает эти данные в коллекцию строго типизированных объектов, с которой можно легко работать.

Объект DataContext

Объект DataContext управляет транзакциями, которые происходят в базе данных, при работе с LINQ to SQL. С помощью объекта DataContext можно выполнять множество действий.


Создавая один из этих объектов, вы заметите, что он принимает несколько необязательных параметров. К ним относятся:

- строка, представляющая местоположение файла базы данных SQL Server Express или имя используемого сервера SQL Server;
- строка соединения;
- другой объект DataContext.

Первые две строки также позволяют включать собственный файл отображения базы данных. Создав экземпляр этого объекта, вы можете программно использовать его для множества типов операций.

Использование *ExecuteQuery*

Одной из простейших вещей, которые можно выполнять с объектом DataContext, является запуск “быстрых” команд, которые вы пишете самостоятельно, используя метод `ExecuteQuery<T>()`. Например, если вы собираетесь получить все продукты из таблицы Products, применяя метод `ExecuteQuery<T>()`, то для этого можете написать примерно такой код.

```
 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();  
        }  
    }  
}
```

Фрагмент кода *ConsoleApplication1.sln*

В этом случае метод `ExecuteQuery<T>()` вызывается с параметром — строкой запроса и возвращает коллекцию объектов `Product`. Запрос, применяемый при вызове метода — это простой оператор `Select`, который не требует никаких дополнительных параметров. Поскольку в запросе не передаются параметры, во втором параметре метода вам нужно либо передать двойные кавычки, либо вовсе не передавать этот параметр. Если бы вы собирались выполнить подстановку каких-либо значений в запрос, то должны были бы сконструировать вызов `ExecuteQuery<T>()` следующим образом:

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

В этом случае `{0}` — место заполнения для значения подставляемого параметра, который вы собираетесь передать, а второй параметр метода `ExecuteQuery<T>()` — параметр, используемый для этой подстановки.


Использование свойства *Connection*

Свойство `Connection` в действительности возвращает экземпляр `System.Data.SqlClient.SqlConnection`, который используется объектом `DataContext`. Это идеальное решение, если вы собираетесь разделить это соединение с другим кодом ADO.NET в приложении, или же когда нужно получить любые свойства или методы `SqlConnection`, которые оно предоставляет. Например, получить строку соединения можно следующим образом:

```
NorthwindDataContext dc = new NorthwindDataContext();
Console.WriteLine(dc.Connection.ConnectionString);
```

Использование транзакции ADO.NET

Если есть транзакция ADO.NET, которую можно использовать, можно присвоить ее экземпляру объекта `DataContext` с помощью свойства `Transaction`. Транзакции можно также использовать через объект `TransactionScope` из .NET 2.0 (понадобится ссылка на сборку `System.Transactions`):

```
 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();
}
}
```

Фрагмент кода *ConsoleApplication1.sln*

В этом случае используется объект `TransactionScope`, и если одна из операций базы данных потерпит неудачу, то все будет приведено в исходное состояние.

Другие методы и свойства объекта *DataContext*

В дополнение к описанным элементам доступно множество других методов и свойств объекта `DataContext`. В табл. 56.2 перечислены доступные методы `DataContext`.

Таблица 56.2. Доступные методы *DataContext*

Метод	Описание
<code>CreateDatabase</code>	Позволяет создать базу данных на сервере.
<code>DatabaseExists</code>	Позволяет определить, существует ли база данных и может ли она быть открыта.
<code>DeleteDatabase</code>	Удаляет ассоциированную базу данных.
<code>ExecuteCommand</code>	Позволяет передавать базе данных команду на выполнение.
<code>ExecuteQuery</code>	Позволяет передавать запросы непосредственно базе данных.
<code>GetChangeSet</code>	Объект <code>DataContext</code> позволяет отслеживать изменения, происходящие в базе данных, и получать доступ к этим изменениям.
<code>GetCommand</code>	Обеспечивает доступ к выполняемым командам.
<code>GetTable</code>	Обеспечивает доступ к коллекции таблиц базы данных.
<code>Refresh</code>	Позволяет обновлять объекты на основе данных, хранящихся в базе.
<code>SubmitChanges</code>	Выполняет подготовленные кодом команды <code>CRUD</code> в базе данных.
<code>Translate</code>	Преобразует <code>IDataReader</code> в объекты.

В дополнение к этим методам объект `DataContext` предоставляет некоторые свойства, перечисленные в табл. 56.3.

Таблица 56.3. Доступные свойства *DataContext*

Свойство	Описание
<code>ChangeConflicts</code>	Предоставляет коллекцию объектов, вызывающих конфликты параллельного доступа при вызове метода <code>SubmitChanges()</code> .
<code>CommandTimeout</code>	Позволяет устанавливать период таймаута, в течение которого разрешено выполняться команде базы данных. Если запрос требует длительного времени на выполнение, вы должны устанавливать это значение побольше.
<code>Connection</code>	Позволяет работать с объектом <code>System.Data.SqlClient.SqlConnection</code> , используемым клиентом.

Свойство	Описание
DeferredLoadingEnabled	Позволяет указать, нужно ли откладывать загрузку отношений “один ко многим” или “один к одному”.
LoadOptions	Позволяет задать или получить значение объекта <code>DataLoadOptions</code> .
Log	Позволяет указать местоположение вывода команды, используемой в запросе.
Mapping	Предоставляет модель <code>MetaModel</code> , на которой основано отображение.
ObjectTrackingEnabled	Указывает, нужно ли отслеживать изменения в объектах внутри базы данных для транзакционных целей. Если вы имеете дело с базой, доступной только для чтения, то должны установить это свойство в <code>false</code> .
Transaction	Позволяет специфицировать локальную транзакцию в базе данных.

Объект `Table<TEntity>`

Объект `Table<TEntity>` — это представление таблиц базы данных, с которыми проводится работа. Например, вы видели использование класса `Product` — экземпляр `Table<Product>`. Как будет показано далее в этой главе, в объекте `Table<TEntity>` доступно множество методов. Некоторые из этих методов описаны в табл. 56.4.

Таблица 56.4. Некоторые методы `Table<TEntity>`

Метод	Описание
<code>Attach</code>	Позволяет присоединить сущность к экземпляру <code>DataContext</code> .
<code>AttachAll</code>	Позволяет присоединить коллекцию сущностей к экземпляру <code>DataContext</code> .
<code>DeleteAllOnSubmit<TSubEntity></code>	Позволяет привести все отложенные действия в состояние готовности к удалению. Все здесь будет приведено в действие при вызове метода <code>SubmitChanges()</code> из объекта <code>DataContext</code> .
<code>DeleteOnSubmit</code>	Позволяет привести отложенное действие в состояние готовности к удалению. Все будет приведено в действие при вызове метода <code>SubmitChanges()</code> из объекта <code>DataContext</code> .
<code>GetModifiedMembers</code>	Предоставляет массив модифицированных объектов. Вы можете получить их текущие и исходные значения.
<code>GetNewBindingList</code>	Предоставляет новый список для привязки к хранилищу данных.
<code>GetOriginalEntityState</code>	Предоставляет экземпляр объекта в первоначальном состоянии.
<code>InsertAllOnSubmit<TSubEntity></code>	Позволяет привести все отложенные действия в состояние готовности к вставке. Все будет приведено в действие при вызове метода <code>SubmitChanges()</code> из объекта <code>DataContext</code> .
<code>InsertOnSubmit</code>	Позволяет привести отложенное действие в состояние готовности к вставке. Все здесь будет приведено в действие при вызове метода <code>SubmitChanges()</code> из объекта <code>DataContext</code> .

Работа без O/R Designer

Хотя новый инструмент O/R Designer в Visual Studio 2010 существенно облегчает задачу создания всего необходимого для LINQ to SQL, важно отметить, что лежащая в основе этого инструмента технология позволяет сделать все самостоятельно. Это дает возможность более тонко контролировать ситуацию и все, что в действительности происходит.

Создание собственного специального объекта

Чтобы решить ту же задачу, что была описана ранее с таблицей Customers, понадобится представить таблицу Customers в виде класса. Первый шаг — создание нового класса в проекте по имени Customer.cs. Код этого класса показан ниже.

```
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; }
    }
}
```

Фрагмент кода *Customer.cs*

Здесь в файле Customer.cs определен объект Customer, который должен использоваться с LINQ to SQL. Класс имеет атрибут Table, который помечает его как класс таблицы. Атрибут класса Table включает свойство по имени Name, определяющее имя используемой таблицы в базе данных, на которую ссылается строка соединения. Применение атрибута Table также означает необходимость добавления в код ссылки на пространство имен System.Data.Linq.Mapping.

В дополнение к атрибуту Table каждое из определенных в этом классе свойств использует атрибут Column. Как уже упоминалось, в коде столбцы из базы данных SQL Server отображаются на свойства.

Выполнение запросов со специальным классом и LINQ

Имея единственный готовый класс `Customer`, вы получаете возможность запрашивать таблицу `Customers` из базы данных `Northwind`. Соответствующий код находится в том же примере консольного приложения и показан ниже:

```
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; // Применяется для вывода используемого SQL

            Table<Customer> myCustomers = dc.GetTable<Customer>();
            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                    item.CompanyName, item.Country);
            }
            Console.ReadLine();
        }
    }
}
```

В этом случае используется объект `DataContext` по умолчанию, а строка соединения с базой данных `Northwind SQL Server Express` передается в виде параметра. Класс `Table` типа `Customer` затем наполняется с помощью метода `GetTable<TEntity>()`. Для этого примера операция `GetTable<TEntity>()` использует определенный вами специальный класс `Customer`:

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

При этом случится вот что: `LINQ to SQL` применяет объект `DataContext` для создания запроса к базе данных `SQL Server` и получит возвращенные строки в виде строго типизированных объектов `Customer`. Это позволит выполнить итерацию по всем объектам `Customer` в коллекции объекта `Table` и получить необходимую информацию, как это делается в представленном ниже операторе `Console.WriteLine()`:

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

Запуск этого кода выдаст следующий результат:

```
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 snabbköp | Sweden

// Часть результатов опущена

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

Ограничение столбцов, вызванных запросом

Вы заметите, что запрос извлек все столбцы, специфицированные в файле класса `Customer`. Если удалить ненужные столбцы, получится новый класс `Customer`, показанный ниже:

```
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; }
    }
}
```

Фрагмент кода *Customer.cs*

В этом случае удалены все столбцы, не используемые приложением. Если теперь запустить консольное приложение и просмотреть сгенерированный SQL-запрос, можно увидеть следующий результат:

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

В запросе к таблице `Customers` используются только три столбца, определенные в классе `Customer`.

Свойство `CustomerID` интересно возможностью определения, что описываемый им столбец является первичным ключом для таблицы, благодаря установке `IsPrimaryKey` в атрибуте `Column`. Эта установка принимает булевское значение и в данном случае равна `true`.

Работа с именами столбцов

Другой важный момент, связанный со столбцами, заключается в том, что имя свойства, которое определяется в классе `Customer`, должно совпадать с именем, используемым в базе данных. Например, если вы измените имя свойства `CustomerID` на `MyCustomerID`, то получите следующее исключение при попытке запустить консольное приложение:

```
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"
```

Чтобы обойти эту проблему, потребуется определить имя столбца в специальном классе Customer. Это можно сделать с использованием атрибута Column, как показано ниже:

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

Подобно атрибуту Table, атрибут Column включает свойство Name, позволяющее указать имя столбца, как оно выглядит в таблице Customers. Если сделать так, это сгенерирует запрос следующего вида:

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

Это также означает, что теперь нужно ссылаться на столбец, используя имя MyCustomerID (например, item.MyCustomerID).

Создание собственного объекта DataContext

Теперь применение простого объекта DataContext будет, вероятно, не самым лучшим подходом, поскольку вы обнаружите, что более высокую степень управления обеспечит создание собственного класса DataContext. Чтобы решить эту задачу, создайте новый класс по имени MyNorthwindDataContext и унаследуйте его от DataContext. Этот класс в его простейшей форме показан ниже.

```
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")
        {
        }
    }
}
```

Фрагмент кода *MyNorthwindDataContext.cs*

Здесь класс MyNorthwindDataContext наследуется от DataContext и предоставляет экземпляр объекта Table<Customer> от созданного ранее класса Customer. Наличие конструктора — еще одно требование к такому классу. Этот конструктор использует конструктор базового класса для инициализации нового экземпляра объекта, ссылающегося на файл (в данном случае соединение устанавливается с файлом базы данных SQL).

Использование собственного объекта DataContext теперь позволяет изменить код приложения следующим образом.

```
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();
}
}
}

```

Фрагмент кода *ConsoleApplication1.sln*

Создав экземпляр объекта `MyNorthwindDataContext`, вы теперь можете поручить ему управление соединением с базой данных. Также вы заметите, что теперь имеете прямой доступ к классу `Customer` через оператор `dc.Customers`.



Обратите внимание, что примеры, приведенные в этой главе — это просто скелеты, в том смысле, что в них отсутствует обработка ошибок и протоколирование, которые обычно обязательны при построении реальных приложений. Это сделано для того, чтобы проиллюстрировать моменты, обсуждаемые в этой главе, и ничего более.

Специальные объекты и O/R Designer

В дополнение к построению специального объекта в собственном файле `.cs` с последующим встраиванием этого класса в созданный вами `DataContext`, для построения файлов классов можно также использовать O/R Designer из Visual Studio 2010. В этом случае соответствующий файл `.cs` создается автоматически, а O/R Designer предлагает визуальное представление файла класса и любых возможных отношений, которые были установлены.

При просмотре в O/R Designer представления файла `.dbml` вы заметите, что в панели инструментов присутствуют три элемента — Class (Класс), Association (Ассоциация) и Inheritance (Наследование).

Для примера перетащите на поверхность проектирования объект `Class` из панели инструментов. Вы увидите графическое представление обобщенного класса, как показано на рис. 56.7.



Рис. 56.7. Графическое представление обобщенного класса

Теперь можно щелкнуть на имени `Class1` и переименовать его в `Customer`. Щелчок правой кнопкой мыши рядом с именем позволит добавить свойства к файлу класса, для чего нужно выбрать в контекстном меню пункт **Add Property** (Добавить свойство). Добавьте в класс `Customer` три свойства — `CustomerID`, `CompanyName` и `Country`. Выделив свойство `CustomerID`, можно сконфигурировать его в диалоговом окне **Properties** (Свойства) в Visual Studio и изменить значение **Primary Key** (Первичный ключ) с `False` на `True`. Также понадобится выделить весь класс, перейти в диалоговое окно **Properties** и изменить свойство **Source** (Источник) на `Customers`, потому что так выглядит имя таблицы, с которой будет работать объект `Customer`. После того, как все это будет сделано, вы получите визуальное представление класса, показанное на рис. 56.8.

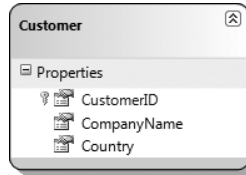


Рис. 56.8. Визуальное представление класса Customer

На рисунке видно, что свойство CustomerID представлено с пиктограммой первичного ключа рядом с названием. Щелкнув на значке + рядом с файлом Northwind.dbml, можно увидеть там два файла — Northwind.dbml.layout и Northwind.designer.cs. Файл Northwind.dbml.layout — это XML-файл, который помогает Visual Studio в визуальном представлении, отображаемом в O/R Designer. Но наиболее важным файлом является Northwind.designer.cs. В этом файле содержится код созданного класса. Открыв этот файл, вы увидите, что именно сгенерировала среда Visual Studio.

Для начала вы найдете класс Customer в коде страницы:

```
[Table(Name="Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    // Код не показан
}
```

Customer — имя класса, заданное в визуальном конструкторе. Класс оснащен атрибутом Table, в котором указано имя Customers, потому что таково имя объекта базы данных Northwind, с которым должен работать данный объект.

Внутри класса Customer находятся три определенных вами свойства. Вот одно из них, CustomerID:

```

[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();
        }
    }
}
```

Фрагмент кода *Customer.cs*

По аналогии с тем, как вы строили для себя класс в ранее приведенном примере, свойства определены с атрибутом Column, для которого доступно несколько собственных свойств. Как видите, первичный ключ определен установкой IsPrimary.

В дополнение к классу Customer вы обнаружите в созданном файле еще один класс, унаследованный от DataContext:

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext : System.Data.Linq.DataContext
{
    // Код не показан
}
```

Объект `DataContext` — `NorthwindDataContext` — позволяет подключаться к базе данных Northwind и получать доступ к таблице `Customers`, как это было и в предыдущих примерах. Вы обнаружите, что использование инструмента O/R Designer упрощает и облегчает создание объекта базы данных. Однако в то же время, если нужен полный контроль, можете кодировать все самостоятельно и получить необходимый результат.

Запрос к базе данных

Как вы уже видели, существует много способов выполнения запроса к базе данных в коде приложения. В некоторых наиболее простых формах запросы выглядят следующим образом:

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

Эта команда извлекает всю таблицу `Products` в экземпляр объекта запроса.

Использование выражений запросов

В дополнение к прямому извлечению всей таблицы посредством `dc.Products` можно также использовать выражение запроса непосредственно в коде, которое также строго типизировано. Ниже приведен пример.

```
⬇ 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();
           }
       }
   }
```

Фрагмент кода `ConsoleApplication1.sln`

В этом случае объект `query` (типа `Table<Product>`) наполняется значением запроса `from p in dc.Products select p`; . Эта команда, хотя и показана в двух строках для лучшей читабельности, при желании также может быть представлена в одной строке.

Выражения запроса

Существует множество выражений запросов, которые можно использовать в коде. В предыдущем примере применялся простой оператор `select`, возвращающий всю таблицу. В табл. 56.5 перечислены другие выражения запросов, которые находятся в вашем распоряжении.

Таблица 56.5. Некоторые доступные выражения запросов

Группа	Синтаксис
Project	<code>select <выражение></code>
Filter	<code>where <выражение>, distinct</code>
Test	<code>any(<выражение>), all(<выражение>)</code>
Join	<code><выражение> join <выражение> on <выражение> equals <выражение></code>
Group	<code>group by <выражение>, into <выражение>, <выражение> group join <решение> on <выражение> equals <выражение> into <выражение></code>
Aggregate	<code>count([<выражение>]), sum(<выражение>), min(<выражение>), max(<выражение>), avg(<выражение>)</code>
Partition	<code>skip [while] <выражение>, take [while] <выражение></code>
Set	<code>union, intersect, except</code>
Order	<code>order by <выражение>, <выражение> [ascending descending]</code>

Фильтрация с использованием выражений

В дополнение к прямым запросам ко всей таблице можно фильтровать элементы, используя конструкции `where` и `distinct`. Следующий пример содержит запрос определенного типа записей из таблицы `Products`:

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

В этом случае запрос выбирает все записи таблицы `Products`, описывающие продукты, которые начинаются с буквы “L”. Это делается с помощью выражения `p.ProductName.StartsWith("L")`. Для свойства `ProductName` доступен широкий выбор методов, которые позволяют тонко настраивать условие фильтрации должным образом. Приведенная ниже операция дает следующий результат:

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

В список можно добавить столько подобных выражений, сколько нужно. Например, ниже показан пример добавления двух конструкций `where` к запросу:

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

В этом случае имеется выражение фильтра, которое ищет элементы с именем продукта, начинающимся с буквы “L”, за которым следует второе выражение, применяющее второй критерий, который утверждает, что элементы также должны заканчиваться на букву “i”. Это должно дать следующий результат:

```
76 | Lakkalikööri
```

Выполнение соединений

В дополнение к работе с одной таблицей можно также работать с несколькими и выполнять соединения (join) в запросах. Перетащите на поверхность проектирования Northwind.dbml две таблицы Customers и Orders. Результат показан на рис. 56.9.

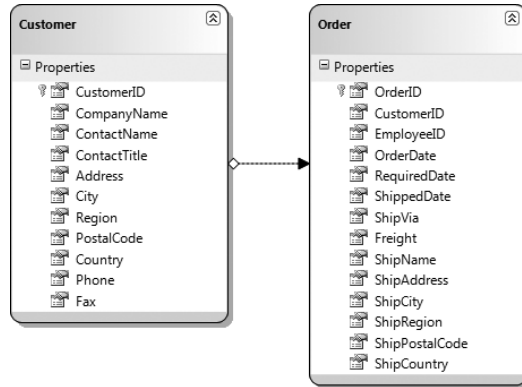


Рис. 56.9. Создание отношения между двумя таблицами

На этом рисунке видно, что после перетаскивания обоих элементов на поверхность проектирования среда Visual Studio будет знать, что между ними есть отношение, и создаст его представление в коде, а также представит с помощью стрелки в визуальном конструкторе.

В результате в запросе можно будет использовать операцию join для работы с обеими таблицами, как показано в следующем примере:

```

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();
        }
    }
}

```

Фрагмент кода *ConsoleApplication1.sln*

В этом примере данные извлекаются из таблицы Customers и соединяются с таблицей Orders по столбцу CustomersID. Это делается с помощью операции join:

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

Здесь оператором select new создается новый объект, который включает в себя столбцы CustomerID, CompanyName и Country из таблицы Customers, а также столбцы OrderID и OrderDate из таблицы Orders.

В итерации по коллекции этого нового объекта интересно то, что в операторе foreach также используется ключевое слово var, потому что в этот момент времени тип неизвестен:

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

Тем не менее, объект item здесь имеет доступ ко всем указанным свойствам. Ниже показано, как выглядит результат выполнения этого примера:

```
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
```

Группирование элементов

В запросах можно легко объединять элементы в группы. В рассматриваемом примере Northwind.dbml перетащите на поверхность проектирования таблицу Categories, и вы увидите, что существует отношение между этой таблицей и таблицей Products, добавленной ранее. В следующем коде показано, каким образом сгруппировать продукты по категориям.

```

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();
        }
    }
}

```

Фрагмент кода *ConsoleApplication1.sln*

В этом примере создается новый объект, представляющий собой группу категорий, который упаковывает всю таблицу `Products` в новую таблицу по имени `g`. Перед этим категории упорядочиваются по имени с использованием операции `orderby`, потому что в качестве порядка указан `ascending` (другой возможный вариант — `descending`). В выводе получается `Category` (переданный через свойство `Key`) и экземпляр `Product`. Одна итерация оператором `foreach` делается по одному разу для каждой категории, а другая — по каждому из продуктов в категории.

Ниже показана часть вывода этого кода.

```
Beverages
  Chai
  Chang
  Guaraná Fantástica
  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
  Vegie-spread
  Louisiana Fiery Hot Pepper Sauce
  Louisiana Hot Spiced Okra
  Original Frankfurter grüne Sobe
```

Помимо описанных в этой короткой главе имеется намного больше доступных команд и выражений.

Хранимые процедуры


До сих пор мы запрашивали информацию из таблиц непосредственно, поручая LINQ создание соответствующих операторов SQL для выполнения нужных операций. При работе с предварительно созданной базой данных, которая интенсивно использует хранимые процедуры, если вы хотите следовать передовой практике применения хранимых процедур в базе данных, то и для этого в LINQ предусмотрены соответствующие средства.

LINQ to SQL трактует работу с хранимыми процедурами как вызовы методов. Как было показано на рис. 56.4, существует инструмент проектирования O/R Designer, позволяющий перетаскивать на поверхность проектирования таблицы, чтобы затем была возможность программно работать с ними. В правой части O/R Designer вы найдете место, куда можно перетаскивать хранимые процедуры.

Любая хранимая процедура, которую вы перетащите в эту часть O/R Designer, превращается в доступный метод объекта `DataContext`. В рассматриваемом примере перетащите в эту часть O/R Designer хранимую процедуру `TenMostExpensiveProducts`.

В следующем коде показано, как вызывать эту хранимую процедуру в базе данных Northwind.

```

 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();
        }
    }
}

```

Фрагмент кода *ConsoleApplication1.sln*

В коде видно, что строки, поступившие из хранимой процедуры, накапливаются в объекте `ISingleResult<Ten_Most_Expensive_ProductsResult>`. Итерация по этому объекту уже не представляет никакой сложности.

На примере кода вы удостоверились, что вызов хранимой процедуры в LINQ to SQL — довольно простая задача.

Резюме

Одним из наиболее впечатляющих средств .NET Framework 4 являются возможности LINQ, которые предлагает эта платформа. В настоящей главе внимание было сосредоточено на использовании LINQ to SQL и некоторых доступных опций для выполнения запросов к базам данных SQL Server.

Применение LINQ to SQL позволяет иметь строго типизированный набор для выполнения операций CRUD над базой данных. Однако помимо этого можно также по-прежнему использовать ранее существовавшие средства доступа, будь то взаимодействие с ADO.NET или работа с хранимыми процедурами.