ГЛАВА 13

LINQ

Одним из наиболее впечатляющих новшеств .NET 3.5 является LINQ (Language Integrated Query — язык интегрированных запросов) — набор расширений языка, позволяющий выполнять запросы, не покидая привычного комфортного окружения языка C#.

В своем простейшем виде LINQ определяет ключевые слова, которые вы используете для построения выражений. Эти выражения запросов могут выбирать, фильтровать, сортировать, группировать и трансформировать данные. Различные расширения LINQ позволяют вам использовать одни и те же выражения запросов с разными источниками данных. Например, *LINQ to Objects* — простейшая форма LINQ — позволяет формулировать и осуществлять запросы к коллекциям и объектам, находящимся в памяти. *LINQ to DataSet* выполняет ту же работу над находящимися в памяти объектами `DataSet`. Еще более интересны два других воплощения LINQ, позволяющие обращаться к внешним данным: *LINQ to SQL*, которое позволяет выполнять запросы к базе данных SQL Server без написания кода доступа к данным, и *LINQ to XML*, позволяющее читать файлы XML, не используя специализированных классов .NET для работы с XML.

LINQ — тесно интегрированная часть .NET 3.5, языка C# и Visual Basic 2008. Однако он не является специфичным средством ASP.NET, и может применяться в приложениях .NET любого типа — от инструментов командной строки до "толстых" клиентов Windows. Хотя вы можете применять LINQ где угодно, в приложении ASP.NET, скорее всего, вы используете LINQ как часть компонента базы данных. Вы можете использовать LINQ в дополнение к коду доступа к данным ADO.NET, или же — с помощью LINQ to SQL — вместо него.

Эта глава даст вам представление о LINQ с точки зрения Web-разработчика. Вы узнаете, как использовать LINQ для ваших страниц ASP.NET, и узнаете, где LINQ поможет усовершенствовать другие подходы к доступу к данным (и где нет). Начнем с рассмотрения сущности LINQ to Objects и LINQ to DataSet, что обеспечит вам большую гибкость в создании пользовательских представлений ваших данных. Значительная часть этой главы будет посвящена рассмотрению LINQ to SQL, предоставляющему более высокоуровневую модель запросов и обновлений базы данных. Вы узнаете, как работает LINQ to SQL, как он вписывается в типичное Web-приложение, и как он может стать практической заменой более традиционного кода ADO.NET. Также вы изучите, как использовать элемент управления `LinqDataSource`, позволяющий создавать неожиданно сложные страницы, связанные с данными, без необходимости в написании какого-либо кода доступа к данным или запросов SQL.

На заметку! Еще об одной разновидности LINQ — LINQ to XML — вы узнаете из главы 14.

Основы LINQ

Простейший подход к LINQ заключается в рассмотрении его работы с коллекциями, находящимися в памяти. Речь идет об LINQ to Objects — простейшей форме LINQ.

По сути LINQ to Objects позволяет вам заменить итеративную логику (такую как блок `foreach`) декларативным выражением LINQ. Например, предположим, что вы хотите получить список всех сотрудников, чьи фамилии начинаются с буквы `D`. Используя функциональный код C#, вы можете запустить цикл по всей коллекции сотрудников и добавлять каждого подходящего сотрудника во вторую коллекцию, как показано ниже:

```
// Получить полную коллекцию сотрудников от вспомогательного метода.
List<EmployeeDetails> employees = db.GetEmployees();
// Найти подходящих сотрудников.
List<EmployeeDetails> matches = new List<EmployeeDetails>();
foreach (EmployeeDetails employee in employees)
{
   if (employee.LastName.StartsWith("D"))
   {
      matches.Add(employee);
   }
}
```

Затем вы можете выполнить другую задачу с коллекцией совпадений, или же отобразить ее на Web-странице, как показано ниже:

```
gridEmployees.DataSource = matches;
gridEmployees.DataBind();
```

Ту же задачу можно решить с помощью выражения LINQ. Следующий пример показывает, как можно переписать код, заменив блок `foreach` запросом LINQ:

```
List<EmployeeDetails> employees = db.GetEmployees();
IEnumerable<EmployeeDetails> matches;
matches = from employee in employees
          where employee.LastName.StartsWith("D")
          select employee;
gridEmployees.DataSource = matches;
gridEmployees.DataBind();
```

Запрос LINQ использует набор новых ключевых слов, включая `from`, `in`, `where` и `select`. Он возвращает новую коллекцию, содержащую только отвечающие условию запроса результаты.

Конечный результат этого кода тот же самый — вы получаете коллекцию по имени `matches`, которая заполнена сотрудниками, чьи фамилии начинаются на *D*, и которая затем отображается в сетке (рис. 13.1). Однако имеется некоторое отличие в реализации, в чем вы убедитесь в следующем разделе.

На заметку! Ключевые слова LINQ — действительная часть языка C#. Этот факт отличает LINQ от технологий вроде Embedded SQL, которые требуют от вас переключения между синтаксисом C и синтаксисом SQL в блоке кода.

Отложенное выполнение

Очевидная разница между подходом `foreach` и кодом, использующим выражение LINQ, заключается в способе типизации коллекции `matches`. В коде `foreach` она создается как коллекция специального типа — в данном случае строго типизированный `List<T>`. В примере с LINQ коллекция `matches` представляется только через интерфейс `IEnumerable<T>`, который она реализует.

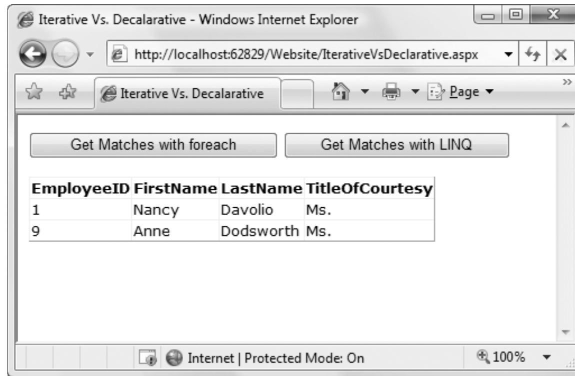


Рис. 13.1. Фильтрация списка сотрудников посредством LINQ

Это отличие продиктовано тем, что LINQ использует *отложенное выполнение*. В противоположность тому, чего вы можете ожидать, объект `matches` не является просто коллекцией, содержащей соответствующие объекты `EmployeeDetails`. Вместо этого он представляет собой специализированный объект LINQ, обладающий способностью извлекать данные тогда, когда вы в них нуждаетесь.

В предыдущем примере объект `matches` — это экземпляр `WhereIterator<T>`, который представляет собой приватный класс, вложенный внутрь класса `System.Linq.Enumerable`. В зависимости от используемого специфического выражения запроса, выражение LINQ может возвращать разные объекты. Например, объединенное выражение, комбинирующее данные из двух разных коллекций, возвратит экземпляр приватного класса `UnionIterator<T>`. Если же вы упростите выражение, исключив из него конструкцию `where`, то получите простой `SelectIterator<T>`.

Совет. Вам не нужно знать, какой именно конкретный класс итератора использует ваш код, потому что вы взаимодействуете с результатами через интерфейс `IEnumerable<T>`. Но если вы любопытны, можете определить тип объекта во время выполнения, используя отладчик Visual Studio (просто наведя курсор на переменную в режиме прерванного выполнения).

Объекты итераторов LINQ добавляют дополнительный слой между определением выражения LINQ и выполнением его. Как только вы осуществляете шаг итерации через итератор LINQ, подобный `WhereIterator<T>`, он извлекает необходимые ему данные. Например, если вы пишете блок `foreach`, который проходит по коллекции `matches`, это действие заставляет вычисляться выражение LINQ.

В предыдущем примере вообще не используется цикл `foreach`, потому что он полагается на привязку данных ASP.NET. Однако на заднем плане поведение точно такое же. Когда вы вызываете метод `GridView.DataBind()`, ASP.NET выполняет итерацию по коллекции `matches` для получения необходимых данных и передает их в `GridView`. Этот шаг инициирует вычисление выражения LINQ точно так же, как если бы вы вручную выполняли итерацию по `matches`.

В зависимости от точного типа выражения, LINQ может выполняться целиком за один шаг, или же часть за частью — по мере итерации. В предыдущем примере данные могут быть извлечены по частям, но если бы вы извлекали информацию из базы данных либо применили порядок сортировки к результатам, то LINQ использовал бы другую стратегию и получил бы все результаты в начале вашего цикла.

На заметку! Не существует технических причин того, что LINQ применяет отложенное выполнение, но есть много причин того, что это — хороший подход. Во многих случаях он позволяет LINQ использовать технику оптимизации производительности, которая в противном случае была бы невозможной. Например, когда используются отношения базы данных с LINQ to SQL, вы можете избежать загрузки связанных данных, которые в действительности не используете.

Как работает LINQ

Подведем краткий итог основ LINQ, которые уже вам известны.

- Для использования LINQ вы создаете выражение LINQ. Правила построения таких выражений вы увидите позднее.
- Возвращаемое значение выражения LINQ является объектом итератора, реализующего `IEnumerable<T>`.
- Когда вы перечисляете элементы объекта итератора, LINQ выполняет свою работу.

Это вызывает один хороший вопрос, а именно: как же LINQ выполняет выражение? Какую именно работу он проделывает, чтобы выдать ваш отфильтрованный результат? Ответ зависит от типа запрашиваемых данных. Например, LINQ to SQL преобразует выражения LINQ в команды базы данных. В результате LINQ to SQL требуется открыть соединение и выполнить запрос базы данных, чтобы получить необходимые данные.

Если вы используете LINQ to Objects, как в предыдущем примере, то процесс, выполняемый LINQ намного проще. Фактически в этом случае LINQ просто использует цикл `foreach` для сканирования коллекции, проходя последовательно от ее начала до конца. Хотя это не звучит так уж впечатляюще, действительное преимущество LINQ заключается в том, что он предоставляет гибкий способ определения запросов, которые могут быть применены к широкому диапазону различных источников данных. Как вы уже знаете, .NET Framework позволяет применять выражения LINQ для запросов к коллекциям, находящимся в памяти, к объектам `DataSet`, к документам XML и (что наиболее полезно) к базам данных SQL Server. Однако независимые разработчики (и будущие версии .NET) могут добавлять собственные поставщики LINQ, поддерживающие тот же синтаксис выражений, но работающие с другими источниками данных. Им просто нужно транслировать выражения LINQ в соответствующие последовательности низкоуровневых шагов. Возможные примеры включают поставщиков LINQ, опрашивающих файловую систему, другие базы данных, службы каталогов, такие как LDAP, и т.д.

На заметку! Код, используемый LINQ to Objects для извлечения данных, почти всегда работает медленнее, чем написанный вручную соответствующий блок `foreach`. Часть накладных расходов обусловлена тем фактом, что здесь задействованы дополнительные делегаты и вызовы методов (как вы увидите далее в этой главе). Однако чрезвычайно маловероятно, что манипуляции с объектами в памяти станут узким местом в производительности приложения серверной стороны, каковым является Web-сайт ASP.NET. Вместо этого такие задачи, как соединение с базой данных, подключение к Web-службе или извлечение информации из файловой системы, происходят на несколько порядков медленнее, и намного вероятнее, что они сильнее отразятся на общей производительности. В результате редко имеет смысл из соображений производительности избегать применения LINQ to Objects. Единственное исключение — когда вы хотите реализовать более совершенный механизм поиска. Например, поиск, которые проходит по огромной коллекции упорядоченной информации, используя индекс, может быть более эффективным, чем запрос LINQ, который выполняет сканирование по всему набору данных от начала до конца.

В LINQ присутствует важная симметрия. Выражения LINQ работают с объектами, реализующими IEnumerable<T> (такими как коллекция List<EmployeeDetails> из предыдущего примера), и возвращают выражения LINQ объекты, реализующие Enumerable<T> (как WhereIterator<T> из предыдущего примера). Поэтому вы можете передать результат одного выражения LINQ другому выражению LINQ и т.д. Такая цепочка выражений LINQ вычисляется в конце, когда выполняется итерация по конечным данным. В зависимости от типа источника данных, который вы опрашиваете, LINQ часто может "сплавить" вашу цепочку выражений в одну операцию, и таким образом, выполнить ее наиболее эффективным способом.

Выражения LINQ

Прежде чем продолжить изучение LINQ далее, вы должны понять, как составляется выражение LINQ. Выражения LINQ обладают внешним сходством с запросами SQL, хотя порядок составляющих их конструкций изменен.

Все выражения LINQ должны иметь конструкцию from, задающую источник данных, и конструкцию select, указывающую данные, которые вы хотите получить (или group, определяющую группы, в которые следует поместить извлекаемые данные). Конструкция from идет первой:

```
matches = from employee in employees
...;
```

Конструкция from идентифицирует две части информации (выделенные полужирным в приведенном коде). Слово, следующее немедленно за *in*, идентифицирует источник данных, в данном случае — объект коллекции по имени employees, содержащий экземпляры EmployeeDetails. Слово, которое находится сразу после *from*, назначает псевдоним, представляющий индивидуальные элементы из источника данных. Для текущего выражения каждый EmployeeDetails называется employee. Вы можете позднее использовать этот псевдоним при построении других частей выражения, таких как конструкции выборки и фильтрации.

Рассмотрим простейший из возможных запрос LINQ. Он просто извлекает полный набор данных из коллекции employees:

```
IEnumerable<EmployeeDetails> matches;
matches = from employee in employees
        select employee;
```

Язык C# включает еще много других операций LINQ, которые невозможно рассмотреть во всех подробностях в нашей книге. (Эта глава дает лишь общий обзор LINQ, а также в ней более подробно рассматриваются некоторые аспекты программирования LINQ, которые представляют особый интерес для Web-разработчиков, такой как LINQ to SQL.) В последующих разделах мы коснемся наиболее важных операций, включая select, where, orderby и group. Вы можете просмотреть полный список операций LINQ в справочной системе .NET Framework. Также вы найдете широкое разнообразие примеров выражений на странице "Microsoft's 101 LINQ Samples" по адресу http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx.

Проекции

Конструкцию select можно изменить, чтобы получить подмножество данных. Например, вот как можно получить список имен сотрудников:

```
IEnumerable<string> matches;
matches = from employee in employees
          select employee.FirstName;
```

или список строк, включающих имена и фамилии:

```
matches = from employee in employees
        select employee.FirstName + employee.LastName;
```

Как видите, вы можете использовать стандартные операции C# для числовых данных и строк для модификации информации во время выборки. Что еще интереснее, так это то, что вы можете динамически определить новый класс-оболочку только для той информации, которую вы хотите вернуть. Например, если вы хотите получить и имя и фамилию, но в отдельных строках, вы можете создать усеченную версию класса `EmployeeDetails`, состоящую только из свойств `FirstName` и `LastName`. Чтобы сделать это, вы используете средство C# 2008, называемое *анонимными типами*. Базовая техника заключается в добавлении ключевого слова `new` к `select`, и присваивании каждого свойства, которое вы хотите создать, в терминах извлекаемого объекта. Вот пример:

```
var matches = from employee in employees
           select new {First = employee.FirstName, Last = employee.LastName};
```

Это выражение при выполнении вернет набор объектов неявно созданного класса. Каждый объект имеет два свойства: `First` и `Last`. Вы нигде не увидите определение этого класса, потому что он генерируется компилятором и получает бессмысленное, автоматически сгенерированное имя. Однако вы можете локально использовать этот класс, обращаясь к свойствам `First` и `Last`, и даже использовать привязку данных (в этом случае ASP.NET извлекает соответствующие значения по имени свойства, используя рефлексию). Способность трансформировать запрашиваемые данные в результаты с отличающейся структурой называется *проекцией*.

В работе этого примера присутствует один трюк. Как вы уже знаете, выражение LINQ возвращает объект-итератор. Класс итератора является обобщенным, а это значит, что он привязан к определенному типу элементов, в данном случае к анонимному классу, имеющему два свойства — `First` и `Last`. Однако поскольку вы не определяете этот класс, вы не можете определить корректную ссылку `IEnumerator<T>`. Решение состоит в использовании ключевого слова `var`.

На рис. 13.2 показан результат привязки коллекции `matches` к сетке.

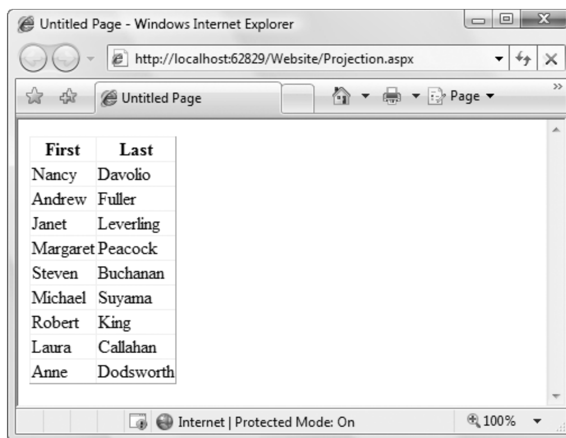


Рис. 13.2. Проекция данных на новое представление

Ключевое слово `var` также нужно применять всякий раз, когда вы хотите сослаться на индивидуальный объект. Один пример — когда выполняется код итерации по набору, возвращенному предыдущим выражением LINQ:

```
foreach (var employee in matches)
{
   // (Что-то делать с employee.First и employee.Last.)
}
```

Напомним, что ключевое слово var разрешается во время компиляции и не может быть использовано как переменная — член класса. В результате такой подход не дает возможности передавать экземпляр анонимного класса между методами.

Совет. Ключевое слово var удобно, даже если вы не используете анонимные типы. В этом случае оно служит просто сокращением, которое избавляет вас от необходимости написания полного имени типа IEnumerable<T>.

Конечно, вам не обязательно использовать анонимные типы при выполнении проекции. Вы можете определить тип формально и затем использовать его в выражении. Например, если вы создадите следующий класс EmployeeName:

```
public class EmployeeName
{
   public string FirstName
   { get; set; }

   public string LastName
   { get; set; }
}
```

то сможете заменить объекты EmployeeDetails на EmployeeName в вашем выражении запроса:

```
IEnumerable<EmployeeName> matches = from employee in employees
  select new EmployeeName {FirstName = employee.FirstName,
  LastName = employee.LastName};
```

Это выражение запроса работает потому, что свойства FirstName и LastName являются общедоступными как для чтения, так и для записи. После создания объекта EmployeeName LINQ устанавливает эти свойства. Альтернативно вы могли бы добавить набор скобок после класса EmployeeName и указать аргументы для параметризованного конструктора:

```
IEnumerable<EmployeeName> matches = from employee in employees
  select new EmployeeName(FirstName, LastName);
```

Фильтрация и сортировка

В первом примере LINQ в этой главе вы видели, как конструкция where позволяет фильтровать результаты, чтобы получить только те, что соответствуют определенному условию. Например, приведенный ниже код служит для нахождения сотрудников, фамилии которых начинаются с определенной буквы:

```
IEnumerable<EmployeeDetails> matches;
matches = from employee in employees
          where employee.LastName.StartsWith("D")
          select employee;
```

Конструкция where принимает условное выражение, вычисляемое для каждого элемента. Если оно истинно, элемент включается в результат. Однако и в этом случае LINQ сохраняет ту же модель отложенного выполнения, а это значит, что конструкция where не вычисляется до тех пор, пока вы действительно не попытаетесь выполнить итерацию по результату.

Как вы, вероятно, догадались, можно комбинировать множество условных выражений с операциями “И” (&&) и “ИЛИ” (||), а также применять операции сравнения (такие как <, <=, > и >=). Например, вы можете создать следующий запрос, чтобы отфильтровать продукты дороже определенной пороговой цены:

```
IEnumerable<Product> matches;
matches = from product in products
          where product.UnitsInStock > 0 && product.UnitPrice > 3.00M
          select product;
```

Одно интересное свойство выражения LINQ заключается в том, что вы можете встраивать в него свои собственные методы. Например, вы можете создать функцию по имени `TestEmployee()`, которая проверяет сотрудника и возвращает `true` или `false` в зависимости от того, нужно его включать в результат или нет:

```
private bool TestEmployee(EmployeeDetails employee)
{
   return employee.LastName.StartsWith("D")
}
```

Затем вы можете использовать метод `TestEmployee()` следующим образом:

```
IEnumerable<EmployeeDetails> matches;
matches = from employee in employees
          where TestEmployee(employee)
          select employee;
```

Операция `orderby` столь же очевидна. Она моделирует синтаксис оператора `SELECT` из SQL. Вы просто предоставляете список из одного или более значений для использования в сортировке, разделяя их запятыми. Вы можете добавить слово `decending` после имени поля, чтобы сортировать в порядке убывания.

Вот пример сортировки:

```
IEnumerable<EmployeeDetails> matches;
matches = from employee in employees
          orderby employee.LastName, employee.FirstName
          select employee;
```

На заметку! Сортировка поддерживается любыми типами, реализующими `IComparable`, а к ним относится большинство основных типов данных .NET (числовые, даты и строки). Можно сортировать, используя данные, не реализующие `IComparable`, но тогда необходимо применять явный синтаксис, описанный в следующем разделе. Таким образом, вы можете передать пользовательский объект `IComparer`, который будет использован для сортировки данных.

Группировка и агрегирование

Группировка позволяет сжать большой набор информации в минимальный набор суммарных результатов.

Группировка является разновидностью проекции, потому что объекты в результирующей коллекции отличаются от объектов в исходной коллекции. Например, предположим, что вы имеете дело с коллекцией объектов `Product` и решаете разбить их на ценовые группы, каждая из которых представит подмножество продуктов определенного ценового диапазона. Каждая группа реализует интерфейс `IGrouping<T, K>` из пространства имен `System.Linq`.

Чтобы использовать группировку, вам нужно принять два решения. Во-первых, вы должны решить, какой критерий следует применять для создания группы. Во-вторых, нужно решить, какую информацию отображать для каждой группы.

Первая задача проста. Вы используете ключевые слова groupby и into для выбора объектов, подлежащих группировке, того, как определяются группы, и какой псевдоним использовать для ссылки на индивидуальные группы. Ниже приведен пример, работающий с коллекцией объектов EmployeeDetails и группирующий их на основе содержимого поля TitleOfCourtesy (Mr., Ms. и т.д.).

```
var matches = from employee in employees
  group employee by employee.TitleOfCourtesy into g
  ...
```

Совет. Согласно принятому соглашению, группам в выражении LINQ назначается псевдоним g.

Объекты помещаются в одну и ту же группу, когда разделяют некоторую часть данных. Чтобы сгруппировать данные в числовые диапазоны, следует написать вычисление, которое произведет одно и то же значение для каждой группы. Например, если вы хотите группировать продукты по цене в диапазоны вроде 0–50, 50–100, 100–150 и т.д., вы должны написать примерно такое выражение:

```
var matches = from product in products
  group product by (int)(product.UnitPrice / 50) into g
  ...
```

Все продукты дешевле 50 будут иметь ключ группировки 0, все продукты ценой от 50 до 100 получат ключ группировки 1, и т.д.

Как только вы оформите группы, вам нужно будет решить, какую информацию о каждой из них нужно вернуть в результате. Каждая группа представляется вашему коду как объект, реализующий интерфейс IGrouping<T, K>. Например, предыдущее выражение LINQ создает группы типа IGrouping<int, Product>, что означает, что типом ключа группировки и типом элемента будет Product.

Этот интерфейс IGrouping<T, K> предоставляет одно свойство — Key, которое возвращает значение, использованное для создания группы. Например, если вы хотите создать простой список строк, отображающих TitleOfCourtesy каждой группы TitleOfCourtesy, то нужное вам выражение будет выглядеть так:

```
var matches = from employee in employees
  group employee by employee.TitleOfCourtesy into g
  select g.Key;
```

Совет. Вы можете заменить ключевое слово var в этом примере на IEnumerable<string>, потому что конечный результат — список строк (показывающий разные значения TitleOfCourtesy). Однако принято использовать ключевое слово var в группирующих запросах, потому что часто вы применяете проекции и анонимные типы для получения более полезной итоговой информации.

Если вы привяжете это к GridView, то увидите результат, показанный на рис. 13.3.

В качестве альтернативы вы можете вернуть всю группу:

```
var matches = from employee in employees
  group employee
     by employee.TitleOfCourtesy into g
  select g;
```

Здесь не слишком помогает привязка данных, потому что ASP.NET не сможет отобразить что-то полезное о каждой группе.

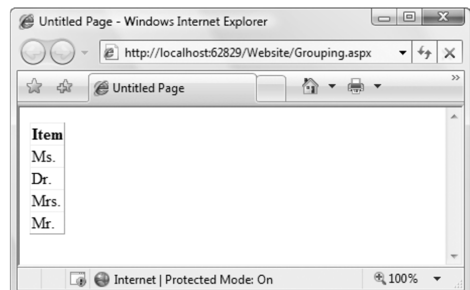


Рис. 13.3. Список групп сотрудников

Однако он предоставит вам возможность выполнять итерацию по каждой группе, используя следующий код:

```
// Цикл по всем группам.
foreach (IGrouping<string, EmployeeDetails> group in matches)
{
    // Цикл по всем объектам EmployeeDetails текущей группы.
    foreach (EmployeeDetails employee in group)
    {
        // Здесь делать что-то с сотрудниками группы
    }
}
```

Это показывает, что даже создав группу, вы имеете возможность обращаться к индивидуальным членам группы.

Но полезнее то, что вы можете использовать агрегатные функции для выполнения вычислений над данными группы. Агрегатные функции LINQ напоминают агрегатные функции базы данных, которые вы, вероятно, использовали ранее, и позволяют вам подсчитывать суммы данных в группе, находить минимум, максимум и среднее значение. Вы можете также фильтровать группы на основе этих вычисляемых значений.

В следующем примере возвращается новый анонимный тип, включающий значение ключа группы и количество объектов в группе. При этом используется встроенный вызов метода по имени `Count()`.

```
var matches = from employee in employees
  group employee by employee.TitleOfCourtesy into g
  select new {Title = g.Key, Employees = g.Count()};
```

Результат показан на рис. 13.4.

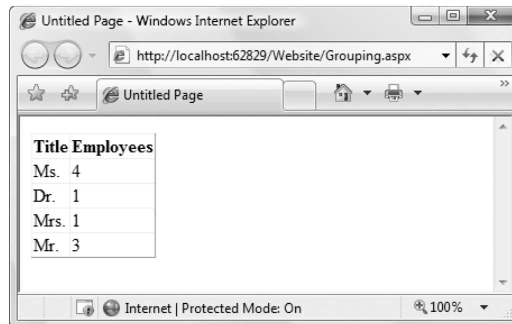


Рис. 13.4. Количество сотрудников в группе

Приведенное выражение LINQ слегка отличается от того, что вы рассматривали до сих пор, потому что использует расширяющий метод. По сути, расширяющие методы — центральная часть функциональности LINQ, которая не представлена выделенными операциями C#. Вместо этого вы должны вызывать метод напрямую. `Count()` — пример расширяющего метода.

Что отличает расширяющие методы от обычных методов — так это тот факт, что расширяющие методы не определены в классе, использующем их. Вместо этого LINQ включает класс `System.Linq.Enumerable`, определяющий несколько десятков расширяющих методов, которые можно вызывать на любом объекте реализующем `IEnumerable<T>`. (Эти расширяющие методы также работают с `IGrouping<T, K>`, потому что он расширяет `IEnumerable<T>`.)

Другими словами, следующая часть предыдущего выражения LINQ:

```
select new {Title = g.Key, Employees = g.Count()};
```

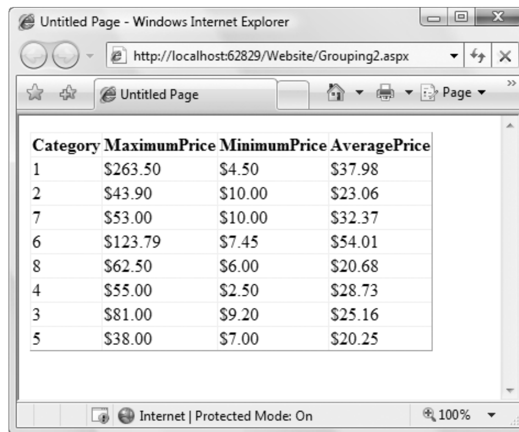
сообщает LINQ, что надо вызвать `System.Linq.Enumerable.Count()`, чтобы вычислить количество элементов в группе.

Наряду с `Count()` в LINQ также определены более мощные расширяющие методы, которые вы можете использовать в сценариях группирования — агрегатные функции `Max()`, `Min()` и `Average()`. Выражения LINQ, использующие эти методы, немного более сложны, потому что они также используют другой средство C#, известное под именем *лямбда-выражений*, позволяющее применять дополнительные параметры к расширяющему методу. В случае методов `Max()`, `Min()` и `Average()` лямбда-выражение позволяет указывать, какое свойство вы хотите использовать в вычислениях.

Ниже приведен пример, использующий эти расширяющие методы для вычисления максимальной, минимальной и средней цены элемента в каждой категории:

```
var categories = from p in products
  group p by p.Category into g
  select new {Category = g.Key,
          MaximumPrice = g.Max(p => p.UnitPrice),
          MinimumPrice = g.Min(p => p.UnitPrice),
          AveragePrice = g.Average(p => p.UnitPrice)};
```

На рис. 13.5 показан результат.



Category	MaximumPrice	MinimumPrice	AveragePrice
1	$263.50	$4.50	$37.98
2	$43.90	$10.00	$23.06
7	$53.00	$10.00	$32.37
6	$123.79	$7.45	$54.01
8	$62.50	$6.00	$20.68
4	$55.00	$2.50	$28.73
3	$81.00	$9.20	$25.16
5	$38.00	$7.00	$20.25

Рис. 13.5. Агрегатная информация о группах продуктов

Хотя этот пример интуитивно понятен, синтаксис лямбда-выражения выглядит несколько необычно. В следующем разделе мы рассмотрим расширяющие методы и лямбда-выражения более подробно.

Внутреннее устройство выражений LINQ

Хотя LINQ использует новые ключевые слова C# (такие как `from`, `in` и `select`), их реализация обеспечивается другими классами. Фактически каждый запрос LINQ транслируется в серию вызовов методов. Вместо того чтобы полагаться на эту трансляцию, вы можете явно самостоятельно вызывать методы. Например, следующее простое выражение LINQ:

```
matches = from employee in employees
select employee;
```

может быть переписано так:

```
matches = employees.Select(employee => employee);
```

Синтаксис здесь выглядит довольно необычно. Это похоже на то, что метод Select() вызывается на коллекции сотрудников. Однако коллекция сотрудников — это обычная коллекция List<T>, и она не включает в себя данного метода. Вместо этого Select() является *расширяющим методом*, который автоматически предоставляется всем классам IEnumerable<T>.

Расширяющие методы

Расширяющие методы — новое средство языка, появившееся в C# 2008. По сути, расширяющие методы позволяют вам определить метод в одном классе, а вызывать его так, будто бы он определен в другом классе. Методы выражений LINQ определены в классе System.Linq.Enumerable, но они могут вызываться с любым объектом IEnumerable<T>.

На заметку! Поскольку расширяющие методы LINQ определены в классе System.Linq.Enumerable, они доступны только в том случае, если этот класс доступен в текущем контексте. Если вы не импортируете пространство имен System.Linq, то не сможете написать явно или неявно никаких выражений LINQ — в этом случае вы получите ошибку компиляции, потому что необходимые методы не будут найдены.

Простейший путь к пониманию этой техники — взглянуть на расширяющий метод. Вот определение расширяющего метода Select() в классе System.Linq.Enumerable:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
  this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{ ... }
```

Расширяющие методы подчиняются небольшому набору правил. Все расширяющие методы должны быть статическими. Расширяющие методы могут возвращать любые типы данных и принимать любое количество параметров. Однако первый параметр — это всегда ссылка на объект, на котором вызывается расширяющий метод (и указывается ключевым словом this). Используемый вами тип данных для этого параметра определяет классы, для которых доступен этот расширяющий метод.

Например, в расширяющем методе Select() первый параметр — IEnumerable<T>:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
  this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

Это говорит о том, что данный расширяющий метод может быть вызван на экземпляре любого класса, реализующего IEnumerable<T> (включая коллекции вроде List<T>). Как видите, метод Select<T> принимает еще один параметр — делегат, используемый для указания выбранного вами подмножества информации. И, наконец, возвращаемое значение метода Select() — это объект IEnumerable<T>; в данном случае это экземпляр приватного класса SelectIterator.

Вот как выглядит полный код, используемый LINQ в методе Enumerable.Select():

```
public static IEnumerable<TResult> Select<TSource, TResult>(
  this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    if (source == null)
    {
        throw new ArgumentNullException("source");
    }
```

```
    if (selector == null)
    {
       throw new ArgumentNullException("selector");
    }
    return SelectIterator<TSource, TResult>(source, selector);
}
```

Лямбда-выражения

Как уже упоминалось, лямбда-выражение — это еще одной новшество синтаксиса C# 2008 в основанных на методах выражениях LINQ. Лямбда-выражение передается методу Select(), как показано ниже:

```
matches = employees.Select(employee => employee);
```

Как вам уже известно, при вызове метода Select() объект сотрудника передается ему в первом параметре. Это — источник запроса. Второй параметр требует делегата, который, указывает на метод. Этот метод выполняет работу по выбору и вызывается по одному разу для каждого экземпляра в коллекции. Метод выборки принимает оригинальное значение (в данном случае — объект сотрудника) и возвращает выбранный результат. Предыдущий пример выполняет наиболее прямолинейную логику выборки — берет оригинальный объект сотрудника и возвращает его без изменений.

В данном примере присутствует некоторая "ловкость рук". Как объяснялось ранее, метод Select() ожидает делегата. Вы можете просто применить обычный делегат, указывающий на именованный метод, который вы создали где-то в вашем классе, но это сделало бы ваш код достаточно громоздким.

Простой подход состоит в применении анонимного метода, что позволяет вам определить метод встроенным образом там, где вы его используете — как аргумент метода Select(). Анонимные методы начинаются со слова delegate, за которым идет объявление сигнатуры метода, а за ним — набор скобок, содержащих код метода. Вот как мог бы выглядеть предыдущий пример, если бы вы использовали анонимный метод:

```
var matches = employees
  .Select(
          delegate(EmployeeDetails employee)
            { return employee; }
         );
```

Лямбда-выражения — простой способ сократить этот код. Лямбда-выражение состоит из двух частей, разделенных символами =>. Первая часть идентифицирует параметры, принимаемые вашим методом. В данном примере лямбда-выражение принимает каждый объект из коллекции и представляет его ссылкой по имени employee. Вторая часть лямбда-выражения определяет значение, которое вы хотите вернуть.

Чтобы лучше это понять, посмотрим, что произойдет, если создать более сложную логику выборки, которая выполняет проекцию. Вы уже видели, что LINQ обеспечивает гибкость в получении только тех свойств, которые вам нужны, или даже в объявлении нового типа. Например, явное выражение LINQ извлекает данные каждого сотрудника и помещает их в новый анонимный тип, включающий только информацию об имени и фамилии:

```
var matches = employees
  .Select(
          delegate(EmployeeDetails employee)
            { return new { First = employee.FirstName,
                           Last = employee.LastName };
            }
          );
```

Теперь вы можете сократить этот код, заменив анонимный метод лямбда-выражением, выполняющим ту же работу:

```
var matches = employees
  .Select(employee =>
        new { First = employee.FirstName, Last = employee.LastName });
```

Составные выражения

Конечно, большинство выражений LINQ более детализированы, чем рассмотренные в данном разделе примеры. Более полное выражение LINQ может добавлять сортировку или фильтрацию, как показано ниже:

```
IEnumerable<EmployeeDetails> matches = from employee in employees
        where employee.LastName.StartsWith("D")
        select employee;
```

Вы можете переписать это выражение со следующим синтаксисом:

```
IEnumerable<EmployeeDetails> matches = employees
          .Where(employee => employee.LastName.StartsWith("D"))
          .Select(employee => employee);
```

Одно замечательное свойство явного синтаксиса LINQ заключается в том, что он проясняет порядок выполнения операций. В предыдущем примере легко увидеть, что все начинается с коллекции сотрудников, затем идет вызов метода `Where()` и, наконец, вызов метода `Select()`. Если вы используете больше операций LINQ, то составите более длинную серию вызовов методов.

Также вы заметите, что метод `Where()` работает подобно методу `Select()`. И `Where()`, и `Select()` — расширяющие методы, и оба используют лямбда-выражения, применяющие простой метод. Метод `Where()` применяет лямбда-выражение, проверяющее каждый элемент и возвращающее `true`, если он должен быть включен в результат. Метод `Select()` применяет лямбда-выражение, трансформирующее каждый элемент данных в нужное вам представление. В классе `System.Linq.Enumerable` вы найдете еще много расширяющих методов, работающих аналогичным образом.

Чаще всего вы будете использовать неявный синтаксис для создания выражений LINQ. Однако может быть немало случаев, когда понадобится явный синтаксис, например, если вам нужно передать параметр в расширяющий метод, который не предусмотрен неявным синтаксисом LINQ. В любом случае понимание того, как выражения отображаются на вызовы методов, как расширяющие методы включаются в объекты `IEnumerable<T>`, и как лямбда-выражения инкапсулируют фильтрацию, сортировку, проекцию и прочие детали, значительно проясняет внутреннюю работу LINQ.

LINQ to DataSet

Как вам известно из главы 8, вы можете использовать метод `DataTable.Select()` для извлечения нескольких интересующих вас записей из `DataTable`, используя SQL-подобное выражение фильтра. Хотя метод `Select()` работает исключительно хорошо, он имеет несколько очевидных ограничений. Во-первых, он основан на строках, а это значит, что возникает риск ошибок, которые не выявляются во время компиляции. Также он ограничен только фильтрацией и не предоставляет других средств, которые представлены операциями LINQ, такими как сортировка, группировка и проекции. Если вам нужно нечто большее, вы можете использовать средства запросов LINQ с `DataTable`.

При использовании LINQ to DataSet вы применяете по сути те же выражения, которые используете для запроса коллекций объектов. В конце концов, `DataSet` — это просто коллекция экземпляров `DataTable`, каждый из которых является коллекцией эк-

земпляров DataRow (вместе с дополнительной информацией о схеме). Однако у DataSet есть одно существенное ограничение — он не предоставляет строго типизированных данных. Вместо этого на вас возлагается обязанность выполнять приведение значений к соответствующим типам. Это представляет некоторую проблему для выражений LINQ, потому что они должны возвращать строго типизированные данные. Другими словами, компилятор нуждается в возможности определения типа данных, возвращаемого вашим выражением LINQ, на этапе компиляции.

Чтобы обеспечить такую возможность, вам понадобится расширяющий метод Field<T>, который предоставлен классом DataRowExtensions из пространства имен System.Data. По сути, метод Field<T> расширяет объект DataRow и предоставляет строго типизированный способ доступа к полю. Ниже приведен пример, в котором используется метод Field<T> во избежание приведения типов при извлечении значения из поля FirstName:

```
string value = dataRow.Field<string>("FirstName");
```

Это не единственное ограничение, которое вам нужно преодолеть в DataSet. Как вы уже знаете, LINQ работает на коллекциях, реализующих IEnumerable<T>. Ни DataTable, ни DataRowCollection не реализуют этот интерфейс — вместо этого DataRowCollection реализует слабо типизированный интерфейс IEnumerable, чего явно недостаточно. Чтобы заполнить этот пробел, вам понадобится другой расширяющий метод по имени AsEnumerable(), предоставляющий коллекцию IEnumerable<T> объектов DataRow для данного DataTable. Метод AsEnumerable() определен в классе DataTableExtensions из пространства имен System.Data.

Для того чтобы получить в свое распоряжение методы Field<T> и AsEnumerable(), убедитесь, что вы импортировали пространство имен System.Data. (Также понадобится ссылка на сборку System.Data.DataSetExtensions.dll, которая автоматически добавляется в файл web.config, ориентированный на .NET 3.5).

Используя DataRowExtensions и DataTableExtensions, вы можете написать выражение LINQ, запрашивающее DataTable в DataSet в соответствие с той же фундаментальной инфраструктурой, что и LINQ to Objects. Приведем пример, извлекающий записи о сотрудниках, чьи фамилии начинаются с буквы "D", как объекты DataRow:

```
DataSet ds = db.GetEmployeesDataSet();
IEnumerable<DataRow> matches = from employee
  in ds.Tables["Employees"].AsEnumerable()
  where employee.Field<string>("LastName").StartsWith("D")
  select employee;
```

Эта коллекция не подходит для привязки данных (если вы привяжете эту коллекцию, то привязываемый элемент управления покажет только общедоступные свойства объекта DataRow вместо коллекции значений полей). Проблема заключается в том, что когда привязываются данные ADO.NET, вам нужно включать схему. Привязка полного DataTable работает потому, что он включает коллекцию Columns с заголовками столбцов и прочей информацией.

Существуют два способа решения этой проблемы. Один из вариантов — применить метод DataTableExtensions.AsDataView() с целью получения DataView для отфильтрованного набора строк:

```
DataSet ds = db.GetEmployeesDataSet();
var matches = from employee in ds.Tables["Employees"].AsEnumerable()
  where employee.Field<string>("LastName").StartsWith("D")
  select employee;
gridEmployees.DataSource = matches.AsDataView();
gridEmployees.DataBind();
```

На заметку! Выражения LINQ to DataSet возвращают экземпляры класса EnumerableRowCollection<T> (который реализует знакомый интерфейс IEnumerable<T>). AsDataView() — расширяющий метод, работающий только с объектами EnumerableRowCollection<T>. В результате вы должны определять переменную matches в предыдущем примере, используя ключевое слово var, или же как EnumerableRowCollection<DataRow>. Если вы объявите ее как IEnumerable<DataRow>, то не получите доступа к методу AsDataView().

Другой столь же эффективный выбор заключается в применении проекции. Например, следующее выражение LINQ помещает детали имен сотрудников в оболочку нового анонимного типа, который может участвовать в привязке:

```
DataSet ds = db.GetEmployeesDataSet();
var matches = from employee in ds.Tables["Employees"].AsEnumerable()
  where employee.Field<string>("LastName").StartsWith("D")
  select new { First = employee.Field<string>("FirstName"),
               Last = employee.Field<string>("LastName") };
gridEmployees.DataSource = matches;
gridEmployees.DataBind();
```

На рис. 13.6 показан довольно скромный результат.

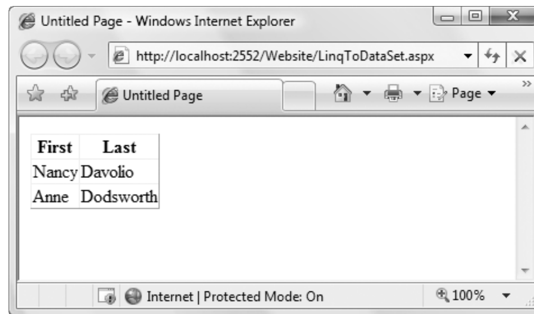


Рис. 13.6. Фильтрация DataSet с помощью LINQ

Оба подхода работают одинаково хорошо. Подход с DataView удобен в автономных "толстых" клиентах, потому что дает возможность манипулировать данными, не жертвуя отслеживанием изменений DataSet. (В приложении ASP.NET отличие обычно не важно.) Подход на основе проекций обеспечивает вам возможность сократить количество полей лишь теми, что вам необходимо видеть.

Конечно, нет необходимости использовать LINQ to DataSet для достижения результата, показанного на рис. 13.6. Вы можете получить то же самое, используя метод DataTable.Select() для фильтрации записей с правильными фамилиями и модифицируя схему GridView, чтобы она показывала только нужные два столбца. Однако LINQ to DataSet позволяет воспользоваться преимуществом операций, которые не имеют прямых эквивалентов в DataSet, вроде средства группировки, описанного выше.

Типизированные DataSet

Типизированные DataSet — другое решение для преодоления ограничений DataSet. Поскольку типизированные DataSet используют строго типизированные классы, вам не нужно полагаться на методы Field<T> и AsEnumerable(), что позволяет составлять намного более читабельные выражения.

Например, если вы используете строго типизированный DataSet для таблицы Employees, вы сможете переписать выражение из предыдущего примера, получив следующий более простой код:

```
var matches = from employee in ds.Employees
  where employee.LastName.StartsWith("D")
  select new { First = employee.FirstName, employee.LastName };
```

Этот код не только проще понимать, но он также выглядит намного более похожим на выражения, которые вы используете для опроса пользовательских классов в обычных коллекциях.

Однако в этой технике имеется одно существенное ограничение. Если вы создаете типизированный DataSet, используя более раннюю версию .NET, то ваши строго типизированные классы DataSet наследуются от DataTable. Если вы создадите DataSet в Visual Studio 2008, они наследуются от TypedTableBase<T> (который, в свою очередь, унаследован от DataTable). TypedTableBase<T> реализует IEnumerable<T>, а DataTable — нет. Поэтому показанное выше выражение LINQ работает, только если вы создадите типизированный DataSet в Visual Studio 2008. Если же у вас старый типизированный DataSet, то вам придется использовать методы AsEnumerable() и Field<T>, как вы делаете это, применяя LINQ с обычным DataSet.

Visual Studio не регенерирует автоматически код типизированного DataSet. Если вы откроете проект библиотеки классов из более ранней версии Visual Studio и сконфигурируете этот проект для .NET 3.5, то Visual Studio не изменить типизированных классов DataSet. Чтобы получить новую модель типизированных DataSet с TypedTableBase<T>, вам придется регенерировать ваш типизированный DataSet. Вы можете перестроить его с нуля, используя дизайнер типизированного DataSet, но более простой подход заключается во внесении минимального изменения на поверхности дизайна с последующей отменой его. Например, если вы модифицируете свойство поля в дизайнере типизированного DataSet, то Visual Studio перегенерирует код типизированного DataSet. К сожалению, в Visual Studio не предусмотрено более прямого подхода к перестройке типизированных DataSet.

Значения null

Метод Field<T> играет важную роль, предоставляя вам строго типизированный доступ к значениям вашего поля. Также он выполняет еще один полезный трюк: преобразует значения null (представленные DBNull.Value) в настоящие null-ссылки. (DataSet изначально не выполняет этого шага, потому что на момент его создания допускающие null типы еще не были частью каркаса.) В результате вы можете проверять null-ссылку вместо сравнения значений с DBNull.Value, что упрощает конечные выражения LINQ. Вот пример:

```
var matches = from product in ds.Tables["Products"].AsEnumerable()
 where product.Field<DateTime>("DiscontinuedDate") != null
 select product;
```

При использовании null-значений убедитесь, что вы не пытаетесь обратиться к члену значения, которое само может быть null. Например, если вы хотите получить продукты, производство которых прекратилось, в определенном диапазоне дат, то должны проверять null-значения перед выполнением сравнения дат, как показано ниже:

```
var matches = from product in ds.Tables["Products"].AsEnumerable()
  where product.Field<DateTime>("DiscontinuedDate") != null &&
        product.Field<DateTime>("DiscontinuedDate").Year > 2006
  select product;
```

Значения `null` не обрабатываются столь же гладко в типизированном `DataSet`. К сожалению, процедуры свойств, жестко привязанные к пользовательским классам `DataRow` в типизированном `DataSet`, генерируют исключения, когда встречают `null`-значения. Чтобы обойти это, вы должны использовать более изощренный синтаксис `Field<T>` при обращении к полю, которое может содержать `null`.

LINQ to SQL

Для многих ASP.NET-разработчиков LINQ to SQL — наиболее ценная часть LINQ. Она позволяет вам использовать обычные выражения LINQ, подобные тем, что вы уже видели, для запросов информации из баз данных SQL Server. Такое волшебство возможно благодаря скрытой трансляции выражений LINQ в запросы SQL. Запросы выполняются тогда, когда вам нужны данные — другими словами, когда вы начинаете выполнять перечисление результатов. И если это вас не очень впечатляет, добавим, что LINQ to SQL также включает механизм для применения обновлений. Он предусматривает отслеживание изменений всех извлеченных вами данных, а это значит, что вы можете модифицировать запрошенные вами объекты и затем фиксировать их в базе данных за один прием.

Совет. Вы можете думать о LINQ to SQL как о некоторой комбинации подхода на основе пользовательских классов со средствами отслеживания данных `DataSet`.

Наиболее очевидный недостаток LINQ to SQL состоит в том, что он ограничен механизмом базы данных SQL Server. Действительно, его стоило бы назвать LINQ to SQL Server. В будущем появятся и другие поставщики LINQ для других баз данных, и LINQ to SQL со временем может охватить их все в рамках одной модели. Но в ближайшее время LINQ to SQL полезен, только если вы решили ограничиться базой данных SQL Server.

LINQ to SQL — впечатляющая технология, но большинству разработчиков ASP.NET она сулит лишь небольшой выигрыш. Как и в случае `DataSet`, разработчики ASP.NET намного чаще станут использовать средства построения запросов LINQ to SQL, чем его средства пакетного обновления. Это связано с тем, что обновления в Web-приложении обычно выполняются поодиночке, а не пакетом. Также они имеют тенденцию происходить немедленно после обратной отправки страницы пользователем. И в этой точке у вас в руках есть оригинальные и новые (обновленные) значения, что облегчает применение простых команд ADO.NET для проведения изменений.

На заметку! В отличие от `DataSet`, LINQ to SQL обладает способностью, позволяющей вам пересоздать объект при последующем запросе и использовать его для применения изменений. (С `DataSet` вы вынуждены сначала выполнять запрос либо вручную отслеживать подробности изменений.) Однако в большинстве случаев все же проще выполнить прямое обновление с помощью объекта `SqlCommand`, чем заботиться о контексте данных LINQ to SQL. С другой стороны, средство пакетного обновления — оружие приложений “толстого” клиента, которому приходится кэшировать данные между обновлениями.

Короче говоря, LINQ to SQL не предоставляет никаких средств, которые невозможно было бы дублировать кодом ADO.NET или вашими собственными объектами, LINQ to Objects (для фильтрации в памяти) и DataSet (когда необходимо отслеживать изменения). Однако LINQ to SQL может облегчить вам жизнь в следующих аспектах.

- *Меньший объем кодирования.* Вам не нужно писать код ADO.NET для выполнения запросов к базе данных. Вы можете также использовать инструмент для генерации необходимых классов данных.

- *Гибкие средства запросов.* Вместо того чтобы бороться с SQL, вы можете использовать модель запросов LINQ. В конечном итоге вы сможете иметь дело с одной согласованной моделью (выражениями LINQ) для доступа ко многим различным типам данных — от баз данных до XML.
- *Отслеживание изменений и пакетные обновления.* Вы можете изменить многие детали данных, которые вы запрашиваете, и фиксировать пакетное обновление, без необходимости написания кода ADO.NET.

LINQ to Entities

LINQ to SQL — не последнее слово Microsoft в мире ORM (object-relational mapping — объектно-реляционное отображение). Фактически другое ориентированное на базы данных расширение LINQ, известное как *LINQ to Entities*, находится в процессе разработки. LINQ to Entities предоставляет средства, превосходящие LINQ to SQL, за счет повышенной сложности. Во-первых, LINQ to Entities использует модель поставщика ADO.NET, из чего следует поддержка любого механизма реляционной базы данных, имеющего соответствующую фабрику поставщиков. Во-вторых, LINQ to Entities поддерживает намного более сложное отображение между реляционными данными и классами. В то время как LINQ to SQL предполагает, что вы хотите работать с классами, основанными на таблицах вашей базы данных (и наследующими их структуру), LINQ to Entities позволяет вам заполнить пробел между реляционными данными и вашей концептуальной моделью данных. Другими словами, он позволяет получать информацию из вашей базы данных и помещать ее в более изощренные интеллектуальные бизнес-объекты.

Подробнее о LINQ to Entities и других разработках LINQ читайте на `http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx`.

Совет. Прежде чем вы получите возможность работать с LINQ to SQL, следует добавить ссылку на сборку `System.Data.Linq.dll`, где расположены все основные типы. По умолчанию новые Web-приложения не включают этой ссылки.

Классы сущностей данных

Когда вы извлекаете информацию из базы данных с LINQ to SQL, она преобразуется из последовательности записей таблицы в группу объектов в памяти. Этот шаг преобразования — сердце LINQ to SQL. Чтобы выполнить его, вы должны явно указать, куда извлеченные данные должны быть помещены. Это делается пометкой вашего класса данных атрибутами из пространства имен `System.Data.Linq`. Например, в главе 8 вы извлекали данные из базы и создавали объект `EmployeeDetails` для каждой записи. Чтобы выполнить то же самое автоматически с LINQ to SQL, нужно декорировать `EmployeeDetails` атрибутом, который отобразит его свойства на поля.

Два ключевых атрибута, которые вам понадобятся — это `Table` и `Column`. Вы применяете `Table` к объявлению класса, чтобы ассоциировать его с таблицей базы данных. Атрибут `Column` применяется к каждому свойству, которое нужно ассоциировать с соответствующим столбцом этой таблицы.

Атрибут `Table` имеет единственное свойство `Name` для указания имени таблицы. Атрибут `Column` принимает несколько свойств вдобавок к имени столбца (`Name`). Вы можете указать, соответствует ли свойство значению первичного ключа (`IsPrimaryKey`), является ли оно автоматически нумеруемым значением идентичности (`IsDbGenerated`) либо временной меткой или номером версии строки (`IsVersion`). В более совершенных сценариях вы можете использовать свойство `Storage` для указания имени лежащей в основе переменной, хранящей значение свойства, если LINQ to SQL должен устанавли-

вать эту переменную непосредственно, минуя общедоступную процедуру `set` свойства. Вы можете также использовать `UpdateCheck` для установки способа обработки параллельного доступа к этому значению при выполнении обновления, что будет продемонстрировано ниже в этой главе.

Ниже приведена отредактированная версия класса `EmployeeDetails` с добавленными атрибутами `Table` и `Column`. Для удобства класс `EmployeeDetails` использует автоматические свойства (это значит, что приватные поля-члены, стоящие за каждым свойством, генерируются автоматически, и потому не показаны).

```
using System.Data.Linq;
[Table(Name="Employees")]
public class EmployeeDetails
{
    [Column(IsPrimaryKey=true)]
    public int EmployeeID { get; set; }
    [Column]
    public string FirstName { get; set; }

    [Column]
    public string LastName { get; set; }

    [Column]
    public string TitleOfCourtesy { get; set; }
    public EmployeeDetails(int employeeID, string firstName, string lastName,
      string titleOfCourtesy)
    {
        EmployeeID = employeeID;
        FirstName = firstName;
        LastName = lastName;
        TitleOfCourtesy = titleOfCourtesy;
    }
    public EmployeeDetails(){}
}
```

В этом примере имена свойств соответствуют именам полей в точности, так что нет необходимости указывать имя поля в атрибуте `Column`.

DataContext

Все выражения LINQ должны действовать как некоторый объект. Поэтому для того, чтобы использовать выражения LINQ для извлечения информации из базы данных, вам понадобится объект, заполняющий пробел. Этот объект должен поддерживать `IEnumerable<T>`, но он не может быть обычным находящимся в памяти контейнером данных (вроде коллекции). Вместо этого он должен быть достаточно интеллектуальным, чтобы извлекать данные из базы, осуществляя запрос при начале перечисления ваших результатов и выполнении выражения LINQ.

Класс, играющий эту роль в LINQ to SQL, называется `DataContext` и находится он в пространстве имен `System.Data.Linq`. Этот класс инкапсулирует всю центральную функциональность, которая вам потребуется при работе с LINQ to SQL, включая способ доступа к таблицам через интерфейс `IEnumerable<T>` и механизм фиксации изменений. `DataContext` — стартовая точка программирования LINQ to SQL.

Прежде чем вы сможете использовать `DataContext`, следует создать его экземпляр. Делая это, вы можете передать объект `SqlConnection`, строку с именем базы данных (для SQL Server Express Edition), строку с именем сервера (для подключения к базе данных по умолчанию) либо полноценную строку соединения (для исключения любых возможностей путаницы).

Как только вы создадите DataContext, вы можете использовать метод GetTable<T> для получения доступа к таблице. Вашим аргументом типа будет класс данных, который вы используете для хранения данных для каждой извлеченной записи. Вот пример:

```
DataContext dataContext = new DataContext(connectionString);
Table<EmployeeDetails> table = dataContext.GetTable<EmployeeDetails>();
```

Обратите внимание, что этот код ничего не говорит явно о том, что он использует таблицу Employee. Эта деталь определена атрибутом Table, прикрепленным к классу EmployeeDetails.

Метод GetTable<T> возвращает объект Table<T>. Технически объект Table<T> представляет вашу таблицу, но не содержит никаких данных. Фактически вы можете выполнять итерацию по содержимому Table<T> как коллекции записей или же привязать его непосредственно к элементу управления данными вроде GridView:

```
gridEmployees.DataSource = table;
gridEmployees.DataBind();
```

Каждая запись в объекте Table<T> представлена экземпляром типа, специфицированного вами изначально, когда вы вызывали метод GetTable<>. В текущем примере это означает, что каждая запись представлена объектом EmployeeDetails, и вы можете осуществлять итерацию по таблице, используя код вроде следующего:

```
foreach (EmployeeDetails employee in table)
{ ... }
```

Когда выполняется итерация по объекту Table<T> (или когда он привязывается к элементу управления, что инициирует то же действие), Table<T> извлекает данные, которые ему нужны. Объект Table<T> создает и открывает соединение с базой данных, выполняет запрос, который получает все записи таблицы, а затем закрывает соединение (по завершении вашего блока foreach). Другими словами, класс Table<T> выполняет ту же самую отложенную загрузку, что и выражение запроса LINQ, т.е. извлекает данные, когда вы выполняете итерацию по нему.

Как вы уже видели, вам не обязательно использовать выражение LINQ для получения полного, не отфильтрованного результата из базы данных. Вместо этого вы просто применяете DataContext.GetTable<T>, чтобы получить соответствующий класс Table<T>. Однако если вы хотите выполнить запрос, использующий фильтрацию, сортировку, проекцию, группировку или любые другие операции LINQ, то сделать это легко. Вы просто применяете объект Table<T> в качестве источника данных для выражения LINQ. Например, вот как вы можете получить сотрудников с фамилиями, начинающимися с определенной буквы:

```
IEnumerable<EmployeeDetails> matches = from employee in table
  where employee.LastName.StartsWith("D")
  select employee;
```

Напомним, что выражение не будет вычисляться до тех пор, пока вы не начнете итерацию по результатам (коллекции matches объектов EmployeeDetail). В этой точке выражение LINQ инициирует объект Table<T> для действительного извлечения нужных вам данных. Однако объект Table<T> достаточно интеллектуален, чтобы принять во внимание выражение LINQ при извлечении данных. Вместо выполнения запроса, извлекающего все записи таблицы, и обработки их на клиенте, он изменяет команду SQL таким образом, чтобы она соответствовала запросу LINQ. В данном примере это означает, что к выражению SQL добавляется конструкция WHERE. Эта конструкция WHERE находит записи о сотрудниках, у которых фамилия начинается с буквы “D”, игнорируя все остальные.