Краткий справочник по LINQ для C#

Скачай LINQPad

LINQ

Карманный справочник

bhv®

O'REILLY®

Джозеф Албахари,
Бен Албахари

Джозеф Албахари
Бен Албахари

LINQ

Карманный справочник

Санкт-Петербург
«БХВ-Петербург»
2009

УДК 681.3.068
ББК 32.973.26-018.1
А45

Албахари, Дж.

А45 LINQ. Карманный справочник: Пер. с англ. / Дж. Албахари, Б. Албахари. — СПб.: БХВ-Петербург, 2009. — 240 с.: ил.

ISBN 978-5-9775-0317-4

Справочник посвящен технологии LINQ (Language Integrated Query) — новой функциональной возможности языка C# 3.0 и платформы Framework, которая позволяет писать безопасные структурированные запросы к локальным коллекциям объектов и удаленным источникам данных. Рассмотрены базовые понятия LINQ, такие как отложенное выполнение, цепочки итераторов и распознавание типов в лямбда-выражениях, различие между локальными и интерпретируемыми запросами, синтаксис запросов C# 3.0, сравнение синтаксиса запросов с лямбда-синтаксисом, а также запросы со смешанным синтаксисом, составление сложных запросов, написание эффективных запросов LINQ для SQL, построение деревьев выражений, запросы LINQ для XML.

Для программистов

УДК 681.3.068
ББК 32.973.26-018.1

ISBN 978-0-596-51924-7 (англ.)
ISBN 978-5-9775-0317-4 (рус.)

Оглавление

Карманный справочник

Технология LINQ (Language Integrated Query, запрос, интегрированный в язык) позволяет вам писать безопасные в смысле типизации структурированные запросы к локальным коллекциям объектов и удаленным источникам данных. Это новая функциональная возможность языка C# 3.0 и платформы Framework.

LINQ позволяет вам строить запросы к любой коллекции, реализующей интерфейс `IEnumerable<>`, будь то массив, список, коллекция XML DOM или удаленный источник данных, такой как таблицы на SQL-сервере. Технология LINQ предлагает сочетание достоинств проверки типов на этапе компиляции и динамического составления запросов.

Системные типы, поддерживающие LINQ, определены в пространствах имен `System.Linq` и `System.Linq.Expressions` в сборке System.Core.

> ***Примечание***
>
> Примеры в этой книге соответствуют примерам из *гл. 8—10* книги "C# 3.0 in a Nutshell", выпущенной издательством O'Reilly, и встроены в интерактивное инструментальное приложение составления запросов LINQPad. Вы можете загрузить его с вебсайта **http://www.linqpad.net/**.

Основы

Базовыми единицами данных в LINQ являются *последовательности* и *элементы*. Последовательность — это любой объект, который реализует обобщенный интерфейс `IEnumerable`, а элемент — это просто элемент последовательности. В следующем примере массив строк `names` — это последовательность, а `Tom`, `Dick` и `Harry` — элементы:

```
string[] names = { "Tom","Dick","Harry" };
```

Такая последовательность называется *локальной*, потому что представляет локальную коллекцию объектов в памяти.

Оператор запроса — это метод, преобразующий последовательность. В типичном случае оператор запроса принимает *входную последовательность* и возвращает результат преобразования — *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операторов запроса, и все они реализованы как статические методы расширения. Они называются *стандартными операторами запроса*.

> ***Примечание***
>
> LINQ также поддерживает последовательности, которые могут быть динамически получены от удаленного источника данных, такого как SQL Server. Эти последовательности, кроме прочего, реализуют интерфейс `IQueryable<>` и поддерживаются соответствующим набором стандартных операторов запроса в классе `Queryable`. Более подробную информацию вы найдете в *разд. "Интерпретируемые запросы"* далее.

Запрос — это выражение, которое преобразует последовательности с помощью операторов запроса. Простейший запрос состоит из одной входной последовательности и одного оператора. Например, мы можем применить оператор `Where` к строковому массиву и извлечь те его элементы, длина которых не меньше четырех символов:

```
string[] names = { "Tom","Dick","Harry" };
IEnumerable<string> filteredNames =
  System.Linq.Enumerable.Where
    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.Write (n + "|");            //
Dick|Harry|
```

Поскольку стандартные операторы запроса реализованы в виде методов расширения, мы можем вызвать `Where` непосредственно для массива `names` так, словно это метод экземпляра:

```
IEnumerable<string> filteredNames =
  names.Where (n => n.Length >= 4);
```

Чтобы можно было откомпилировать эту строчку, вы должны импортировать пространство имен `System.Linq`. Вот полный код примера:

```
using System;
using System.Linq;
class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom","Dick","Harry" };
    IEnumerable<string> filteredNames =
      names.Where (n => n.Length >= 4);
```

```
    foreach (string name in filteredNames)
Console.Write (name + "|");
  }
}
// Результат: Dick|Harry|
```

> ***Примечание***
>
> Если вы не знакомы с такими понятиями языка C#, как лямбда-выражения, методы расширения и неявное приведение типов, посетите сайт **www.albahari.com/cs3primer**.

Мы могли бы еще больше сократить запрос с помощью неявного приведения типа переменной `filteredNames`:

```
var filteredNames = names.Where (n => n.Length
>= 4);
```

Большинство операторов запроса принимает лямбда-выражение в качестве аргумента. Лямбда-выражение помогает направить и сформировать запрос. В нашем примере лямбда-выражение выглядит так:

```
n => n.Length >= 4
```

Входной аргумент соответствует входному элементу. В этом случае входной аргумент `n` представляет имя в массиве и имеет тип `string`. Оператор `Where` требует, чтобы лямбда-выражение возвращало значение типа `bool`. Когда оно истинно, элемент должен быть включен в выходную последовательность.

В этой книге мы будем называть такие запросы *лямбда-запросами*. В языке C# имеется специальный синтаксис для написания запросов, и он назы-

вается *синтаксисом, облегчающим восприятие запроса*. Перепишем предыдущий пример в соответствии с этим синтаксисом:

```
IEnumerable<string> filteredNames =
  from n in names
  where n.Contains ("a")
  select n;
```

Лямбда-синтаксис и синтаксис, облегчающий восприятие, дополняют друг друга. В следующих разделах мы обсудим их более подробно.

Лямбда-запросы

Лямбда-запросы являются самым гибким, фундаментальным видом запросов. В этом разделе мы покажем, как составлять цепочки операторов для формирования сложных запросов и представим вам несколько новых операторов.

Цепочки операторов запросов

Чтобы строить более сложные запросы, вы добавляете новые операторы, образуя цепочку. Например, в следующем запросе из массива извлекаются все строки с буквой "a", после чего они сортируются по длине и переводятся в верхний регистр:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
```

```
  .Select  (n => n.ToUpper());
foreach (string name in query)
  Console.Write (name + "|");
// Результат: JAY|MARY|HARRY|
```

`Where`, `OrderBy` и `Select` — стандартные операторы запроса, которые транслируются в методы расширения из класса `Enumerable`.

Мы уже обсуждали оператор `Where`, возвращающий отфильтрованную версию входной последовательности. Оператор `OrderBy` возвращает отсортированную версию последовательности, поданной на его вход, а результатом оператора `Select` является последовательность, у которой каждый входной элемент подвергся преобразованию, или *проекции*, со стороны лямбда-выражения (в нашем случае `n.ToUpper()`). В цепочке операторов данные проходят слева направо, то есть, вначале они фильтруются, затем сортируются и после этого проецируются.

> ***Примечание***
>
> Оператор запроса никогда не изменяет входную последовательность. Вместо нее он возвращает новую. Это соответствует парадигме *функционального программирования*, из которой исходит LINQ.

Приведем сигнатуры этих методов расширения (незначительно упростив сигнатуру оператора `OrderBy`):

```
static IEnumerable<TSource> Where<TSource> (
  this IEnumerable<TSource> source,
  Func<TSource,bool> predicate)
```

```
static IEnumerable<TSource>
OrderBy<TSource,TKey> (
  this IEnumerable<TSource> source,
  Func<TSource,TKey> keySelector)
static IEnumerable<TResult>
Select<TSource,TResult> (
  this IEnumerable<TSource> source,
  Func<TSource,TResult> selector)
```

Когда операторы запроса выстраиваются в цепочку, как в нашем примере, выходная последовательность одного оператора становится входной для следующего. Получается что-то вроде конвейерной линии, изображенной на рис. 1.

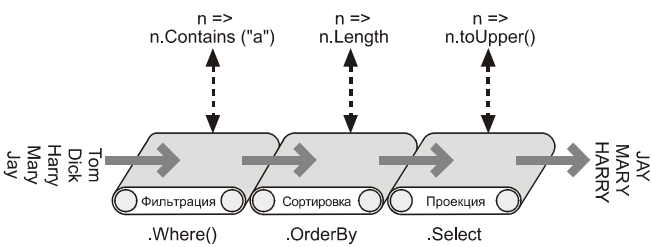

Рис. 1. Цепочка операторов запроса

Индентичный запрос может быть сформирован *последовательно*, как в следующем коде:

```
var filtered = names.Where   (n => n.Contains
("a"));
var sorted = filtered.OrderBy (n => n.Length);
var finalQuery = sorted.Select  (n =>
n.ToUpper());
```

Последовательность `finalQuery` композиционно идентична последовательности `query`, сконструи-

рованной ранее. Здесь каждый промежуточный шаг содержит допустимый запрос, который может быть выполнен:

```
foreach (string name in filtered)
  Console.Write (name + "|");        //
Harry|Mary|Jay|
Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|");        //
Jay|Mary|Harry|
Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");        //
JAY|MARY|HARRY|
```

Составление лямбда-выражений

В предыдущих примерах мы передавали оператору `Where` такое лямбда-выражение:

```
n => n.Contains ("a")      // На входе строка,
                           // на выходе булево
                              выражение
```

> ***Примечание***
>
> Выражение, возвращающее значение типа `bool`, называется *предикатом*.

Предназначение лямбда-выражения зависит от конкретного оператора запроса. С оператором `Where` оно показывает, должен ли элемент попадать в выходную последовательность. У оператора `OrderBy` лямбда-выражение отображает каждый элемент входной последовательности на ключ сортировки, а у оператора `Select` оно определяет, как

должен быть преобразован элемент из входной последовательности перед подачей его в выходную.

> ***Примечание***
>
> В операторе запроса лямбда-выражение всегда относится к отдельным элементам входной последовательности, а не к последовательности в целом.

Вы можете относиться к лямбда-выражению как к *обратному вызову*. Оператор запроса вычисляет значение лямбда-выражения "по требованию", — как правило, один раз для каждого элемента входной последовательности. Лямбда-выражения позволяют вам внести свою логику в операторы запроса. Это делает операторы запроса гибкими и в то же время простыми по внутреннему устройству. Приведем полную реализацию метода `Enumerable.Where`, опустив то, что касается обработки исключений:

```
public static IEnumerable<TSource>
Where<TSource> (
  this IEnumerable<TSource> source,
  Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Лямбда-выражения и сигнатуры делегатов *Func*

Стандартные операторы запроса используют обобщенные делегаты `Func`. Это семейство неспе-

циализированных делегатов в пространстве имен `System.Linq`, отличительная особенность которых формулируется следующим образом:

Аргументы, указывающие тип, в делегате `Func` стоят точно в таком же порядке, что и в лямбда-выражении.

Иными словами, `Func<TSource,bool>` соответствует лямбда-выражению `TSource=>bool`: принимает аргумент типа `TSource` и возвращает значение типа `bool`.

Аналогичным образом `Func<TSource,TResult>` соответствует лямбда-выражению `TSource=>TResult`.

Приведем определения всех делегатов `Func` (обратите внимание, что тип возвращаемого значения всегда указывается в качестве последнего обобщенного аргумента).

```
delegate TResult Func <T> ();
delegate TResult Func <T, TResult>
                 (T arg1);
delegate TResult Func <T1, T2, TResult>
                 (T1 arg1, T2 arg2);
delegate TResult Func <T1, T2, T3, TResult>
                 (T1 arg1, T2 arg2, T3 arg3);
delegate TResult Func <T1, T2, T3, T4, TResult>
                (T1 arg1, T2 arg2, T3 arg3, T4
arg4);
```

Лямбда-выражения и типы элементов

В стандартных операторах запроса используются следующие имена обобщенных типов.

Обозначение обобщенного типа	Смысл
`TSource`	Тип элемента входной последовательности
`TResult`	Тип элемента выходной последовательности — отличается от `TSource`
`TKey`	Тип *ключа*, используемого при сортировке, группировании или объединении

Тип `TSource` определяется входной последовательностью. Типы `TResult` и `TKey` *выводятся из вашего лямбда-выражения*.

Рассмотрим, например, сигнатуру оператора запроса `Select`:

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

Делегат `Func<TSource,TResult>` соответствует лямбда-выражению `TSource=>TResult`, которое отображает входной элемент на выходной элемент. Типы `TSource` и `TResult` не совпадают, и лямбда-выражение может изменить тип любого элемента. Более того, лямбда-выражение *определяет тип выходной последовательности*. В следующем запросе оператор `Select` преобразует строковые элементы в целочисленные:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
```

```
IEnumerable<int> query = names.Select (n =>
n.Length);
foreach (int length in query)
  Console.Write (length);        // 34543
```

Компилятор *автоматически распознает* тип `TResult` по типу значения, возвращаемого лямбда-выражением. В данном случае делается вывод, что `TResult` — это `int`.

Естественный порядок элементов

В технологии LINQ важную роль играет оригинальный порядок элементов во входной последовательности, который определяет работу некоторых операторов запросов, таких как `Take`, `Skip` и `Reverse`.

Оператор `Take` выводит первые `x` элементов, отбрасывая остальные; оператор `Skip` игнорирует первые `x` элементов и выводит остальные; оператор `Reverse` меняет порядок следования элементов на обратный.

Операторы `Where` и `Select` сохраняют первоначальный порядок элементов входной последовательности. Вообще, технология LINQ старается не изменять порядок элементов входной последовательности, когда это возможно.

Прочие операторы

Не все операторы запросов возвращают последовательность. *Поэлементные* операторы извлекают из

входной последовательности только один элемент. К этой группе относятся операторы `First`, `Last`, `Single` и `ElementAt`:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();       // 10
int lastNumber   = numbers.Last();        // 6
int secondNumber = numbers.ElementAt(1);  // 9
```

Операторы *агрегирования* возвращают скалярное значение, как правило, числового типа:

```
int count = numbers.Count();          // 5;
int min = numbers.Min();              // 6;
```

Квантификаторы возвращают булево значение:

```
bool hasTheNumberNine = numbers.Contains (9);
// true
bool hasMoreThanZeroElements = numbers.Any( );
// true
bool hasAnOddElement = numbers.Any
                       (n => n % 2 == 1);
// true
```

Поскольку эти операторы не возвращают коллекцию, вы не можете передавать их результаты другим операторам. Иными словами, они должны стоять на последнем месте в запросе (или подзапросе).

Некоторые операторы запроса принимают две последовательности. Например, оператор `Concat` присоединяет одну последовательность к другой, а `Union` делает то же самое, но удаляет повторяющиеся элементы. Операторы объединения тоже попадают в эту категорию.

Синтаксис, облегчающий восприятие запроса

Язык C# предоставляет синтаксическое сокращение для LINQ-запросов, называемое синтаксисом, облегчающим восприятие, или просто синтаксисом запроса.

В предыдущем разделе мы написали запрос для извлечения строк, содержащих букву "a", сортировки их по длине и перевода их в верхний регистр. Вот как выглядит тот же запрос в соответствии с синтаксисом, облегчающим его восприятие:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a") // Фильтровать элементы
  orderby n.Length         // Сортировать элементы
  select  n.ToUpper();     // Проецировать каждый элемент
  foreach (string name in query)
    Console.Write (name + "/");
// Результат: JAY/MARY/HARRY/
```

Запрос с синтаксисом, облегчающим восприятие, всегда начинается с конструкции `from` и заканчивается конструкцией либо `select`, либо `group`. Конструкция `from` объявляет *переменную итерации* (в данном случае `n`). Вы можете считать, что она используется для перебора элементов входной

коллекции, аналогично оператору `foreach`. Полное описание этого синтаксиса приведено на рис. 2.

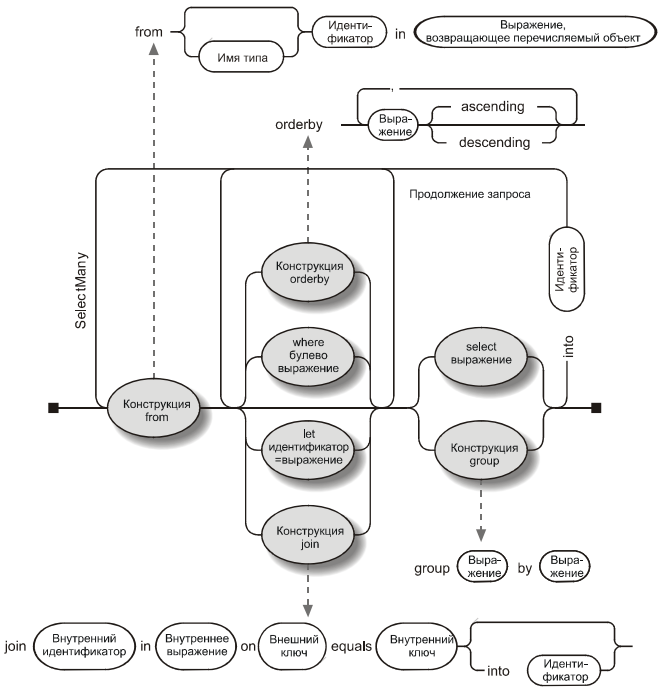

Рис. 2. Синтаксис, облегчающий восприятие запроса

Компилятор обрабатывает запросы с синтаксисом, облегчающим восприятие, переводя их в лямбда-синтаксис. Это делается механически, подобно тому как оператор `foreach` транслируется в вызовы `GetEnumerator` и `MoveNext`. Фактически это означает, что все, написанное вами в соответствии с синтакси-

сом, облегчающим восприятие, могло быть написано и с соблюдением лямбда-синтаксиса. Запрос из нашего примера транслируется в следующий код:

```
IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select  (n => n.ToUpper());
```

Затем операторы `Where`, `OrderBy` и `Select` будут откомпилированы по тем же правилам, что и запросы, изначально написанные с соблюдением лямбда-синтаксиса. В этом случае они связываются с методами расширения из класса `Enumerable`, потому что пространство имен `System.Linq` импортировано, а коллекция `names` реализует интерфейс `IEnumerable<string>`. Впрочем, компилятор не оказывает какого-то особого предпочтения классу при трансляции запросов, написанных в синтаксисе, облегчающим восприятие. Вы можете считать, что он просто механически подставляет слова "Where," "OrderBy" и "Select" в оператор, а затем компилирует так, словно вы сами написали эти имена методов. Такой подход обеспечивает гибкость компиляции. Например, операторы в запросах LINQ к SQL, которые мы напишем в следующих разделах, будут связываться с методами расширения из класса `Queryable`.

> ***Примечание***
>
> Если мы уберем из программы директиву `using System.Linq`, запрос с синтаксисом, облегчающим восприятие, не будет откомпилирован, потому что методы `Where`, `OrderBy` и `Select` будет не к чему привязывать. Запросы с синтаксисом, облегчающим восприятие, не *компи-*

лируются, если вы не импортировали пространство имен (или не написали метод экземпляра для каждого оператора запроса!)

Переменные итерации

Идентификатор, следующий непосредственно за ключевым словом `from`, называется *переменной итерации*. В наших примерах переменная итерации `n` появляется в каждом предложении запроса. И, тем не менее, всякий раз она перебирает элементы *другой* последовательности:

```
from    n in names          // n — это переменная
итерации
where   n.Contains ("a")    // n непосредственно
из массива
orderby n.Length            // n после фильтрации
select  n.ToUpper()         // n после сортировки
```

Это становится ясно после изучения результата механической трансляции в лямбда-синтаксис, проведенной компилятором:

```
names.Where   (n => n.Contains ("a"))
     .OrderBy (n => n.Length)
     .Select  (n => n.ToUpper( ))
```

Каждый экземпляр `n` имеет область видимости, ограниченную соответствующим лямбда-выражением.

Синтаксис, облегчающий восприятие, и SQL-синтаксис

Синтаксис, облегчающий восприятие LINQ-запросов, внешне напоминает синтаксис SQL-запросов,

но они сильно отличаются друг от друга. Запрос LINQ приводится к выражению на языке C# и поэтому подчиняется стандартным правилам языка. Например, в LINQ-запросе вы не можете использовать переменную до ее объявления. В SQL-запросе вы ссылаетесь на псевдоним таблицы в конструкции `SELECT` до ее определения в конструкции `FROM`.

Подзапрос в LINQ является всего лишь еще одним выражением C# и поэтому не требует специального синтаксиса. Подзапросы SQL пишутся в соответствии со специальными правилами.

В LINQ-запросах данные логически переходят от одного оператора к другому слева направо. В языке SQL порядок операторов не такой жесткий.

Запрос LINQ состоит из *конвейера операторов*, принимающих и возвращающих упорядоченные последовательности. Запрос SQL является *сетью* предложений, работающих, как правило, с *неупорядоченными наборами данных*.

Синтаксис, облегчающий восприятие, и лямбда-синтаксис

Синтаксис, облегчающий восприятие, и лямбда-синтаксис имеют свои достоинства.

Первый отличается простотой на запросах, включающих в себя:

- конструкцию `let`, объявляющую новую переменную наряду с переменной итерации;

- оператор `SelectMany`, `Join` или `GroupJoin`, после которого внешняя переменная-ссылка итерации.

(Конструкцию `let` мы опишем позже, в *разд. "Стратегии построения сложных запросов"*, а операторы `SelectMany`, `Join` и `GroupJoin` — в *разд. "Проецирование"* и *"Объединение"*.)

Промежуточную позицию занимают запросы, содержащие простые формы операторов `Where`, `OrderBy` и `Select`. Для них подходит любой синтаксис, и выбор, в основном, определяется вашим вкусом.

Для запросов, состоящих из одного оператора, лучше пользоваться лямбда-синтаксисом. Он лаконичнее, и запросы получаются короче.

Наконец, существует много операторов, для которых нет ключевого слова в синтаксисе, облегчающем восприятие. В этом случае вы должны использовать лямбда-синтаксис, хотя бы частично. Это относится ко всем операторам, не попавшим в следующий список:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending,
ThenByDescending
Group, Join, GroupJoin
```

Запросы со смешанным синтаксисом

Если оператор запроса не поддерживается синтаксисом, облегчающим восприятие, вы можете ком-

бинировать этот синтаксис с лямбда-синтаксисом. Единственное требование, которое при этом выдвигается, — каждая составляющая "понятного" синтаксиса должна быть полной (то есть начинаться с конструкции `from` и заканчиваться конструкцией `select` или `group`).

Например:

```
int count = (from name in names
             where n.Contains ("a")
             select name
            ).Count();
```

Бывают ситуации, в которых запросы со смешанным синтаксисом оказываются самыми эффективными в терминах функциональности и простоты. Избегайте оказывать предпочтение какому-то одному из двух вариантов синтаксиса. В противном случае вы не сможете уверенно и безошибочно писать запросы в смешанном синтаксисе!

Отложенное выполнение

Важной особенностью большинства операторов запроса является тот факт, что они выполняются не тогда, когда сконструированы, а при *переборе элементов* (то есть когда для соответствующего перечислителя вызывается метод `MoveNext`):

```
var numbers = new List<int>();
numbers.Add (1);
// Построить запрос
IEnumerable<int> query = numbers.Select (n => n
* 10);
```

```
numbers.Add (2);    // "Незаметно" добавить еще
один элемент
foreach (int n in query)
  Console.Write (n + "|");    // 10|20|
```

Дополнительный элемент, который мы "тайком" добавили *после* конструирования запроса, попадает в результат, потому что ни фильтрация, ни сортировка не происходят, пока не начнет выполняться оператор `foreach`. Это называется *отложенным* или *"ленивым"* выполнением. Отложенное выполнение характерно для всех стандартных операторов запроса, кроме

- операторов, возвращающих один элемент или скалярное значение, таких как `First` или `Count`;
- операторов преобразования типа:

  `ToArray, ToList, ToDictionary, ToLookup`

Эти операторы приводят к немедленному выполнению запроса, потому что у возвращаемых ими результатов нет механизма, который обеспечивал бы отложенное выполнение. Например, метод `Count` возвращает целое число, которое затем никак не "перебирается". Следующий запрос выполняется немедленно:

```
int matches = numbers.Where (n => n <
2).Count();    // 1
```

Отложенное выполнение играет важную роль, потому что оно отделяет конструирование запроса от его выполнения. Это позволяет вам конструировать запрос за несколько шагов, а также делать LINQ-запросы к SQL.

Примечание

Подзапросы дают вам еще один уровень косвенности. Отложенному выполнению подлежит все содержимое подзапроса, включая методы агрегирования и преобразования типов (см. *разд. "Подзапросы"*).

Повторное выполнение

Отложенное выполнение имеет одно важное последствие: запрос с отложенным выполнением выполняется повторно, если происходит повторный перебор элементов:

```
var numbers = new List<int>() { 1, 2 };
IEnumerable<int> query = numbers.Select (n => n
* 10);
foreach (int n in query)
  Console.Write (n + "|");   // 10|20|
numbers.Clear();
foreach (int n in query)
  Console.Write (n + "|");   // <ничего>
```

Существует пара ситуаций, в которых повторное выполнение нежелательно:

- иногда вы хотите "заморозить" или кэшировать результаты, полученные в определенный момент;
- некоторые запросы требуют интенсивных вычислений (или обращаются к удаленной базе данных), и вы не хотите повторять их без необходимости.

Аннулировать повторное выполнение можно, вызвав оператор преобразования типа, например, `ToArray` или `ToList`. Оператор `ToArray` копирует результат запроса в массив, а оператор `ToList` копирует результат в обобщенный список `List<>`:

```
var numbers = new List<int>() { 1, 2 };
List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList(); // Выполняется немедленно; результат — в списке List<int>
numbers.Clear( );
Console.WriteLine (timesTen.Count);      // По-прежнему 2
```

Внешние переменные

Если лямбда-выражения в вашем запросе ссылаются на локальные переменные, то эти переменные захватываются и подчиняются семантике *внешних переменных*. Другими словами, важно значение переменной в момент выполнения запроса, а не в момент ее захвата:

```
int[] numbers = { 1, 2 };
int factor = 10; // далее мы захватим эту переменную:
var query = numbers.Select (n => n * factor);
factor = 20; // изменить значение захваченной переменной
foreach (int n in query)
  Console.Write (n + "|");   // 20|40|
```

Вы можете попасть в эту ловушку, когда строите запрос внутри цикла `foreach`. Например, в следующем коде необходимо воспользоваться вре-

менной переменной, чтобы успешно удалить все гласные из строки:

```
IEnumerable<char> query = "Not what you might
expect";
foreach (char vowel in "aeiou")
{
  char temp = vowel;
  query = query.Where (c => c != temp);
}
```

Без этой вспомогательной переменной запрос будет учитывать лишь последнее значение переменной `vowel` (т. е. "u") для каждого следующего фильтра, так что будут удалены только буквы "u".

Механика отложенного выполнения

Операторы запроса обеспечивают отложенное выполнение тем, что возвращают *декорирующие* последовательности.

В отличие от традиционного класса коллекций, такого как массив или связный список, последовательность-декоратор не имеет собственной структуры для хранения элементов. На самом деле она является оболочкой для другой последовательности, которую вы предоставляете на этапе выполнения, и с этой последовательностью декоратор поддерживает постоянную связь. Как только вы запросите данные у декоратора, он запросит данные у входной последовательности, для которой является оболочкой.

Примечание

Преобразование, выполняемое оператором запроса, как раз и вызывает "декорирование". Если выходная последовательность не является результатом преобразования, она будет заместителем, но не декоратором.

Вызов оператора `Where` всего лишь конструирует последовательность-декоратор, исходя из ссылки на входную последовательность, лямбда-выражения и других аргументов. Элементы входной последовательности перебираются только в том случае, когда выполняется перебор элементов декоратора.

На рис. 3 показано, как составлен следующий запрос:

```
IEnumerable<int> lessThanTen =
  new int[] {5, 12, 3}.Where (n => n < 10);
```

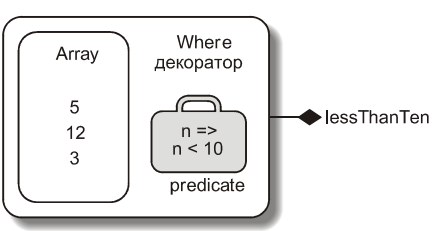

Рис. 3. Последовательность-декоратор

Когда вы перебираете элементы последовательности `lessThanTen`, вы фактически обращаетесь к массиву через декоратор `Where`.

Хотим вас обрадовать. Если вы когда-нибудь решите написать собственный оператор запроса, то вам будет просто реализовать последовательность-декоратор с помощью итератора C#. Вот, например, как можно написать свой метод `Select`:

```
static IEnumerable<TResult>
Select<TSource,TResult> (
  this IEnumerable<TSource> source,
  Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Этот метод представляет собой итератор благодаря присутствию оператора `yield return`. Функционально он является сокращением для следующего кода:

```
static IEnumerable<TResult>
Select<TSource,TResult> (
  this IEnumerable<TSource> source,
  Func<TSource,TResult> selector)
{
  return new SeLectSequence (source, selector);
}
```

Здесь *SeLectSequence* — это класс (написанный компилятором), перечислитель которого инкапсулирует необходимую логику в методе-итераторе.

Таким образом, когда вы вызываете оператор, например, `Select` или `Where`, вы всего лишь создаете экземпляр перечисляемого класса, который декорирует входную последовательность.

Цепочки декораторов

Выстраивание операторов запроса в цепочку приводит к наслоению декораторов. Рассмотрим следующий запрос:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }
  .Where   (n => n < 10)
  .OrderBy (n => n)
  .Select  (n => n * 10);
```

Каждый оператор запроса создает экземпляр нового декоратора, являющийся оболочкой для предыдущей последовательности (и получается что-то вроде матрешки). Объектная модель этого запроса изображена на рис. 4. Обратите внимание, что она полностью сконструирована до выполнения перебора элементов.

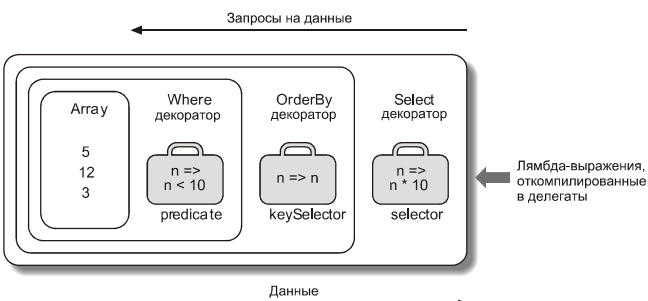

Рис. 4. Вложенные последовательности-декораторы

Когда вы перебираете элементы последовательности `query`, вы отправляете запрос оригинальному

массиву, преобразованному многослойной конструкцией (или цепочкой) из декораторов.

> ***Примечание***
>
> Добавление оператора `ToList` в конец этого запроса заставила бы все предшествующие операторы выполниться немедленно, а вся объектная модель "сложилась" бы в обычный список.

Декоратор оператора `Select` ссылается на декоратор оператора `OrderBy`, который ссылается на декоратор оператора `Where`, который ссылается на массив. Отложенное выполнение обеспечивается тем, что вы строите идентичную объектную модель, если составляете запрос последовательно:

```
IEnumerable<int>
  source    = new int[] { 5, 12, 3 },
  filtered  = source   .Where   (n => n < 10),
  sorted    = filtered .OrderBy (n => n),
  query     = sorted   .Select  (n => n * 10);
```

Как выполняются запросы

Перебор элементов в предыдущем запросе имеет такой результат:

```
foreach (int n in query)
  Console.WriteLine (n); // 30/50
```

"За кулисами" оператор `foreach` вызывает метод `GetEnumerator` декоратора оператора `Select` (последнего, или самого внешнего оператора), который и запускает всю процедуру. Результатом будет

цепочка перечислителей, структурно отражающая цепочку декораторов. На рис. 5 показано, как выполняется запрос по мере перебора элементов последовательностей.

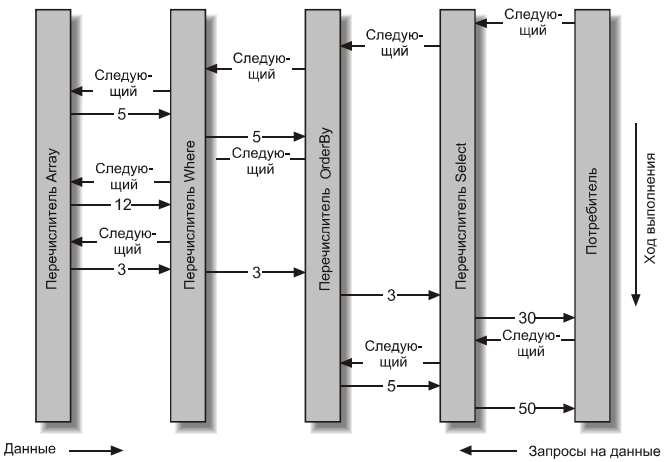

Рис. 5. Выполнение локального запроса

Вспомним, что мы сравнивали запрос с конвейерной линией. Развивая эту аналогию, можно сказать, что LINQ-запрос — это "ленивый" конвейер, который включается только по требованию. Конструирование запроса приводит к созданию конвейерной линии. Все необходимые детали находятся на своих местах, но конвейерные ленты и "лямбда-рабочие" неподвижны. Когда потребитель запрашивает деталь (начинает перебирать элементы в запросе), включается самый правый конвейер. Это приводит к поочередному включению остальных

конвейерных лент, по мере возникновения необходимости в очередном элементе последовательности. Технология LINQ придерживается модели "вытягивания", где инициатором является потребитель, а не "проталкивание", где инициатор — поставщик. Как мы убедимся впоследствии, это очень важно для обеспечения масштабируемости LINQ-запросов к базам данных.

Подзапросы

Подзапрос — это запрос, содержащийся в лямбда-выражении другого запроса. В следующем примере подзапрос используется для упорядочивания фамилий музыкантов по алфавиту:

```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick
Wright" };
IEnumerable<string> query =
  musos.OrderBy (m => m.Split().Last());
```

Метод `m.Split` преобразует каждую строку в коллекцию слов, для которой мы затем вызываем оператор запроса `Last`. Здесь `Last` является подзапросом, а последовательнось `query` ссылается на *внешний запрос*.

Подзапросы разрешены, потому что вы можете поставить любое допустимое выражение языка C# в правую часть лямбда-выражения. Подзапрос — это всего лишь еще одно выражение C#. Отсюда следует, что правила для подзапросов естественно вы-

текают из правил для лямбда-выражений (и из общих правил для операторов запросов).

Область видимости подзапроса ограничена содержащим его выражением, причем он может ссылаться на внешний аргумент лямбда-выражения (или переменную итерации в синтаксисе, облегчающем восприятие).

`Last` — очень простой подзапрос. В следующем запросе из массива извлекаются все строки, длина которых равна длине кратчайшей строки:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 =>
n2.Length)
                                .Select  (n2 =>
n2.Length).First()
  );
// Результат: Tom, Jay
```

А вот этот же запрос в синтаксисе, облегчающем восприятие:

```
IEnumerable<string> comprehension =
  from   n in names
  where  n.Length ==
    (from n2 in names
     orderby n2.Length
     select n2.Length).First()
  select n;
```

Поскольку внешняя переменная итерации (`n`) видна подзапросу, мы не можем использовать ее в качестве итерационной переменной подзапроса.

Подзапрос выполняется, когда вычисляется значение содержащего его лямбда-выражения. То есть он выполняется по требованию, на усмотрение внешнего запроса. Можно сказать, что выполнение происходит "снаружи вовнутрь". Локальные запросы придерживаются этой модели буквально, а интерпретируемые (например, запросы LINQ к SQL) — *концептуально*.

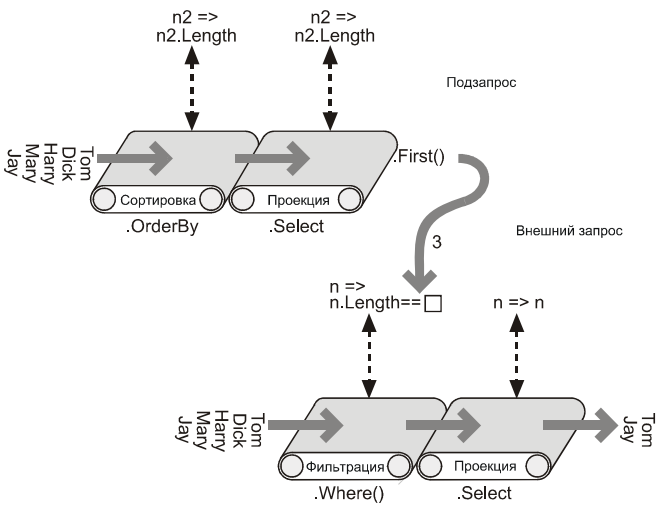

Рис. 6. Композиция подзапроса

Подзапрос выполняется, когда это необходимо для внешнего запроса. В нашем примере подзапрос (верхний конвейер на рис. 6) выполняется один раз для каждой итерации внешнего цикла.

Мы можем сформулировать предыдущий подзапрос лаконичнее:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length ==
         names.OrderBy (n2 =>
n2.Length).First().Length
  select n;
```

А функция агрегирования `Min` позволяет еще больше упростить запрос:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.Min (n2 => n2.Length)
  select n;
```

В *разд. "Интерпретируемые запросы"* мы покажем, как можно опрашивать удаленные источники данных, такие как SQL-таблицы. В нашем примере создается идеальный LINQ-запрос к SQL: он будет обработан как единое целое, а данные будут следовать на сервер и обратно только один раз. Однако такой запрос неэффективен в случае локальной коллекции, потому что его подзапрос заново выполняется на каждом шаге внешнего цикла. Этого можно избежать, выполнив подзапрос отдельно (чтобы он перестал быть подзапросом):

```
int shortest = names.Min (n => n.Length);
IEnumerable<string> query = from   n in names
                            where  n.Length ==
shortest
                            select n;
```

Примечание

Составление подзапросов в таком стиле рекомендуется практически во всех случаях, когда опрашиваются локальные коллекции. Исключением является *коррелированный* подзапрос, то есть подзапрос, ссылающийся на внешнюю переменную итерации. Мы обсудим коррелированные подзапросы в *разд. "Проецирование"*.

Подзапросы и отложенное выполнение

Даже если в подзапрос входит оператор, возвращающий один элемент, или оператор агрегирования (например, `First` или `Count`), это не приводит к немедленному выполнению *внешнего* запроса; принцип отложенного выполнения внешнего запроса остается в силе. Дело в том, что подзапросы вызываются *косвенно*, то есть через делегат в случае локального запроса или через дерево выражений в случае интерпретируемого запроса.

Интересная ситуация возникает, когда вы включаете подзапрос в выражение в операторе `Select`. Если это локальный запрос, вы фактически *проецируете последовательность запросов*, каждый из которых подлежит отложенному выполнению. Эффект, как правило, прозрачен, а производительность при этом возрастает.

Стратегии построения сложных запросов

В этом разделе мы опишем три стратегии построения сложных запросов:

- последовательное конструирование запроса;
- применение ключевого слова `into`;
- создание оболочек для запросов.

Все эти стратегии направлены на создание *цепочек* запросов и их результаты на этапе выполнения идентичны.

Последовательное построение запросов

В начале книги мы демонстрировали последовательное построение лямбда-запроса:

```
var filtered = names.Where   (n => n.Contains
("a"));
var sorted = filtered.OrderBy (n => n);
var query = sorted.Select  (n => n.ToUpper());
```

Поскольку каждый из участвующих здесь операторов возвращает последовательность-декоратор, результирующий запрос представляет собой многослойную цепочку декораторов, которую вы получили бы и из запроса, содержащего единственное выражение. Однако у последовательного построения запросов есть пара потенциальных достоинств:

- запросы легче писать;

- вы можете добавлять операторы запроса *с учетом некоторых условий*.

Последовательное построение часто оказывается уместно при использовании синтаксиса, облегчающего восприятие. Предположим, например, что нам нужно воспользоваться классом `Regex`, чтобы удалить все гласные из списка имен, а затем представить в алфавитном порядке те имена, длина которых по-прежнему превышает два символа. В лямбда-синтаксисе мы можем написать соответствующий запрос в виде одного выражения, выполняя проецирования до фильтрации:

```
IEnumerable<string> query = names
  .Select  (n => Regex.Replace (n, "[aeiou]",
"",))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
// Результат: { "Dck", "Hrry", "Mry" }
```

Прямой перевод этого кода в синтаксис, облегчающий восприятие, проблематичен, потому что предложения в этом синтаксисе должны идти в порядке `where`-`orderby`-`select`; в противном случае компилятор их не воспримет. А если мы реорганизуем запрос, поставив проецирование на последнее место, результат будет другим:

```
IEnumerable<string> query =
  from    n in names
  where   n.Length > 2
  orderby n
  select  Regex.Replace (n, "[aeiou]", "");
// Результат: { "Dck", "Hrry", "Jy", "Mry", "Tm"
}
```

К счастью, существует несколько способов прийти к первоначальному результату, используя синтаксис, облегчающий восприятие. Первый заключается в последовательном построении запроса:

```
IEnumerable<string> query =
  from   n in names
  select Regex.Replace (n, "[aeiou]", "");
query = from n in query
  where n.Length > 2
  orderby n
  select n;
// Результат: { "Dck", "Hrry", "Mry" }
```

Ключевое слово *into*

Ключевое слово `into` позволяет вам "продолжить" запрос после проецирования и является синтаксическим инструментом для последовательного построения запроса. Пользуясь ключевым словом `into`, мы можем переписать предыдущий пример так:

```
IEnumerable<string> query =
  from   n in names
  select Regex.Replace (n, "[aeiou]", "")
  into noVowel
  where noVowel.Length > 2
  orderby noVowel
  select noVowel;
```

Примечание

В синтаксисе, облегчающем восприятие, ключевое слово `into` интерпретируется двумя очень

разными способами, в зависимости от контекста. Значение, которое мы сейчас обсуждаем, — это индикация *продолжения запроса* (второе значение — индикация оператора `GroupJoin`).

Единственное место, где вы можете поставить `into`, это после конструкции `select` или `group`. Ключевое слово `into`, так сказать, заново запускает запрос, и вы можете дальше писать конструкции `where`, `orderby` и `select`.

> ***Примечание***
>
> Хотя с точки зрения синтаксиса, облегчающего восприятие, удобно считать, что `into` заново запускает запрос, при трансляции в окончательную лямбда-форму будет получен *один* запрос. То есть, ключевое слово `into` не дает никакого выигрыша в производительности. Впрочем, потерь вы тоже не несете!

Эквивалентом для `into` в лямбда-синтаксисе служит длинная цепочка операторов.

Правила определения областей видимости

Все переменные запроса становятся недоступны после ключевого слова `into`. Следующий код не пройдет компиляцию:

```
var query =
  from n1 in names
  select n1.ToUpper()
  into n2
```

```
  where n1.Contains ("x")  // Ошибка: n1 вне об-
ласти видимости
  select n2;
```

Чтобы понять, почему это так, рассмотрим, как этот код транслируется в лямбда-синтаксис:

```
var query = names
  .Select (n1 => n1.ToUpper())
  .Where  (n2 => n1.Contains ("x"));
```

Оригинальное имя (`n1`) утрачено к тому времени, когда начинает работать фильтр `Where`. Входная последовательность для оператора `Where` содержит только имена в верхнем регистре, и фильтрация на основе `n1` невозможна.

Создание оболочек для запросов

Запрос, построенный последовательно, может быть сформулирован в виде единственного оператора, если вокруг одного запроса создать оболочку из другого. В общем виде это выглядит следующим образом. Запрос

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

переписывается так:

```
var finalQuery = from ... in (tempQueryExpr)
```

Создание оболочки семантически идентично последовательному построению запросов или использованию ключевого слова `into` (без промежуточной переменной). Во всех случаях окончательным результатом является линейная цепочка

операторов запроса. Рассмотрим, например, такой запрос:

```
IEnumerable<string> query =
  from   n in names
  select Regex.Replace (n, "[aeiou]", "");
query = from n in query
        where n.Length > 2
        orderby n
        select n;
```

Перепишем его с использованием оболочек:

```
IEnumerable<string> query =
  from n1 in
  (
    from   n2 in names
    select Regex.Replace (n2, "[aeiou]", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

При преобразовании в лямбда-синтаксис мы получим ту же линейную цепочку операторов, что и в предыдущих примерах:

```
IEnumerable<string> query = names
  .Select  (n => Regex.Replace (n,"[aeiou]",
"",))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

(Компилятор не генерирует заключительный оператор `.Select (n => n)` в силу его избыточности.)

Запросы в оболочках могут сбить с толку того, кто изучает код, потому что внешне напоминают подзапросы, обсуждавшиеся ранее. И у тех, и у других существуют внешний и внутренний запрос. Однако

после преобразования в лямбда-синтаксис вы можете убедиться, что создание оболочек является всего лишь стратегией для построения цепочки операторов. Конечный результат не имеет ничего общего с подзапросом, у которого внутренний запрос находится в лямбда-выражении другого запроса.

Возвращаясь к аналогии с конвейерной линией, заметим, что запрос внутри оболочки соответствует предыдущей ленте конвейера. В отличие от него подзапрос находится где-то над конвейером и включается по сигналу от "лямбда-механизма", связанного с конвейером (см. рис. 6).

Стратегии проецирования

Инициализаторы объектов

До сих пор наши конструкции `select` проецировали элементы скалярных типов. Пользуясь инициализаторами объектов языка C#, вы можете проецировать результаты в более сложные типы. Предположим, например, что в первом шаге запроса мы хотим удалить гласные из имен в списке, но при этом стремимся сохранить и первоначальные версии для последующих запросов. Можно написать следующий вспомогательный класс:

```
class TempProjectionItem
{
  public string Original;     // Первоначальное
имя
```

```
  public string Vowelless;   // Имя без гласных
}
```

а затем проецировать результаты с помощью инициализаторов:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original = n, Vowelless = Regex.Replace (n,
"[aeiou]", "")
  };
```

Получится коллекция типа `IEnumerable<TempProjectionItem>`, которую мы можем передать следующему запросу:

```
IEnumerable<string> query = from   item in temp
                            where
item.Vowelless.Length > 2
                            select
item.Original;
// Результат: Dick, Harry, Mary
```

Анонимные типы

Анонимные типы позволяют вам структурировать промежуточные результаты, не прибегая к помощи специального класса. Мы можем убрать класс `TempProjectionItem` из предыдущего примера и воспользоваться анонимным типом:

```
var intermediate = from n in names
  select new
  {
```

```
    Original = n,
    Vowelless = Regex.Replace (n,"[aeiou]", "")
  };
IEnumerable<string> query =
  from   item in intermediate
  where  item.Vowelless.Length > 2
  select item.Original;
```

Результат получится точно такой же, но мы избавили себя от необходимости писать еще один класс. Эту работу за нас сделает компилятор, который напишет временный класс с полями, отражающими структуру нашей проекции. Впрочем, отсюда вытекает, что запрос `intermediate` будет иметь следующий тип:

```
IEnumerable
<имя_случайно_выбранное_компилятором>
```

Единственный способ объявить переменную такого типа состоит в использовании ключевого слова `var`. В этой ситуации слово `var` — нечто большее, чем просто средство для поддержания аккуратности кода. Оно необходимо.

Весь запрос можно переписать лаконичнее, если применить ключевое слово `into`:

```
var query = from n in names
  select new
  {
     Original = n,
     Vowelless = Regex.Replace (n,"[aeiou]", "")
  }
  into temp
  where temp.Vowelless.Length > 2
  select temp.Original;
```

Синтаксис, облегчающий восприятие, предоставляет сокращение для написания подобных запросов: ключевое слово `let`.

Ключевое слово *let*

Ключевое слово `let` вводит новую переменную вместе с переменной итерации.

С его помощью мы можем написать запрос, извлекающий из входной последовательности строки, длина которых без гласных превышает два символа:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query =
  from n in names
  let vowelless = Regex.Replace (n,"[aeiou]",
"")
  where vowelless.Length > 2
  orderby vowelless
  select n;  // Благодаря конструкции let переменная n
             // находится в области видимости
```

Компилятор обрабатывает конструкцию `let`, проецируя запрос во временный анонимный тип, который содержит переменную итерации и новую переменную выражения. Иными словами, компилятор транслирует этот запрос в код предыдущего примера.

Конструкция `let` решает две задачи:

- она проецирует новые элементы вместе с уже существующими;

- она позволяет многократно использовать выражение в запросе, не переписывая его.

Подход с применением конструкции `let` особенно уместен в нашем примере, потому что он позволяет конструкции `select` проецировать либо оригинальное имя (`n`), либо его версию без гласных (`v`).

Вы можете использовать любое количество предложений `let`, до или после предложения `where` (см. рис. 2). Конструкция `let` может ссылаться на переменные, представленные в предыдущих конструкциях `let` (с учетом границ области видимости, установленных конструкцией `into`). Она транспарентно *проецирует* все имеющиеся переменные.

Нет необходимости в том, чтобы выражение `let` возвращало значение скалярного типа; иногда бывает полезно возвращать, например, подпоследовательность.

Интерпретируемые запросы

Технология LINQ предоставляет вам две параллельно существующих архитектуры: *локальные* запросы для локальных коллекций объектов и *интерпретируемые* запросы к удаленным источникам данных. До сих пор мы рассматривали только архитектуру локальных запросов, работающих с коллекциями, реализующими интерфейс `IEnumerable<>`. Локальные запросы транслируются в операторы класса `Enumerable`, которые, в свою очередь, транслируются в цепочки последовательностей-декораторов. Де-

легаты, которых они принимают (выраженные в синтаксисе, облегчающем восприятие, в лямбда-синтаксисе или в соответствии с традиционным синтаксисом делегатов) являются полностью локальными для IL-кода (кода, написанного на промежуточном языке), как и любой другой метод C#.

Интерпретируемые запросы являются *описательными*. Они работают с последовательностями, реализующими интерфейс `IQueryable<>`, и транслируются в операторы класса `Queryable`, который порождает *деревья выражений*, интерпретируемые на этапе выполнения.

> ***Примечание***
>
> Операторы запроса в классе `Enumerable` на самом деле могут работать с последовательностями, реализующими интерфейс `IQueryable<>`. Трудность в том, что результирующие запросы всегда выполняются локально на клиенте, и поэтому в классе `Queryable` содержится второй набор операторов запроса.

На платформе .NET Framework существуют две реализации интерфейса `IQueryable`:

- запросы LINQ к SQL;
- запросы LINQ к сущностям.

Кроме того, метод расширения `AsQueryable` генерирует оболочку `IQueryable` вокруг обычной перечисляемой коллекции. Мы опишем метод `AsQueryable` в *разд. "Построение выражений для запросов"*.

В этом разделе мы воспользуемся запросами LINQ к SQL для иллюстрации архитектуры интерпретируемых запросов.

> ***Примечание***
>
> Интерфейс `IQueryable<>` является расширением интерфейса `IEnumerable<>` и содержит дополнительные методы для конструирования деревьев выражений. В большинстве случаев вы можете не задумываться о тонкостях работы этих методов, потому что Framework вызывает их косвенно. В *разд. "Построение выражений для запросов"* интерфейс `IQueryable<>` рассмотрен более подробно.

Предположим, что мы создаем на SQL-сервере простую таблицу с информацией о клиентах и заносим в нее несколько имен с помощью SQL-скрипта:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

Когда таблица готова, мы можем написать интерпретируемый LINQ-запрос на языке C#, чтобы узнать, у каких клиентов имена содержат букву "a":

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

```
using System.Data.Linq.Mapping;
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string
Name;
}
class Test
{
  static void Main( )
  {
    var dataContext = new DataContext ("cx
string... ");
    Table<Customer> customers =
      dataContext.GetTable <Customer>( );
    IQueryable<string> query = from c in
customers
      where c.Name.Contains ("a")
      orderby c.Name.Length
      select  c.Name.ToUpper();
    foreach (string name in query)
      Console.WriteLine (name);
  }
}
```

В реализации "LINQ к SQL" этот запрос транслируется в следующий запрос на языке SQL:

```
SELECT UPPER([t0].[Name]) AS [value]
FROM [Customer] AS [t0]
WHERE [t0].[Name] LIKE '%a%'
ORDER BY LEN([t0].[Name])
```

который возвращает такой результат:

```
JAY
MARY
HARRY
```

Как работают интерпретируемые запросы

Рассмотрим, как обрабатывается запрос, приведенный выше.

Во-первых, компилятор преобразует запрос из синтаксиса, облегчающего восприятие, в лямбда-синтаксис. Это происходит точно так, как и в случае с локальными запросами:

```
IQueryable<string> query = customers
  .Where   (n => n.Name.Contains ("a"))
  .OrderBy (n => n.Name.Length)
  .Select  (n => n.Name.ToUpper());
```

Затем компилятор транслирует методы оператора запроса. Здесь локальные и интерпретируемые запросы обрабатываются по-разному: интерпретируемые транслируются в операторы запроса класса `Queryable`, а не класса `Enumerable`.

Чтобы понять, почему это так, мы должны обратить внимание на переменную `customers`, источник данных, на котором строится весь запрос. Переменная имеет тип `Table<>`, реализующий интерфейс `IQueryable<>` (подтип интерфейса `IEnumerable<>`). Это означает, что у компилятора есть выбор, как транслировать оператор `Where`: он может либо вызвать метод расширения из класса `Enumerable`, либо следующий метод расширения из класса `Queryable`:

```
public static IQueryable<TSource> Where<TSource>
(
  this IQueryable<TSource> source,
  Expression <Func<TSource,bool>> predicate)
```

Компилятор выбирает `Queryable.Where`, потому что его сигнатура *более специфична*.

Обратите внимание, что метод `Queryable.Where` принимает предикат в оболочке типа `Expression<TDelegate>`. Для компилятора это означает, что необходимо оттранслировать лямбда-выражение (выражение `n=>n.Name.Contains("a")`) в *дерево выражений*, а не в откомпилированный делегат. Дерево выражений — это объектная модель, в основе которой лежат типы из пространства имен `System.Linq.Expressions`. Оно может быть проанализировано на этапе выполнения, чтобы запрос LINQ к SQL мог впоследствии оттранслировать это дерево в SQL-оператор.

Поскольку метод `Queryable.Where` сам возвращает последовательность, реализующую `IQueryable<>`, тот же процесс происходит и с операторами `OrderBy` и `Select`. Конечный результат изображен на рис. 7. В сером прямоугольнике находится дерево выражений, описывающее весь запрос, которое можно обойти на этапе выполнения.

Выполнение

Интерпретируемые запросы, как и локальные, подчиняются модели поведения, включающей в себя отложенное выполнение. Отсюда следует, что SQL-оператор не генерируется, пока вы не начнете перебирать элементы в запросе. Более того, повторный перебор элементов в том же запросе приведет к повторному обращению с запросом к базе данных.

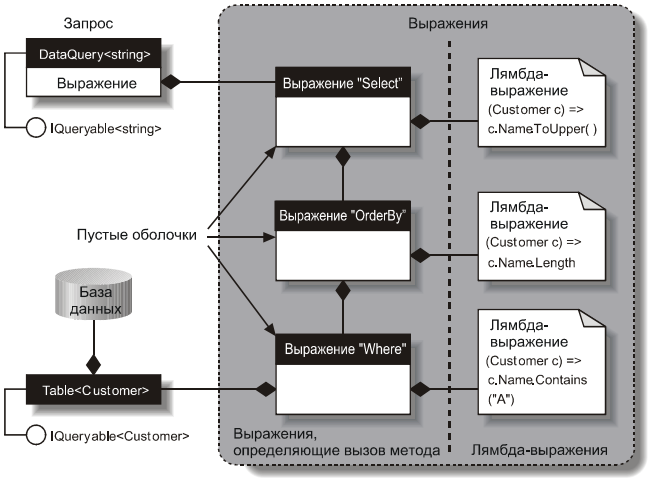

Рис. 7. Композиция интерпретируемого запроса

Выполнение интерпретируемых запросов отличается от выполнения локальных. Когда вы перебираете элементы последовательности в интерпретируемом запросе, самая внешняя последовательность запускает программу, которая обходит все дерево выражений, обрабатывая его как единое целое. В нашем примере запрос LINQ к SQL транслирует дерево выражений в SQL-оператор, который затем выполняется и возвращает результат в виде последовательности.

Примечание

Для успешной работы запросу LINQ к SQL необходима некоторая информация о схеме базы данных. Атрибуты `Table` и `Column`, которыми мы снабдили класс `Customer`, служат как раз для этой цели. В *разд. "Запросы LINQ к SQL"* мы подробно опишем эти атрибуты.

Ранее мы говорили, что LINQ-запрос напоминает конвейерную линию на производстве. Когда вы включаете конвейер `IQueryable`, это не приводит к включению всей линии, как было в случае с локальными запросами. Запускается только лента `IQueryable`, имеющая специальный перечислитель, который вызывает менеджера по производству. Менеджер осматривает всю производственную линию, которая содержит не откомпилированный код, а "заготовки" (выражения для вызова метода) с инструкциями, "наклеенными" на их переднюю часть (с деревьями лямбда-выражений). Менеджер обходит все выражения и, в данном случае, переписывает их на отдельный листок бумаги (SQL-оператор). Затем инструкции на листке выполняются, а результаты передаются потребителю. Включается только одна конвейерная лента; остальная часть производственной линии представляет собой сеть из пустых оболочек, существующих исключительно для описания того, что нужно сделать.

Такая реализация имеет определенные практические последствия. Например, когда вы делаете локальный запрос, вы можете написать собственные методы запроса (это очень легко, если пользоваться итераторами), дополнив ими стандартный набор методов. В случае удаленных запросов это затруднительно и, к тому же, нежелательно. Если вы напишете свой метод расширения `MyWhere`, принимающий последовательность `IQueryable<>`, это будет выглядеть так, словно вы положили собственную заготовку на производственный конвейер. Менеджер просто не будет знать, как обращаться с

вашей заготовкой. Даже если вы вмешиваетесь в производственный процесс на этом этапе, ваше решение будет жестко ориентировано на конкретного поставщика данных, например, на технологию "LINQ к SQL", и не сможет работать с другими реализациями интерфейса `IQueryable`. Одним из достоинств стандартного набора методов в классе `Queryable` является то, что они определяют *стандартный словарь* для выдачи запроса к *любой* удаленной коллекции. Как только вы попытаетесь расширить этот словарь, ваш код перестанет быть универсальным.

Другим последствием этого подхода является потенциальная неспособность провайдера `IQueryable` справиться с некоторыми запросами, даже если они включают в себя только стандартные методы. Например, технология LINQ к SQL ограничена возможностями сервера базы данных, и некоторые LINQ-запросы не транслируются в SQL. Если вы знаете язык SQL, интуиция подскажет вам, какие это запросы. Впрочем, иногда вам придется поэкспериментировать, чтобы посмотреть, вызовет ли конкретный запрос ошибку на этапе выполнения. В некоторых случаях вы, возможно, удивитесь, что запрос работает! Шансы на успех возрастают, если пользоваться последней версией Microsoft SQL Server.

Оператор *AsEnumerable*

Оператор `Enumerable.AsEnumerable` является простейшим из всех операторов запросов.

Вот его прямое определение:

```
public static IEnumerable<TSource>
AsEnumerable<TSource>
               (this IEnumerable<TSource> source)
{
    return source;
}
```

Предназначение этого метода в том, чтобы приводить последовательность `IQueryable<T>` к типу `IEnumerable<T>`, вынуждая последующие операторы запроса привязываться к операторам `Enumerable`, а не `Queryable`. В результате оставшаяся часть запроса будет выполнена локально.

Для иллюстрации предположим, что у нас на SQL-сервере имеется таблица `MedicalArticles` ("Статьи по медицине"), и мы хотим воспользоваться запросом LINQ к SQL, чтобы получить все статьи о гриппе ("influenza "), аннотация к которым не превышает 100 слов. Для последнего предиката нам потребует регулярное выражение:

```
Regex wordCounter = new Regex (@"\b(\w|[-
'])+\b");


var query = dataContext.MedicalArticles
  .Where (article => article.Topic ==
"influenza" &&
    wordCounter.Matches (article.Abstract).Count
< 100);
```

Проблема в том, что наш сервер SQL Server не поддерживает регулярные выражения, поэтому запрос LINQ к SQL возбуждает исключение, смысл которого в невозможности оттранслировать запрос в SQL-оператор. Мы можем решить проблему, разбив за-

прос на два шага: вначале получим все статьи по гриппу с помощью запроса LINQ к SQL, а затем *на локальном уровне* отфильтруем статьи, оставив лишь те, у которых аннотация меньше 100 слов:

```
Regex wordCounter = new Regex (@"\b(\w|[-
'])+\b");


IEnumerable<MedicalArticle> sqlQuery =
  dataContext.MedicalArticles
  .Where (article => article.Topic ==
"influenza");


IEnumerable<MedicalArticle> localQuery =
sqlQuery
  .Where (article =>
    wordCounter.Matches (article.Abstract).Count
< 100);
```

Поскольку последовательность `sqlQuery` имеет тип `IEnumerable<MedicalArticle>`, второй запрос привязывается к локальным операторам запроса, заставляя эту часть фильтрации работать на стороне клиента.

С помощью метода `AsEnumerable` мы можем сделать все то же самое в одном запросе:

```
Regex wordCounter = new Regex (@"\b(\w|[-
'])+\b");


var query = dataContext.MedicalArticles
  .Where (article => article.Topic ==
"influenza")
  .AsEnumerable()
  .Where (article =>
    wordCounter.Matches (article.Abstract).Count
< 100);
```

Альтернативой методу `AsEnumerable` является метод `ToArray` или `ToList`. Преимущество метода `AsEnumerable` заключается в том, что он не форсирует немедленное выполнение запроса. К тому же он не создает дополнительную структуру для хранения данных.

> ***Примечание***
>
> Перенося обработку запроса с сервера базы данных на клиент, мы можем понизить производительность, особенно если будем запрашивать много строк. Более эффективным (зато и более сложным) подходом к нашей задаче будет использование интеграции SQL и CLR и создание функции для базы данных, которая реализовала бы регулярное выражение.

Запросы LINQ к SQL

Во всей книге мы пользуемся запросами LINQ к SQL для иллюстрации интерпретируемых запросов. В этом разделе обсуждаются важнейшие функциональные особенности этой технологии.

Классы сущностей в технологии LINQ к SQL

Технология LINQ к SQL позволяет вам использовать любой класс для представления данных, если вы сумеете декорировать его соответствующими атрибутами.

Приведем простой пример:

```
[Table]
public class Customer
{
  [Column(IsPrimaryKey=true)]
  public int ID;
  [Column]
  public string Name;
}
```

Атрибут `[Table]` из пространства имен `System.Data.Linq.Mapping` сообщает запросу LINQ к SQL, что объект этого типа представляет строку в таблице базы данных. По умолчанию предполагается, что имя таблицы совпадает с именем класса. Если это не так, вы можете указать ее имя следующим образом:

```
[Table (Name="Customers")]
```

Класс, декорированный атрибутом `[Table]`, называется *сущностью* в технологии "LINQ к SQL". Чтобы им можно было пользоваться, его структура должна близко (или абсолютно) соответствовать структуре таблицы базы данных, что превращает его в низкоуровневую конструкцию.

Атрибутом `[Column]` помечается поле или свойство, которое отображает столбец таблицы. Если имя столбца отличается от имени поля или свойства, вы можете указать имя столбца следующим образом:

```
[Column (Name="FullName")]
public string Name;
```

Свойство `IsPrimaryKey` внутри атрибута `[Column]` указывает, что столбец участвует в формировании

первичного ключа таблицы и необходим для поддержания идентичности объекта, а также для внесения обновлений в базу данных.

Вместо объявления открытых полей вы можете объявить открытые свойства в комбинации с закрытыми полями. Это позволит вам реализовать логику проверки допустимости значений в методах, обращающихся к свойствам. Если вы пойдете этим путем, вы сможете проинструктировать запрос LINQ к SQL так, чтобы он обходил ваши методы обращения к свойствам и записывал значение прямо в поле после получения его из базы данных:

```
string _name;

[Column (Storage="_name")]
public string Name
{ get { return _name; } set { _name = value; } }
```

Конструкция `Column(Storage="_name")` предписывает запросу LINQ к SQL заносить значение в поле `_name` (а не в свойство `Name`) при заполнении сущности значениями. Использование рефлексии позволяет полю быть закрытым, что мы и наблюдаем в этом примере.

Объект *DataContext*

После того как вы определили классы сущностей, вы можете приступить к выдаче запросов, создав объект `DataContext` и вызвав для него метод `GetTable`.

В следующем примере используется класс `Customer`, определенный ранее:

```
var dataContext = new DataContext ("cx
string...");
Table<Customer> customers =
  dataContext.GetTable <Customer>();

// вывести количество строк в таблице
Console.WriteLine (customers.Count());

// получить информацию о клиенте (Customer),
у которого ID равен 2
Customer cust = customers.Single (c =>c.ID == 2);
```

> ***Примечание***
>
> Оператор `Single` идеален для извлечения строки по первичному ключу. В отличие от оператора `First`, он возбуждает исключение, если возвращено более одного элемента.

Объект `DataContext` выполняет два действия. Во-первых, он выступает в качестве генератора таблиц, которые вы можете опросить. Во-вторых, он отслеживает все изменения, которые вы вносите в сущности, так что вы можете потом сохранить эти изменения в таблице:

```
var dataContext = new DataContext
("cx string...");
Table<Customer> customers =
  dataContext.GetTable <Customer>();
Customer cust = customers.OrderBy
(c =>c.Name).First();
cust.Name = "UpdatedName";
dataContext.SubmitChanges();
```

Объект `DataContext` следит за всеми сущностями, экземпляры которых он создает, так что он возвратит вам ту же сущность, если вы запросите те же строки в таблице. Иными словами, объект `DataContext` на протяжении своей жизни не выдаст две разных сущности, ссылающиеся на одну строку таблицы (где строка идентифицируется первичным ключом).

> ***Примечание***
>
> Вы можете отключить такую линию поведения, если сбросите в значение `false` свойство `ObjectTrackingEnabled` объекта `DataContext`. (Отключение отслеживания объектов лишит вас возможности вносить изменения в данные.)

Чтобы проиллюстрировать отслеживание объектов, предположим, что клиент, имя которого идет первым по алфавиту, имеет также и наименьшее значение ID. В следующем примере переменные `a` и `b` будут ссылаться на один и тот же объект:

```
var dataContext = new DataContext ("cx
string...");
Table<Customer> customers =
  dataContext.GetTable <Customer>();


Customer a = customers.OrderBy (c =>
c.Name).First();
Customer b =customers.OrderBy (c =>
c.ID).First();
```

Отсюда вытекают два интересных следствия. Во-первых, рассмотрим, что происходит, когда выдается второй запрос LINQ к SQL. Отправляется запрос к базе данных, и она возвращает одну строку.

Затем LINQ изучает первичный ключ этой строки и выполняет просмотр кэша сущности объекта `DataContext`. Найдя совпадение, LINQ возвращает уже существующий объект, *не обновляя никакие значения*. То есть, если другой пользователь только что обновил поле `Name` этого клиента в базе данных, новое значение игнорируется. Это очень важно для исключения непредвиденных побочных эффектов (в данный момент времени с объектом `Customer` может работать другой пользователь) и для поддержания параллельной обработки. Если вы изменили свойства объекта `Customer`, но еще не вызвали метод `SubmitChanges`, вы, конечно, не захотите, чтобы эти значения были автоматически переписаны другими.

> ***Примечание***
>
> Чтобы получить свежую информацию из базы данных, вы должны либо создать новый экземпляр `DataContext`, либо вызвать метод `Refresh` существующего объекта `DataContext`, передав ему одну или несколько сущностей, которые вы хотите обновить.

Вторым следствием отслеживания объектов является невозможность явно выполнять проецирование в тип сущности, чтобы выделить подмножество полей строки. Обязательно возникнут проблемы. Например, если вы хотите извлечь только имя клиента, допустимым является любой из следующих подходов:

```
customers.Select (c => c.Name);
customers.Select (c => new { Name = c.Name } );
```

```
customers.Select (c => new
                    MyCustomType { Name = c.Name }
);
```

А вот такой подход недопустим:

```
customers.Select (c => new Customer { Name =
c.Name } );
```

Дело в том, что сущности `Customer`, в конечном счете, будут заполнены лишь частично. Поэтому, когда вы в следующий раз запросите *все* столбцы записи о клиенте, вы получите те же объекты `Customer`, у которых заполнено только свойство `Name`.

> ***Примечание***
>
> В многоуровневых приложениях вы не можете использовать единственный статический экземпляр объекта `DataContext` в среднем слое для обработки всех запросов, потому что объект `DataContext` не обеспечивает безопасное выполнение потоков. Решение состоит в том, чтобы методы среднего уровня создавали новый объект `DataContext` на каждый клиентский запрос. На самом деле, это даже выгодно, потому что ответственность за обработку одновременных попыток обновления перекладывается на сервер, который лучше подготовлен для этой работы. Например, сервер базы данных обеспечивает изоляцию транзакций.

Автоматическое генерирование сущностей

Поскольку классы сущностей в технологии выдачи запросов LINQ к SQL должны иметь ту же струк-

туру, что и отображаемые ими таблицы, возникает желание генерировать их автоматически на основании имеющейся схемы базы данных. Вы можете делать это либо с помощью инструментального средства SqlMetal, вызываемого из командной строки, либо с помощью генератора запросов LINQ к SQL в Visual Studio. Обе утилиты генерируют сущности в виде частичных классов, чтобы вы могли реализовать дополнительную логику в отдельных файлах.

В качестве бонуса вы получаете сильно типизированный класс `DataContext`. Он является всего лишь подклассом `DataContext` со свойствами, возвращающими таблицы того же типа, что и каждая из сущностей. Это избавляет вас от необходимости вызывать метод `GetTable`:

```
var dataContext = new MyTypedDataContext
("...");


Table<Customer> customers =
dataContext.Customers;
Console.WriteLine (customers.Count());
```

или еще проще:

```
Console.WriteLine
(dataContext.Customers.Count());
```

Генератор запросов LINQ к SQL автоматически создает идентификатор во множественном числе (добавляет в конец букву "s" в соответствии в грамматикой английского языка — прим. перев.) там, где это необходимо. В нашем примере генерируется `dataContext.Customers`, а не `dataContext.Customer`, даже если SQL-таблица и класс сущности называются `Customer`.

Ассоциирование

Инструменты генерирования сущностей выполняют еще одну полезную работу. Для каждого отношения, определенного в базе данных, на каждой стороне отношения автоматически генерируются соответствующие свойства. Предположим, например, что мы определяем клиента и таблицу покупок с отношением "один ко многим":

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)

create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Если мы воспользуемся автоматически сгенерированными классами сущностей, мы сможем написать следующие запросы:

```
var dataContext = new MyTypedDataContext
("...");

// Получить информацию обо всех покупках,
// сделанных первым клиентом (в алфавитном
порядке):
```

```
Customer cust1 = dataContext.Customers
  .OrderBy (c => c.Name).First( );
foreach (Purchase p in cust1.Purchases)
  Console.WriteLine (p.Price);

// Найти клиента, сделавшего самое дешевое при-
обретение:

Purchase cheapest = dataContext.Purchases
  .OrderBy (p => p.Price).First( );

Customer cust2 = cheapest.Customer;
```

Если теперь окажется, что `cust1` и `cust2` — одно и то же лицо, то `c1` и `c2` будут *ссылаться на один и тот же объект*, то есть выражение `cust1==cust2` возвратит `true`.

Рассмотрим сигнатуру автоматически сгенерированного свойства `Purchases` сущности `Customer`:

```
[Association (Storage="_Purchases",
              OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases
{ get {...} set {...}}
```

Класс `EntitySet` аналогичен стандартному запросу со встроенным оператором `Where`, извлекающим нужные сущности. Атрибут `[Association]` предоставляет информацию, необходимую для запроса LINQ к SQL. Как обычно, выполнение этого запроса откладывается. Иными словами, если вы пользуетесь классом `EntitySet`, запрос не выполняется, пока вы не приступите к перебору элементов соответствующей коллекции.

Вот как выглядит свойство `Purchases.Customer` на другом конце отношения:

```
[Association (Storage="_Customer",
              ThisKey="CustomerID",
              IsForeignKey=true)]
public Customer Customer { get {...} set {...} }
```

Хотя это свойство имеет тип `Customer`, поле `_Customer`, лежащее в его основе, имеет тип `EntityRef`. Этот тип реализует отложенную загрузку, поэтому соответствующий объект `Customer` не извлекается из базы данных, пока вы не запросите его.

Отложенное выполнение запросов LINQ к SQL

Как и в случае с локальными запросами, в отношении запросов LINQ к SQL действует принцип отложенного выполнения. Это позволяет вам строить запросы последовательно. Однако в одном аспекте запросы LINQ к SQL имеют особую семантику отложенного выполнения. Речь идет о подзапросах, входящих в состав выражения `Select`:

- у локальных запросов получается "двойное" отложенное выполнение, потому что с функциональной точки зрения вы имеете последовательность *запросов*. Поэтому, если вы перебираете элементы внешней последовательности с результатами, но не перебираете элементы внутренних последовательностей, подзапрос никогда не будет выполнен;

- подзапрос запроса LINQ к SQL выполняется в то же время, что и внешний запрос. Это позволяет сократить количество пересылок данных на сервер и обратно.

Например, следующий запрос будет выполнен за одну пересылку данных туда и обратно, как только управление будет передано первому оператору `foreach`:

```
var dataContext = new MyTypedDataContext
("...");


var query = from c in dataContext.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };
foreach (var customerPurchaseResults in query)
  foreach (var namePrice in
customerPurchaseResults)
    Console.WriteLine (namePrice.Name + " spent
" +
                       namePrice.Price);
```

Любые объекты `EntitySet`, которые вы явно проецируете, полностью заполняются за одну пересылку данных на сервер и обратно:

```
var query = from c in dataContext.Customers
            select new { c.Name, c.Purchases };


foreach (var row in query)
  foreach (Purchase p in row.Purchases)
    Console.WriteLine (row.Name + " spent " +
p.Price);
```

Однако, если мы будем перебирать свойства класса `EntitySet` без предварительного проецирования,

вступает в силу принцип отложенного выполнения. В следующем примере запрос LINQ к SQL выполняет еще один запрос `Purchases` на каждом шаге цикла:

```
foreach (Customer c in dataContext.Customers)
  foreach (Purchase p in c.Purchases) // пересылка данных туда и обратно
    Console.WriteLine (c.Name + " spent " + p.Price);
```

Эта модель предпочтительнее, когда вы хотите *избирательно* выполнять внутренний цикл, в зависимости от результатов проверки, которая может быть выполнена только на стороне клиента:

```
foreach (Customer c in dataContext.Customers)
  if (myWebService.HasBadCreditHistory (c.ID))
   foreach (Purchase p in c.Purchases) // пересылка данных туда и обратно
      Console.WriteLine (...);
```

В *разд. "Проецирование"* мы более подробно рассмотрим подзапросы `Select`.

Класс *DataLoadOptions*

Класс `DataLoadOptions` используется двояко:

- он позволяет вам заранее указать фильтр для ассоциаций в объектах `EntitySet` (метод `AssociateWith`);
- он позволяет вам потребовать предварительную ("нетерпеливую") загрузку некоторых объектов `EntitySet` (методом `LoadWith`) для минимизации.

Предварительное указание фильтра

Рассмотрим использование метода `AssociateWith` класса `DataLoadOptions`:

```
DataLoadOptions options = new DataLoadOptions();
options.AssociateWith< Customer>
  (c => c.Purchases.Where (p => p.Price >
1000));
dataContext.LoadOptions =options;
```

Здесь экземпляр `DataContext` получает предписание всегда фильтровать свойство `Purchases` класса `Customer` с учетом заданного предиката.

Метод `AssociateWith` не влияет на семантику отложенного выполнения. Когда используется конкретное отношение, оно воспринимается как инструкция неявно добавить в уравнение соответствующий фильтр.

"Нетерпеливая" загрузка

Второе применение класса `DataLoadOptions` заключается в "нетерпеливой" загрузке некоторых объектов `EntitySet` вместе с их родителем. Например, предположим, что вам нужно загрузить информацию обо всех клиентах и их покупку за одну пересылку SQL-данных на сервер и обратно. Следующий код делает как раз то, что нужно:

```
DataLoadOptions options = new DataLoadOptions();
options.LoadWith <Customer> (c => c.Purchases);
dataContext.LoadOptions = options;

foreach (Customer c in dataContext.Customers)
  foreach (Purchase p in c.Purchases)
```

```
    Console.WriteLine (c.Name + " bought a " +
                       p.Description);
```

Здесь фактически говорится, что, как только извлекается объект `Customer`, его свойство `Purchases` должно быть тоже извлечено. Вы можете запросить загрузку не только "детей", но и "внуков":

```
options.LoadWith <Customer> (c => c.Purchases);
options.LoadWith <Purchase> (p =>
p.PurchaseItems);
```

Вы можете комбинировать методы `LoadWith` и `AssociateWith`. В следующем коде дается предписание, что, как только извлекается информация о покупателе, тут же должна быть извлечена информация о его дорогих приобретениях:

```
options.LoadWith <Customer> (c => c.Purchases);
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price >
1000));
```

Обновления

Технология запросов LINQ к SQL отслеживает изменения, которые вы вносите в сущности, и позволяет вам сохранять их в базе данных с помощью метода `SubmitChanges` объекта `DataContext`. Класс `Table<>` предоставляет методы `InsertOnSubmit` и `DeleteOnSubmit` для вставки и удаления строк таблицы. Вот, например, как нужно вставлять строку:

```
var dataContext = new MyTypedDataContext ("cx
string");

Customer cust = new Customer { ID=1000,
Name="Bloggs" };
```

```
dataContext.Customers.InsertOnSubmit (cust);
dataContext.SubmitChanges();
```

Впоследствии мы сможем прочитать эту строку, а затем удалить ее:

```
var dataContext = new MyTypedDataContext
("...");

Customer cust = dataContext.Customers.Single
                (c => c.ID == 1000);
cust.Name = "Bloggs2";
dataContext.SubmitChanges();          // Обновляет информацию о клиенте

dataContext.Customers.DeleteOnSubmit (cust);
dataContext.SubmitChanges();          // Удаляет информацию о клиенте
```

Метод `DataContext.SubmitChanges` собирает вместе все изменения, которые были внесены в сущности с момента создания объекта `DataContext` (или с момента последнего вызова метода `SubmitChanges`), а затем выполняет SQL-оператор, записывающий их в базу данных. Учитывается значение `TransactionScope`, но если оно недоступно, все операторы заключаются в новую транзакцию.

Вы также можете добавить в объект `EntitySet` новые или уже существующие строки, вызывая метод `Add`. Запрос LINQ к SQL автоматически заполняет внешние ключи, когда вы вызываете этот метод:

```
var p1 = new Purchase { ID=100,
Description="Bike",
                        Price=500 };
var p2 = new Purchase { ID=101,
Description="Tools",
                        Price=100 };
```

```
Customer cust = dataContext.Customers.Single
                (c => c.ID == 1);

cust.Purchases.Add (p1);
cust.Purchases.Remove (p2);

dataContext.SubmitChanges();      // Заносит ин-
формацию о покупках
```

> ***Примечание***
>
> Если вы не хотите усложнять себе жизнь генерированием уникальных ключей, вы можете воспользоваться полем с автоувеличением (IDENTITY на сервере SQL Server) или методом `Guid` для получения первичного ключа.

В следующем примере запрос LINQ к SQL автоматически записывает 100 в поле `CustomerID` ("идентификатор клиента") каждой новой сущности `Purchase` ("покупка"). Методу известно, что это нужно сделать, потому что мы определили ассоциирование для свойства `Purchases`:

```
[Association (Storage="_Purchases",
              OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases
{ get {...} set {...}
```

Если бы сущности `Customer` и `Purchase` были сгенерированы в Visual Studio или с помощью SqlMetal, сгенерированные классы содержали бы дополнительный код для синхронизации обеих сторон отношения. Иными словами, присваивание значения свойству `Purchase.Customer` приводило бы к автоматическому добавлению нового клиента

в набор сущностей `Customer.Purchases`, и наоборот. Мы можем проиллюстрировать это утверждение, переписав предыдущий код таким образом:

```
var dataContext = new MyTypedDataContext
("...");


Customer cust = dataContext.Customers.Single
                (c => c.ID == 1);
new Purchase { ID=100, Description="Bike",
Price=500,
               Customer=cust  };
new Purchase { ID=101, Description="Tools",
Price=100,
               Customer=cust  };
dataContext.SubmitChanges();     // Вставляет ин-
формацию о покупках
```

Когда вы удаляете строку из `EntitySet`, поле внешнего ключа автоматически получает значение `null`. В следующем коде отменяется ассоциирование двух покупок с клиентом, сделавшим их:

```
var dataContext = new MyTypedDataContext
("...");


Customer cust = dataContext.Customers.Single
                (c => c.ID == 1);


cust.Purchases.Remove
  (cust.Purchases.Single (p => p.ID == 100));
cust.Purchases.Remove
  (cust.Purchases.Single (p => p.ID == 101));


dataContext.SubmitChanges( );  // отправить
SQL-запрос на сервер
```

Поскольку здесь делается попытка присвоить значение `null` полю `CustomerID` у каждой покупки, свойство `Purchase.CustomerID` должно иметь тип, допускающий значение `null`; в противном случае будет возбуждено исключение. (Более того, поле или свойство `CustomerID` в классе сущности тоже должно иметь тип, допускающий значение `null`.)

Что касается дочерних сущностей, их надо удалять из таблицы `Table<>`:

```
Customer cust = dataContext.Customers.Single
                (c => c.ID == 1);

var dc = dataContext;
dc.Purchases.DeleteOnSubmit
  (dc.Purchases.Single (p => p.ID == 100));
dc.Purchases.DeleteOnSubmit
  (dc.Purchases.Single (p => p.ID == 101));

dataContext.SubmitChanges();          // отпра-
вить SQL-запрос на сервер
```

Построение выражений запросов

До сих пор мы при помощи условий выстраивали цепочку операторов запросов, когда нам нужно было составлять запросы динамически. Хотя этот подход приемлем во многих ситуациях, иногда приходится выполнять более тонкую работу и динамически строить лямбда-выражения, которые затем передаются операторам.

В этом разделе мы будем работать с классом `Product`:

```
[Table] public partial class Product
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string
Description;
  [Column]                    public bool
Discontinued;
  [Column]                    public DateTime
LastSale;
}
```

Делегаты и деревья выражений

Вспомним, что

- локальные запросы, в которых используются операторы типа `Enumerable`, принимают делегаты;
- интерпретируемые запросы, в которых используются операторы типа `Queryable`, принимают деревья выражений.

Мы можем убедиться в этом, сравнив сигнатуры оператора `Where` для типов `Enumerable` и `Queryable`:

```
public static IEnumerable<TSource>
Where<TSource> (this
  IEnumerable<TSource> source,
  Func<TSource,bool> predicate)


public static IQueryable<TSource> Where<TSource>
(this
```

```
  IQueryable<TSource> source,
  Expression<Func<TSource,bool>> predicate)
```

Будучи встроенными в запрос, лямбда-выражение выглядит одинаково в операторах `Enumerable` и `Queryable`:

```
IEnumerable<Product> q1 = localProducts.Where
                          (p =>
!p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where
                          (p =>
!p.Discontinued);
```

Однако, когда вы присваиваете лямбда-выражение промежуточной переменной, вы должны явно указать, должно ли оно быть оттранслировано в делегат (то есть `Func<>`) или в дерево выражений (то есть `Expression<Func<>>`).

Компиляция деревьев выражений

Вы можете преобразовать дерево выражений в делегат, вызвав метод `Compile`. Это особенно удобно, когда вы пишете методы, возвращающие многократно используемые выражения. Чтобы проиллюстрировать такой подход, мы добавим в класс `Product` статический метод, который возвращает предикат, принимающий значение `true`, если товар не снят с производства и был продан в течение последних 30 дней:

```
public partial class Product
{
  public static Expression<Func<Product, bool>>
  IsSelling()
```

```
  {
    return p => !p.Discontinued &&
                p.LastSale >
DateTime.Now.AddDays (-30);
  }
}
```

(Мы определили это в отдельном частичном классе, чтобы основной класс не был переписан автоматическим генератором объектов `DataContext`, таким как генератор запросов LINQ к SQL в Visual Studio.)

Метод, который мы здесь представили, можно использовать как с интерпретируемыми, так и с локальными запросами:

```
void Test()
{
  var dataContext = new MyTypedDataContext
("...");
  Product[] localProducts =
    dataContext.Products.ToArray();

  IQueryable<Product> sqlQuery =
    dataContext.Products.Where
(Product.IsSelling());

  IEnumerable<Product> localQuery =
    localProducts.Where
(Product.IsSelling.Compile());
}
```

> ***Примечание***
>
> Преобразование в обратном направлении, от делегата к дереву выражений, невозможно. Деревья выражений — более гибкая конструкция.

Оператор *AsQueryable*

Оператор `AsQueryable` позволяет вам писать целые запросы, которые могут работать как с локальными, так и с удаленными последовательностями:

```
IQueryable<Product> FilterSortProducts
  (IQueryable<Product> input)
{
   return from p in input
          where ...
          order by ...
          select p;
}

void Test()
{
  var dataContext = new MyTypedDataContext
("...");
Product[]localProducts =
    dataContext.Products.ToArray();

  var sqlQuery =
    FilterSortProducts (dataContext.Products);
  var localQuery =
    FilterSortProducts
(localProducts.AsQueryable());
  ...
}
```

Оператор `AsQueryable` создает вокруг локальной последовательности оболочку `IQueryable<>`, чтобы последующие операторы запросов транслировались в деревья выражений. Когда вы впоследст-

вии будете перебирать элементы результата, деревья выражений будут откомпилированы неявным образом, и элементы локальной последовательности будут перебираться как обычно.

Деревья выражений

Ранее мы говорили, что присваивание лямбда-выражения переменной типа `Expression<TDelegate>` заставляет компилятор C# сгенерировать дерево выражений. С теми же затратами вы можете добиться этого вручную на этапе выполнения (то есть динамически построить дерево выражений "с нуля"). Результат можно привести к типу `Expression<TDelegate>` и использовать в запросах LINQ к SQL, но можно и откомпилировать в обычный делегат, вызвав метод `Compile`.

Объектная модель документов для выражений

Дерево выражений является миниатюрной объектной моделью документов. Каждый узел дерева представлен типом из пространства имен `System.Linq.Expressions`. Все типы приводятся на рис. 8.

Базовым классом для всех узлов является необобщенный класс `Expression`. Обобщенный класс `Expression<TDelegate>` фактически является "типизированным лямбда-выражением", и его можно

было бы назвать `LambdaExpression<TDelegate>`, если бы это не приводило к таким громоздким конструкциям, как, например, эта:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

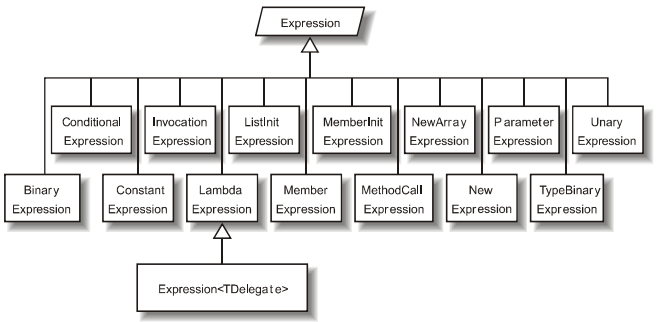

Рис. 8. Типы выражений

Базовым типом для `Expression<>` является необобщенный класс `LambdaExpression`. Он обеспечивает унификацию типов для деревьев лямбда-выражений: любой типизированный объект `Expression<>` может быть приведен к `LambdaExpression`.

Чертой, отличающей объекты класса `LambdaExpression` от обычных объектов `Expression`, является наличие *параметров* у лямбда-выражений. Чтобы создать дерево выражений, не нужно создавать экземпляры узловых типов напрямую. Вызывайте статические методы, предоставляемые классом `Expression`.

Вот их список:

`Add`	`MakeBinary`
`AddChecked`	`MakeMemberAccess`
`And`	`MakeUnary`
`AndAlso`	`MemberBind`
`ArrayIndex`	`MemberInit`
`ArrayLength`	`Modulo`
`Bind`	`Multiply`
`Call`	`MultiplyChecked`
`Coalesce`	`Negate`
`Condition`	`NegateChecked`
`Constant`	`New`
`Convert`	`NewArrayBounds`
`ConvertChecked`	`NewArrayInit`
`Divide`	`Not`
`ElementInit`	`NotEqualOr`
`Equal`	`OrElse`
`ExclusiveOr`	`Parameter`
`Field`	`Power`
`GreaterThan`	`Property`
`GreaterThanOrEqual`	`PropertyOrField`
`Invoke`	`Quote`
`Lambda`	`RightShift`
`LeftShift`	`Subtract`
`LessThan`	`SubtractChecked`
`LessThanOrEqual`	`TypeAs`
`ListBind`	`TypeIs`
`ListInit`	`UnaryPlus`

На рис. 9 представлено дерево выражений, создаваемое в результате такого присваивания:

```
Expression<Func<string, bool>> f = s => s.Length
< 5;
```

Это можно продемонстрировать следующим образом:

```
Console.WriteLine (f.Body.NodeType);      //
LessThan
Console.WriteLine
  (((BinaryExpression) f.Body).Right);    // 5
```

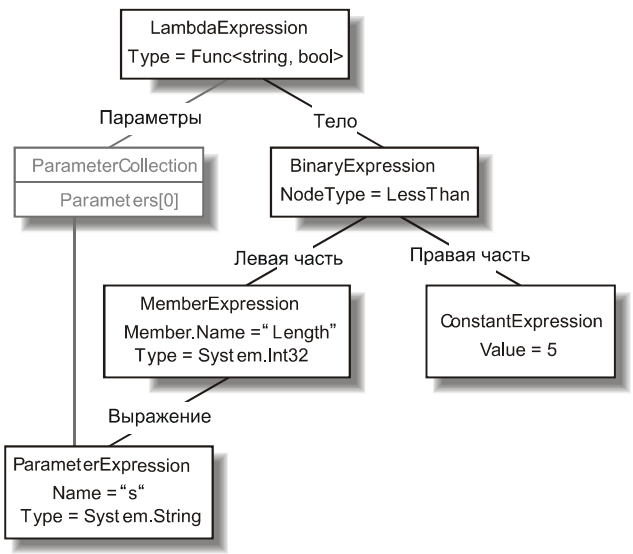

Рис. 9. Дерево выражений

Давайте теперь построим это выражение вручную. Будем придерживаться принципа "снизу вверх". На самом нижнем уровне дерева находится

`ParameterExpression`, параметр лямбда-выражения по имени "s", имеющий тип `string`:

```
ParameterExpression p = Expression.Parameter
  (typeof (string), "s");
```

Следующий шаг состоит в построении выражений `MemberExpression` и `ConstantExpression`. В первом случае нам нужно обратиться к свойству `Length` параметра "s":

```
MemberExpression stringLength =
  Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant
(5);
```

Теперь настала очередь сравнения `LessThan`:

```
BinaryExpression comparison =
  Expression.LessThan (stringLength, five);
```

Заключительный шаг — составление лямбда-выражения, связывающего тело выражения с коллекцией параметров:

```
Expression<Func<string, bool>> lambda =
  Expression.Lambda<Func<string, bool>>
(comparison, p);
```

Удобным способом тестирования лямбда-выражения является компиляция его в делегат:

```
Func<string, bool> runnable = lambda.Compile();
Console.WriteLine (runnable ("kangaroo"));  //
False
Console.WriteLine (runnable ("dog"));       //
True
```

Примечание

Простейший способ выяснить, какой тип выражения нужно использовать, заключается в

изучении имеющегося лямбда-выражения с помощью отладчика Visual Studio.

Мы продолжаем обсуждение выражений на нашем сайте **http://www.albahari.com/expressions/**.

Обзор операторов

В этом разделе дано описание операторов запросов LINQ, перечисленных в табл. 1.

Таблица 1. *Операторы запросов LINQ*

Категория	Оператор
Фильтрация	`Where`, `Distinct`, `Take`, `TakeWhile`, `Skip`, `SkipWhile`
Проецирование	`Select`, `SelectMany`
Объединение	`Join`, `GroupJoin`
Упорядочивание	`OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `Reverse`
Группирование	`GroupBy`
Операции над множествами	`Concat`, `Union`, `Intersect`, `Except`
Методы преобразования (импорт)	`OfType`, `Cast`
Методы преобразования (экспорт)	`ToArray`, `ToList`, `ToDictionary`, `ToLookup`, `AsEnumerable`, `AsQueryable`

Таблица 1 *(окончание)*

Категория	Оператор
Поэлементные операции	`First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`, `ElementAt`, `ElementAtOrDefault`, `DefaultIfEmpty`
Методы агрегирования	`Aggregate`, `Average`, `Count`, `LongCount`, `Sum`, `Max`, `Min`
Квантификаторы	`All`, `Any`, `Contains`, `SequenceEqual`
Методы генерирования коллекций	`Empty`, `Range`, `Repeat`

Во всех примерах предполагается, что массив `names` определен следующим образом:

```
string[] names = { "Tom", "Dick", "Harry",
"Mary", "Jay" };
```

В примерах с запросами LINQ к SQL используется переменная `dataContext` типа `DataContext`:

```
var dataContext = new DemoDataContext;
...
public class DemoDataContext : DataContext
{
  public DemoDataContext (string cxString)
    : base (cxString) {}
  public Table<Customer> Customers
   { get { return GetTable<Customer>; } }
  public Table<Purchase> Purchases
   { get { return GetTable<Purchase>; } }
}
```

```
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)]  public int ID;
  [Column]                     public string
Name;
  [Association (OtherKey="CustomerID")]
  public EntitySet<Purchase> Purchases
    = new EntitySet<Purchase>;
}
[Table] public class Purchase
{
    [Column(IsPrimaryKey=true)]  public int ID;
    [Column]                     public int?
CustomerID;
    [Column]                     public string
Description;
    [Column]                     public decimal
Price;
    [Column]                     public DateTime
Date;
  EntityRef<Customer> custRef;
  [Association (Storage="custRef",
                ThisKey="CustomerID",
                IsForeignKey=true)]
  public Customer Customer
  {
    get { return custRef.Entity; }
    set { custRef.Entity = value; }
  }
}
```

Примечание

Классы сущностей для запросов LINQ к SQL представлены в упрощенном виде по сравне-

нию с тем, что обычно генерируется автоматическими инструментами: они не содержат код, обновляющий противоположную сторону отношения, когда их сущностям присваиваются новые значения.

Соответствующие определения SQL-таблиц выглядят так:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer (ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Фильтрация

Метод	Описание	Эквивалентен SQL
`Where`	Возвращает подмножество элементов, удовлетворяющих данному условию	`WHERE`
`Take`	Возвращает первые `count` элементов и игнорирует остальные	`WHERE ROW_NUMBER()`… *или* `TOP n` подзапрос

(окончание)

Метод	Описание	Эквивалентен SQL
`Skip`	Игнорирует первые `count` элементов и возвращает остальные	`WHERE ROW_NUMBER()…` *или* `NOT IN (SELECT TOP n…)`
`TakeWhile`	Возвращает элементы из входной последовательности, пока предикат равен `true`	Возбуждается исключение
`SkipWhile`	Игнорирует элементы из входной последовательности, пока предикат равен `true`, а затем возвращает остальные	Возбуждается исключение
`Distinct`	Возвращает коллекцию, из которой исключены повторяющиеся элементы	`SELECT DISTINCT…`

Примечание

В справочных таблицах столбец "Эквивалентен SQL" необязательно соответствует тому, что генерирует реализация интерфейса `IQueryable`, например, технология запросов LINQ к SQL. В этом столбце показано, что вы, скорее всего, написали бы, если бы составляли SQL-запросы вручную. Когда очевидное соответствие отсутствует, столбец оставлен пустым. Если же соответствие невозможно в принципе, в столбце написано "Возбуждается исключение".

> Когда приводится код реализации `Enumerable`, в нем опущены проверка аргументов на значение `null` и индексация предикатов.

Каким бы методом фильтрации вы ни пользовались, вы всегда получите либо то же самое, либо меньшее количество элементов по сравнению с оригинальной последовательностью. Вы никогда не получите больше! Кроме того, элементы на выходе идентичны оригинальным; они никак не преобразуются.

Оператор *Where*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат	`TSource => bool` или `(TSource,int) => bool` *

* Запрещено в запросах LINQ к SQL.

Синтаксис, облегчающий восприятие

`where` *`булево_выражение`*

Описание

Оператор `Where` возвращает элементы входной последовательности, удовлетворяющие заданному предикату.

Например:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query =
  names.Where (name => name.EndsWith ("y"));
// Результат: { "Harry", "Mary", "Jay" }
```

В синтаксисе, облегчающем восприятие, этот запрос выглядит так:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith
("y")
                            select n;
```

Конструкция `where` может появиться в запросе более одного раза, причем она может чередоваться с конструкцией `let`:

```
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
select u;        // Результат: { "HARRY", "MARY"
}
```

Для таких операторов действуют стандартные правила относительно области видимости, принятые в C#. Иными словами, вы не можете ссылаться на переменную до ее появления с помощью переменной итерации или конструкции `let`.

Индексированная фильтрация

Предикат оператора `Where` может принимать второй (необязательный) аргумент типа `int`. Он указывает позицию каждого элемента входной последовательности, а предикат может использовать эту

информацию, принимая решение по поводу фильтрации. Например, в следующем коде пропускается каждый второй элемент:

```
IEnumerable<string> query =
  names.Where ((n, i) => i % 2 == 0);
// Результат: { "Tom", "Harry", "Jay" }
```

При попытке воспользоваться индексированной фильтрацией в запросе LINQ к SQL возбуждается исключение.

Оператор *Where* в запросе LINQ к SQL

Следующие методы класса `string` транслируются в оператор `LIKE` языка SQL:

```
Contains, StartsWith, EndsWith
```

Например, запрос `c.Name.Contains ("abc")` транслируется в `customer.Name LIKE '%abc%'` (точнее говоря, в параметризованную версию этого кода). Вы можете выполнить более сложное сравнение, вызвав метод `SqlMethods.Like`, который отображается непосредственно в SQL-оператор `LIKE`. Вы также можете выяснить порядок следования строк с помощью метода `CompareTo` класса `string`. Он отображается в SQL-операторы < и >:

```
dataContext.Purchases.Where (p =>
p.Description.CompareTo
("C") < 0)
```

Технология запросов LINQ к SQL позволяет вам применять оператор `Contains` к локальной коллекции внутри предиката фильтра. Например:

```
string[] chosenOnes = { "Tom", "Jay" };
from c in dataContext.Customers
```

```
where chosenOnes.Contains (c.Name)
...
```

Этот код отображается в SQL-оператор `IN`, то есть:

```
WHERE customer.Name IN ("Tom", "Jay")
```

Если локальная коллекция является массивом сущностей или элементов нескалярного типа, запрос LINQ к SQL может породить оператор `EXISTS`.

Операторы *Take* и *Skip*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Количество элементов, которые необходимо возвратить или пропустить	`int`

Оператор `Take` возвращает первые n элементов и игнорирует остальные, а оператор `Skip` игнорирует первые n элементов и возвращает остальные. Эти два метода полезны в сочетании, когда вы реализуете веб-страницу, позволяющую пользователю просматривать большое количество записей, соответствующих его запросу. Предположим, например, пользователь ищет в базе данных книги, в названиях которых встречается слово "mercury" (ртуть), и таковых оказывается 100 штук.

Следующий запрос возвращает первые 20 названий:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

А этот запрос возвращает названия с 21 по 40:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

Запрос LINQ к SQL транслирует операторы `Take` и `Skip` в функцию `ROW_NUMBER` на сервере SQL Server 2005 или в подзапрос `TOP` *n* на более ранних версиях этого сервера.

Операторы *TakeWhile* и *SkipWhile*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат	`TSource => bool` или `(TSource,int) => bool`

Оператор `TakeWhile` перебирает элементы входной последовательности и возвращает каждый из них, пока заданный предикат имеет значение `true`.

После этого все остальные элементы игнорируются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100);
// Результат: { 3, 5, 2 }
```

Оператор `SkipWhile` перебирает элементы входной последовательности, пропуская их, пока заданный предикат имеет значение `true`. После этого он возвращает все оставшиеся элементы:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100);
// Результат: { 234, 4, 1 }
```

Операторы `TakeWhile` и `SkipWhile` не имеют соответствий в SQL, и их использование в запросе LINQ к SQL приводит к возникновению исключения.

Оператор *Distinct*

Оператор `Distinct` возвращает входную последовательность, избавленную от повторяющихся элементов. Для выяснения равенства можно использовать только класс, установленный по умолчанию. Следующий код возвращает неповторяющиеся буквы строки:

```
char[] distinctLetters =
  "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters);    // HeloWrd
```

Можно вызывать методы LINQ непосредственно для строки, потому что тип `string` реализует интерфейс `IEnumerable<char>`.

Проецирование

Метод	Описание	Эквивалентен SQL
`Select`	Преобразует каждый входной элемент в соответствии с лямбда-выражением	`SELECT`
`SelectMany`	Преобразует каждый входной элемент, а затем делает плоскими и конкатенирует полученные последовательности	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`

Примечание

Для запросов LINQ к SQL операторы `Select` и `SelectMany` являются самыми гибкими конструкциями, а для локальных запросов самыми *эффективными* объединяющими конструкциями являются операторы `Join` и `GroupJoin`.

Оператор *Select*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`

(окончание)

Аргумент	Тип
Селектор результата	`TSource => TResult` или `(TSource,int) => TResult`[a]

[a] Запрещено в запросах LINQ к SQL.

* Синтаксис, облегчающий восприятие

```
select выражение_проецирования
```

Описание

От оператора `Select` вы всегда получаете то количество элементов, которое было на входе. Однако каждый элемент может быть как угодно преобразован лямбда-функцией.

В следующем коде выбираются названия всех шрифтов, установленных на компьютере (из пространства имен `System.Drawing`):

```
IEnumerable<string> query =
  from f in FontFamily.Families
  select f.Name;
foreach (string name in query) Console.WriteLine
(name);
```

В этом примере предложение `select` преобразует объект `FontFamily` в его имя. Вот лямбда-эквивалент этого кода:

```
IEnumerable<string> query =
  FontFamily.Families.Select (f => f.Name);
```

Операторы `Select` часто используются для проецирования в анонимный тип:

```
var query =
  from f in FontFamily.Families
  select new
  {
    f.Name,
    LineSpacing = f.GetLineSpacing
(FontStyle.Bold)
  };
```

Проекция без преобразования иногда применяется в запросах, составленных в синтаксисе, облегчающем восприятие, для удовлетворения требования, гласящего, что запрос должен заканчиваться конструкцией `select` или `group`. В следующем коде выбираются шрифты, поддерживающие перечеркивание символов:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;
foreach (FontFamily ff in query)
  Console.WriteLine (ff.Name);
```

В таких случаях компилятор опускает проецирование при трансляции запроса в лямбда-синтаксис.

Индексированная проекция

Выражение-селектор может принимать необязательный аргумент, который действует как индексатор, указывающий позицию каждого элемента во

входной последовательности. Это возможно только в локальных запросах:

```
string[] names = {
"Tom","Dick","Harry","Mary","Jay" };
IEnumerable<string> query = names
  .Select ((s,i) => i + "=" + s);
// Результат: { "0=Tom", "1=Dick", ... }
```

Подзапросы *Select* и иерархии объектов

Вы можете вложить подзапрос в конструкцию `select`, чтобы построить иерархию объектов. В следующем коде возвращается коллекция, описывающая каждый каталог в каталоге D:\source, и подколлекция, соответствующая файлам в этих каталогах:

```
DirectoryInfo[] dirs =
  new DirectoryInfo
(@"d:\source").GetDirectories();
var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System)
== 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,
    Files = from f in d.GetFiles()
            where (f.Attributes &
FileAttributes.Hidden) == 0
            select new { FileName = f.Name,
f.Length, }
  };
```

Внутреннюю часть этого запроса можно назвать *коррелированным подзапросом*. Подзапрос является коррелированным, если он ссылается на объект во внешнем запросе. В данном случае он ссылается на `d`, каталог, чьи элементы подвергаются перебору.

> ***Примечание***
>
> Подзапрос внутри оператора `Select` позволяет отобразить одну иерархию объектов на другую или отразить реляционную объектную модель на иерархическую.

В локальных запросах подзапрос, вложенный в оператор `Select`, приводит к "вдвойне" отложенному выполнению. В нашем примере файлы ни фильтруются, ни проецируются, пока не начнет работать внутренний оператор `foreach`.

Подзапросы и объединения в запросах LINQ к SQL

Проекции, вызванные подзапросами, хорошо совместимы с технологией запросов LINQ к SQL и могут быть использованы для выполнения объединений в духе SQL. В следующем коде мы получаем имя каждого клиента и информацию о его самой дорогой покупке:

```
var query =
  from c in dataContext.Customers
  select new {
               c.Name,
```

```
                Purchases = from p in
dataContext.Purchases
                            where p.CustomerID ==
c.ID && p.Price > 1000
                            select new {
p.Description, p.Price }
            };
```

Примечание

Такой стиль идеально годится для интерпретируемых запросов. В запросе LINQ к SQL внешний запрос и подзапрос обрабатываются как единое целое, что позволяет избежать лишних пересылок данных на сервер и обратно. Однако с локальными запросами это решение неэффективно, потому что приходится перебирать все возможные комбинации внешних и внутренних элементов ради получения нескольких комбинаций. Более удачным решением для локальных запросов будет использование операторов `Join` и `GroupJoin`, описанных в следующих разделах.

Этот запрос ставит в соответствие друг другу объекты из двух разных коллекций, и к нему можно относиться как к определенному виду "объединения". Отличие такого объединения от традиционного объединения (или подзапроса), принятого в базах данных, состоит в том, что здесь мы не преобразуем выходную последовательность в плоский (двухмерный) результирующий набор. Мы отображаем реляционную структуру данных в иерархическую, а не плоскую.

Приведем тот же запрос, упрощенный за счет использования ассоциативного свойства `Purchases` сущности `Customer`:

```
from c in dataContext.Customers
select new
{
  c.Name,
  Purchases = from p in c.Purchases
              where p.Price > 1000
              select new { p.Description,
p.Price }
};
```

Оба запроса аналогичны левому внешнему объединению SQL в том смысле, что мы получаем всех клиентов во внешнем перечислении, независимо от того, совершали ли они покупки. Для эмуляции внутреннего объединения (при котором клиенты, не сделавшие дорогих покупок, будут исключены) нам понадобится добавить фильтрующее условие к коллекции покупок:

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
            c.Name,
            Purchases =
              from p in c.Purchases
              where p.Price > 1000
              select new { p.Description,
p.Price }
          };
```

Получилось не вполне элегантно, потому что один и тот же предикат (`Price > 1000`) написан дважды.

Этого можно избежать с помощью конструкции `let`:

```
from c in dataContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description,
p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

Это достаточно гибкий стиль построения запросов. Изменив `Any` на `Count`, например, мы сможем так модифицировать запрос, что он будет возвращать только клиентов, совершивших, как минимум, две дорогих покупки):

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

Проецирование в конкретные типы

Проецирование в анонимные типы полезно для получения промежуточных результатов, однако он теряет смысл, когда вы, например, хотите отправить результирующий набор клиенту, потому что анонимные типы могут существовать только как локальные переменные внутри метода. Альтернативным решением является проецирование в конкретные типы, такие как `DataSet` или пользовательские классы бизнес-сущностей. Пользовательский класс бизнес-сущности — это класс, который вы пишете, добавляя в него некоторые

свойства. Он аналогичен аннотированному классу `[Table]` в технологии запросов LINQ к SQL, но разрабатывается специально для того, чтобы скрыть детали нижнего уровня (относящиеся к базе данных). Например, вы можете исключить из класса бизнес-сущности поля с внешними классами. Предположив, что у нас уже есть пользовательские классы сущностей `CustomerEntity` и `PurchaseEntity`, мы можем проецировать результаты в них:

```
IQueryable<CustomerEntity> query =
  from c in dataContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases = (
      from p in c.Purchases
      where p.Price > 1000
      select new PurchaseEntity
      {
        Description = p.Description,
        Value = p.Price
      }
    ).ToList()
  };
// Форсировать выполнение запроса, преобразуя
// выходную информацию в более удобный тип List:
List<CustomerEntity> result = query.ToList();
```

Обратите внимание, что до сих пор нам не приходилось пользоваться операторами `Join` или `SelectMany`. Дело в том, что мы сохраняем иерархическую структуру данных, как показано на

рис. 10. В технологии LINQ вы часто можете обойтись без традиционного для SQL подхода, состоящего в переводе таблиц в двухмерный результирующий набор.

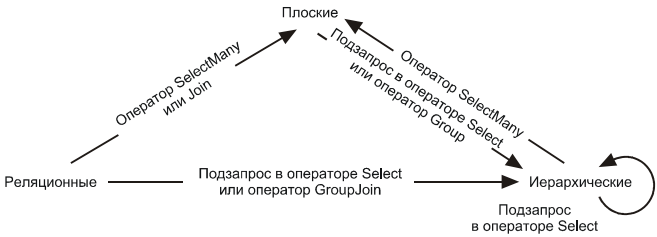

Рис. 10. Проецирование иерархии объектов

Оператор *SelectMany*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата	`TSource => IEnumerable<TResult>` **или** `(TSource,int) => IEnumerable<TResult>` [a]

[a] Запрещено в запросах LINQ к SQL.

Синтаксис, облегчающий восприятие

```
from идентификатор1 in перечисляемое_выражение1
from идентификатор2 in перечисляемое_выражение2
```

Описание

Оператор `SelectMany` выполняет конкатенацию подпоследовательностей в одну плоскую выходную последовательность.

Вспомним, что для каждого входного элемента оператор `Select` возвращает ровно один выходной элемент. В отличие от него оператор `SelectMany` возвращает от 0 до *n* выходных элементов. Эти элементы поступают из дочерней последовательности (то есть подпоследовательности), генерируемой лямбда-выражением.

Оператор `SelectMany` можно применять для расширения дочерних последовательностей, перевода вложенных коллекций в плоскую форму и объединения двух коллекций в плоскую выходную последовательность. Вспомнив аналогию с конвейерной линией, заметим, что оператор `SelectMany` поставляет необработанный материал на конвейер. Каждый входной элемент "включает" поставку необработанного материала. Этот материал подается селекторным лямбда-выражением и должен быть последовательностью. Иными словами, лямбда-выражение должно генерировать дочернюю последовательность на каждый входной элемент. Окончательный результат является конкатенацией дочерних последовательностей, сгенерированных для входных элементов.

Начнем с простейшего примера и предположим, что у нас имеется массив полных имен:

```
string[] fullNames =
  { "Anne Williams", "John Fred Smith", "Sue
Green" };
```

Мы хотим преобразовать его в плоскую коллекцию слов:

```
"Anne", "Williams", "John", "Fred", "Smith",
"Sue", "Green"
```

Оператор `SelectMany` идеально подходит для этой задачи, потому что каждый входной элемент должен быть отображен в переменное количество выходных элементов. Все, что мы должны сделать, — это написать селекторное выражение, преобразующее каждый входной элемент в дочернюю последовательность. С такой работой справится метод `string.Split`: он принимает строку и разбивает ее на слова, возвращая результат в виде массива:

```
string testInputElement = "Anne Williams";
string[] childSequence  =
testInputElement.Split();
// childSequence содержит { "Anne", "Williams"
};
```

Вот как будет выглядеть наш запрос `SelectMany` и возвращаемый им результат:

```
IEnumerable<string> query =
  fullNames.SelectMany (name => name.Split());
foreach (string name in query)
  Console.Write (name + "|");
// Результат:
Anne|Williams|John|Fred|Smith|Sue|Green|
```

Примечание

Если вы замените здесь оператор `SelectMany` на `Select`, вы получите такой же результат в иерархическом виде. Следующий код возвращает последовательность строковых массивов,

для перебора которой потребуются вложенные операторы `foreach`:

```
IEnumerable<string[]> query =
  fullNames.Select (
    name => name.Split());
foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write (name + "/");
```

Достоинство оператора `SelectMany` в том, что он возвращает одну *плоскую* последовательность.

Оператор `SelectMany` поддерживается синтаксисом, облегчающим восприятие, в котором он вызывается с помощью *дополнительного генератора*, то есть дополнительного предложения `from` в запросе. Ключевое слово `from` имеет два значения в синтаксисе, облегчающем восприятие. В начале запроса оно задает оригинальную переменную итерации и входную последовательность. *В любом другом месте* запроса оно транслируется в метод `SelectMany`. Вот так будет выглядеть наш запрос в синтаксисе, облегчающем восприятие:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  select name;
```

Обратите внимание, что дополнительный генератор определяет новую переменную запроса, в данном случае `name`. Она с этого момента становится переменной итерации, а старая переменная итера-

ции "понижается в должности" и становится *внешней переменной итерации*.

Внешние переменные итерации

В предыдущем примере переменная `fullName` превратилась во внешнюю переменную итерации после оператора `SelectMany`. Внешние переменные итерации остаются в зоне видимости до тех пор, пока запрос либо не закончится, либо не дойдет до предложения `into`. Расширенная область видимости этих переменных является решающим соображением в пользу синтаксиса, облегчающего восприятие, по сравнению с лямбда-синтаксисом.

В качестве иллюстрации мы можем взять предыдущий запрос и включить переменную `fullName` в завершающую итерацию:

```
IEnumerable<string> query =
  from fullName in fullNames     // Внешняя переменная
  from name in fullName.Split()  // Переменная итерации
  select name + " came from " + fullName;
Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

"За кулисами" компилятор должен проделать несколько трюков, чтобы разрешить внешние ссылки. Чтобы оценить его работу, попробуйте написать тот же запрос в соответствии с лямбда-синтаксисом. Это не так-то просто! Задача стано-

вится еще сложнее, если вы поставите конструкцию `where` или `orderby` перед проецированием:

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
select name + " came from " + fullName;
```

Проблема в том, что оператор `SelectMany` возвращает плоскую последовательность из элементов-потомков, в нашем случае — плоскую последовательность слов. Оригинальный внешний элемент, от которого она произошла (`fullName`), утрачена. Решение заключается в "переносе" внешнего элемента вместе с каждым потомком во временный анонимный тип:

```
from fullName in fullNames
from x in
  fullName.Split()
    .Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

Единственное изменение, которое мы здесь сделали, состояло в заключении элемента-потомка (`name`) в оболочку анонимного типа, которая содержит также и `fullName`. Примерно так же транслируется конструкция `let`. Вот окончательный вид запроса в лямбда-синтаксисе:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName =>
     fName.Split()
       .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
```

```
  .Select  (x => x.name + " came from " +
x.fName);
```

Примечание

Оператор `SelectMany` имеет перегруженную версию, выполняющую `SelectMany` и `Select` за один шаг. Воспользуемся этим, чтобы (незначительно) упростить предыдущий код, и заменим выделенный фрагмент на такой:

```
.SelectMany (
  fName => fName.Split( ),
  (fName, name) => new { name, fName }
)
```

Мышление в соответствии с синтаксисом, облегчающим восприятие

Только что мы продемонстрировали преимущество синтаксиса, облегчающего восприятие, в тех случаях, когда вам нужна внешняя переменная итерации. В подобных ситуациях полезно не просто применять этот синтаксис, но и мысли в его терминах.

Существуют две базовые модели написания дополнительных генераторов. Первая заключается в *расширении последовательностей и преобразовании их в плоскую форму.* С этой целью вы вызываете свойство или метод для существующей переменной запроса в вашем дополнительном генераторе. Так мы поступили в предыдущем примере:

```
from fullName in fullNames
from name in fullName.Split()
```

Здесь мы перешли от перебора полных имен к перебору слов. Аналогичный запрос LINQ к SQL получается, когда вы расширяете дочерние ассоциативные свойства. В следующем запросе перечисляются все клиенты и их покупки:

```
IEnumerable<string> query =
  from c in dataContext.Customers
  from p in c.Purchases
  select c.Name + " bought a " + p.Description;
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Здесь мы "расширили" каждого покупателя в подпоследовательность его покупок.

Вторая модель состоит в получении векторного произведения или перекрестного объединения, при котором каждому элементу одной последовательности ставится в пару каждый элемент другой. Для этого вы должны создать генератор, у которого селекторное выражение возвращает последовательность, не связанную с переменной итерации:

```
int[] numbers = { 1, 2, 3 };
string[] letters = { "a", "b" };
IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString()
+ l;
// Результат: { "1a", "1b", "2a", "2b", "3a",
"3b" }
```

Стиль запроса является основой для *объединений*, выполненных с помощью `SelectMany`.

Объединение с помощью оператора *SelectMany*

Вы можете воспользоваться оператором `SelectMany` для объединения двух последовательностей. Просто отфильтруйте результат векторного умножения. Предположим, например, что мы хотим расставить игроков попарно. Для начала можно написать такой запрос:

```
string[] players = { "Tom", "Jay", "Mary" };
IEnumerable<string> query =
  from name1 in players
  from name2 in players
  select name1 + " vs " + name2;
```

Результат:

```
{"Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
 "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
 "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary"}
```

Этот запрос можно перевести на человеческий язык так: "Для каждого игрока перебрать всех игроков и поставить игрока 1 против игрока 2". Хотя мы получили то, что просили (перекрестное объединение), от результатов будет мало пользы, пока мы не выполним фильтрацию:

```
IEnumerable<string> query =
  from name1 in players
  from name2 in players
  where name1.CompareTo (name2) < 0
  orderby name1, name2
  select name1 + " vs " + name2;
```

Результат:

```
{ "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

Фильтрующий предикат образует *условие объединения*. Наш запрос можно назвать *объединением не по равенству*, потому что в условии объединения отсутствует операция равенства.

Далее мы продемонстрируем остальные виды объединений в запросах LINQ к SQL.

Оператор *SelectMany* в запросах LINQ к SQL

Оператор `SelectMany` в запросах LINQ к SQL может выполнять перекрестные объединения, объединения не по равенству, внутренние объединения и левые внешние объединения. Вы можете пользоваться этим оператором как со стандартными ассоциациями, так и со специализированными отношениями, — точно так же, как пользовались оператором `Select`. Разница между двумя операторами состоит в том, что `SelectMany` возвращает плоский, а не иерархический набор результатов.

Перекрестное объединение в запросах LINQ к SQL пишется в точности так, как в предыдущем разделе. В следующем запросе каждому покупателю ставится в соответствие каждая покупка (то есть, выполняется перекрестное объединение):

```
var query =
  from c in dataContext.Customers
  from p in dataContext.Purchases
  select c.Name + " might have bought a " +
p.Description;
```

Впрочем, более осмысленным вариантом является сопоставление покупателей только с их собствен-

ными покупками. Для этого вы должны добавить предложение `where` с объединяющим предикатом. Следующий код выполняет стандартное объединение по равенству в стиле SQL:

```
var query =
  from c in dataContext.Customers
  from p in dataContext.Purchases
  where c.ID == p.CustomerID
  select c.Name + " bought a " + p.Description;
```

> ***Примечание***
>
> Этот запрос хорошо транслируется в SQL. В следующем разделе мы увидим, как его можно расширить, чтобы поддерживать внешние объединения. Переформулировка таких запросов с помощью LINQ-оператора `Join` фактически делает их *менее* расширяемыми. В этом смысле технология LINQ противоположна SQL.

Если в сущностях для ваших запросов LINQ к SQL имеются ассоциативные свойства для отношений, вы можете выразить тот же запрос, расширяя подколлекцию, а не фильтрацию векторного произведения:

```
from c in dataContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

Преимущество такого подхода в том, что он позволяет обойтись без объединяющего предиката. Мы перешли от фильтрации векторного произведения к расширению последовательности и переводу ее в плоскую форму. Однако оба запроса будут оттранслированы в один и тот же SQL-код.

В целях дополнительной фильтрации вы можете добавить в такой запрос предложения `where`. Например, если мы захотим получить только тех клиентов, имена которых начинаются на "J", мы сможем поставить такой фильтр:

```
from c in dataContext.Customers
where c.Name.StartsWith ("J")
from p in c.Purchases
select new { c.Name, p.Description };
```

Этот запрос LINQ к SQL будет работать столь же успешно, если мы передвинем предложение `where` на одну строчку вниз. Однако перенос предложения `where` в локальном запросе сделает этот запрос менее эффективным. В локальных запросах вы должны фильтровать *до* объединения.

С помощью дополнительных предложений `from` вы можете добавлять новые таблицы. Например, если у каждой записи о покупке имеются дочерние строки, вы можете получить плоский набор результатов с информацией о клиентах и их покупках, причем каждая покупка будет иметь подробное описание:

```
from c in dataContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description,
pi.DetailLine };
```

Каждое предложение `from` представляет новую *дочернюю* таблицу. Чтобы включить в результат данные о *родительской* таблице (с помощью ассоциативного свойства), вы не добавляете предложение `from`, а просто обращаетесь к свойству. Например,

если клиента обслуживал конкретный продавец ("salesperson"), то вы можете запросить имя продавца:

```
from c in dataContext.Customers
select new { Name = c.Name, SalesPerson =
c.SalesPerson.Name };
```

В этом случае мы не применяем оператор `SelectMany`, потому что нет подколлекции, которую нужно сделать плоской. Свойства, устанавливающие родительские ассоциации, возвращают один элемент.

Внешние объединения с помощью оператора *SelectMany*

Ранее мы видели, что подзапрос `Select` возвращает результат, аналогичный левому внешнему объединению.

```
from c in dataContext.Customers
select new {
             c.Name,
             Purchases =
               from p in c.Purchases
               where p.Price > 1000
               select new { p.Description,
p.Price }
           };
```

В этом примере возвращается каждый внешний элемент (клиент), независимо от того, покупал ли он что-нибудь. Перепишем этот запрос с использованием оператора `SelectMany`, чтобы получить од-

ну плоскую коллекцию, а не иерархический набор результатов:

```
from c in dataContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

По ходу преобразования запроса в "плоскую" форму мы переключились на внутреннее объединение. Теперь в результат попадают только клиенты, сделавшие одну или несколько дорогих покупок. Чтобы получить левое внешнее объединение с плоским результирующим набором, мы должны применить оператор запроса `DefaultIfEmpty` к внутренней последовательности. Этот метод возвращает `null`, если его входная последовательность не содержит элементов. Приведем такой запрос без предиката, указывающего цену:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
             c.Name,
             p.Description,
             Price = (decimal?) p.Price
           };
```

Этот код прекрасно работает с запросами LINQ к SQL и возвращает всех клиентов, даже тех, кто ничего не купил. Однако, если мы пытаемся выполнить его как локальный запрос, дело закончится неудачей, потому что когда `p` равняется `null`, свойства `p.Description` и `p.Price` возбуждают исключение `NullReferenceException`. Чтобы запрос был

устойчивым при любом сценарии, нужно переписать код так:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
              c.Name,
              Descript = p == null ? null :
p.Description,
              Price = p == null ? (decimal?) null
: p.Price
           };
```

Теперь введем ценовой фильтр. Мы не можем воспользоваться предложением `where`, как раньше, потому что оно выполнялось бы после метода `DefaultIfEmpty`:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

Правильное решение состоит в соединении метода `Where` с подзапросом *до* вызова метода `DefaultIfEmpty`:

```
from c in dataContext.Customers
from p in c.Purchases.Where (p => p.Price >
1000)
                        .DefaultIfEmpty()
select new {
              c.Name,
              Descript = p == null ? null :
p.Description,
              Price = p == null ? (decimal?) null
: p.Price
           };
```

Технология запросов LINQ к SQL транслируется в этот код в левое внешнее объединение. Это достаточно эффективный образец для написания запросов такого типа.

> ***Примечание***
>
> Если вы привыкли писать внешние объединения на языке SQL, вы, возможно, не заметите более простое решение, которое заключается в использовании подзапроса `Select`, и предпочтете громоздкий, но знакомый подход с получением плоского результата в духе SQL. Иерархический результирующий набор, возвращенный подзапросом `Select`, лучше годится для запросов, ориентированных на внешнее объединение, потому что избавляет вас от необходимости обрабатывать "лишние" значения `null`.

Объединение

Метод	Описание	Эквивалентен SQL
`Join`	Применяет стратегию просмотра для попарного объединения элементов двух коллекций и возвращает плоский результирующий набор	`INNER JOIN`
`GroupJoin`	То же, что и выше, но возвращается *иерархический* набор результатов	`INNER JOIN`, `LEFT OUTER JOIN`

Операторы *Join* и *GroupJoin*

Аргументы оператора *Join*

Аргумент	Тип
Внешняя последовательность	`IEnumerable<TOuter>`
Внутренняя последовательность	`IEnumerable<TInner>`
Внешний селектор ключа	`TOuter => TKey`
Внутренний селектор ключа	`TInner => TKey`
Селектор результата	`(TOuter,TInner) => TResult`

Аргументы оператора *GroupJoin*

Аргумент	Тип
Внешняя последовательность	`IEnumerable<TOuter>`
Внутренняя последовательность	`IEnumerable<TInner>`
Внешний селектор ключа	`TOuter => TKey`
Внутренний селектор ключа	`TInner => TKey`
Селектор результата	`(TOuter,IEnumerable<TInner>) => TResult`

Тип возвращаемого значения:
`IEnumerable<TResult>`

Синтаксис, облегчающий восприятие

```
from внешняя-переменная in внешняя-
перечисляемая-последовательность
join внутренняя-переменная in внутренняя-
перечисляемая-последовательность
  on внешнее-ключевое-выражение equals внутрен-
нее-ключевое-выражение
  [ into идентификатор ]
```

Описание

Операторы `Join` и `GroupJoin` объединяют две входных последовательности в одну выходную. Оператор `Join` возвращает плоскую последовательность, а `GroupJoin` — иерархическую.

Операторы `Join` и `GroupJoin` реализуют стратегию, альтернативную той, что обеспечивают операторы `Select` и `SelectMany`. Преимущество операторов `Join` и `GroupJoin` состоит в том, что они эффективны для локальных запросов к коллекциям, хранящимся в памяти, потому что они вначале загружают внутреннюю последовательность в таблицу просмотра, тем самым исключая необходимость в повторном переборе всех внутренних элементов. Недостаток этих операторов заключается в том, что они предлагают эквиваленты только для внутреннего и левого внешнего объединения. Что касается перекрестного объединения и объединения не по равенству, их по-прежнему приходится делать с помощью операторов `Select` и `SelectMany`. На запросах LINQ к SQL операторы `Join` и `GroupJoin` не имеют каких-то реальных преимуществ над операторами `Select` и `SelectMany`.

В следующей таблице дается сравнение различных стратегий объединения.

Стратегия	Select + SelectMany	Select + Select	Join	GroupJoin	GroupJoin + SelectMany
Форма результата	Плоская	Вложенная	Плоская	Вложенная	Плоская
Эффективность локальных запросов	Низкая	Низкая	Высокая	Высокая	Высокая
Внутренние объединения	Поддерживаются	Поддерживаются	Поддерживаются	Поддерживаются	Поддерживаются
Левые внешние объединения	Поддерживаются	Поддерживаются	—	Поддерживаются	Поддерживаются
Перекрестные объединения	Поддерживаются	Поддерживаются	—	—	—
Объединения не по равенству	Поддерживаются	Поддерживаются	—	—	—

Оператор *Join*

Оператор `Join` выполняет внутреннее объединение, возвращая плоскую выходную последовательность.

Чтобы продемонстрировать работу оператора `Join`, воспользуемся самым простым способом — напишем запрос LINQ к SQL. В следующем коде мы запрашиваем информацию обо всех клиентах и их покупках, причем обходимся без ассоциативного свойства:

```
IQueryable<string> query =
  from c in dataContext.Customers
  join p in dataContext.Purchases
    on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

Результат соответствует тому, что мы получили бы, воспользовавшись оператором `SelectMany`:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

Чтобы увидеть преимущество `Join` над `SelectMany`, мы должны преобразовать этот запрос в локальный. Вначале скопируем записи о клиентах и покупках в массивы, а затем выдадим запрос к этим массивам:

```
Customer[] customers =
dataContext.Customers.ToArray();
Purchase[] purchases =
dataContext.Purchases.ToArray();
var slowQuery =
  from c in customers
```

```
  from p in purchases where c.ID == p.CustomerID
  select c.Name + " bought a " + p.Description;
var fastQuery =
  from c in customers
  join p in purchases on c.ID equals
p.CustomerID
  select c.Name + " bought a " + p.Description;
```

Хотя оба запроса возвращают одинаковые результаты, оператор `Join` работает значительно быстрее, потому что его реализация в классе `Enumerable` предварительно загружает внутреннюю коллекцию (`purchases`) в таблицу просмотра.

В синтаксисе, облегчающем восприятие, общее правило для конструкции `join` может быть сформулировано так:

`join` *внутренняя-переменная* `in` *внутренняя-последовательность*

`on` *внешнее-ключевое-выражение* `equals` *внутреннее-ключевое-выражение*

Операторы объединения в запросах LINQ поразному обрабатывают *внешнюю последовательность* и *внутреннюю последовательность*. На уровне синтаксиса разница выглядит так:

- *Внешняя последовательность* — это входная последовательность (в данном случае, `customers`).
- *Внутренняя последовательность* — это новая коллекция, определяемая вами (в данном случае, `purchases`).

Оператор `Join` выполняет внутреннее объединение, то есть клиенты, не сделавшие покупок, не попадают в выходную последовательность. При

внутренних объединениях вы можете поменять местами внутреннюю и внешнюю последовательности в запросе и получить такие же результаты:

```
from p in purchases
join c in customers on p.CustomerID equals c.ID
...
```

Вы можете добавлять в запрос предложения `join`. Например, если каждая покупка состоит из нескольких товаров, вы можете выполнить объединение с ними:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
join pi in purchaseItems on p.ID equals
pi.PurchaseID
...
```

Последовательность `purchases` действует как *внутренняя* в первом объединении и как *внешняя* во втором. Такой же результат вы можете получить (правда, неэффективным способом) с помощью вложенных операторов `foreach`:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," +
p.Price +
                                    "," +
pi.Detail);
```

В синтаксисе, облегчающем восприятие запросов, переменные из более ранних объединений остаются в области видимости, подобно тому, как внеш-

ние переменные итерации ведут себя в запросах в стиле `SelectMany`. Кроме того, вам разрешается вставлять предложения `where` и `let` между предложениями `join`.

Объединение по нескольким ключам

Вы можете выполнить объединение по нескольким ключам с помощью анонимных типов:

```
from x in seqX
join y in seqY on new { K1 = x.Prop1, K2 =
x.Prop2 }
           equals new { K1 = y.Prop3, K2 =
y.Prop4 }
...
```

Чтобы этот код работал, оба анонимных типа должны иметь идентичную структуру. Затем компилятор реализует их одним внутренним типом, делая ключи объединения совместимыми.

Объединение в лямбда-синтаксисе

Объединение, написанное в синтаксисе, облегчающем восприятие:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

в лямбда-синтаксисе выглядит следующим образом:

```
customers.Join (          // внешняя коллекция
      purchases,          // внутренняя коллекция
```

```
      c => c.ID,          // внешний селектор
ключа
      p => p.CustomerID, // внутренний селектор
ключа
      (c, p) => new       // селектор результата
         { c.Name, p.Description, p.Price }
 );
```

Выражение-селектор результата в конце кода создает элементы выходной последовательности. Если у вас до проецирования стоят какие-либо дополнительные предложения, например, `orderby`:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

то вы должны создать временный анонимный тип в селектор результата в соответствии с лямбда-синтаксисом. Тогда и `c`, и `p` будут в области видимости в следующем объединении:

```
customers.Join (                  // внешняя кол-
лекция
      purchases,                  // внутренняя
коллекция
      c => c.ID,                  // внешний селек-
тор ключа
      p => p.CustomerID,          // внутренний се-
лектор ключа
      (c, p) => new { c, p } ) // селектор ре-
зультата
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a "
                          + x.p.Description);
```

При выполнении объединений синтаксис, облегчающий восприятие, оказывается предпочтительнее: он не требует от вас никаких трюков.

Оператор *GroupJoin*

Оператор `GroupJoin` делает то же самое, что и `Join`, но возвращает не плоский результат, а иерархическую последовательность, разбитую на группы по внешним элементам. Кроме того, этот оператор позволяет выполнять левые внешние объединения.

Синтаксис, облегчающий восприятие, для оператора `GroupJoin` такой же, как и для `Join`, но требует наличия ключевого слова `into`.

Вот наиболее типичный пример:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals
p.CustomerID
  into custPurchases
  select custPurchases; // custPurchases — это
последовательность
```

> ***Примечание***
>
> Предложение `into` транслируется в `GroupJoin`, только когда оно появляется непосредственно после предложения `join`. После предложений `select` или `group` оно означает *продолжение запроса*. Эти две интерпретации ключевого слова `into` сильно различаются, хотя и имеют одну общую черту: в обоих случаях объявляется новая переменная запроса.

Результатом является последовательность, элементы которой перебираются следующим образом:

```
foreach (IEnumerable<Purchase> purchaseSequence
in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Однако в этом мало пользы, потому что во внешней последовательности нет ссылок на внешнего покупателя. Как правило, вы будете ссылаться на внешнюю переменную итерации в проекции:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Этот код дает тот же результат, что и следующий (неэффективный) подзапрос `Select`:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases =
    purchases.Where (p => c.ID == p.CustomerID)
};
```

По умолчанию `GroupJoin` выполняет действия, эквивалентные левому внешнему объединению. Чтобы получить внутреннее объединение (где из результата исключены клиенты, которые ничего не купили), вы должны отфильтровать последовательность `custPurchases`:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
```

```
into custPurchases
where custPurchases.Any()
select ...
```

Предложения, стоящие после `into` (которое соответствует оператору `GroupJoin`), работают с *последовательностями* внутренних элементов-потомков, а не с *отдельными* потомками. Это означает, что для фильтрации отдельных покупок вы должны вызывать `Where` до объединения:

```
from c in customers
join p in purchases.Where (p2 => p2.Price >
1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

Лямбда-запросы с оператором `GroupJoin` конструируются так же, как с оператором `Join`.

Плоские внешние объединения

Когда вы хотите одновременно получить внешнее объединение и плоский результирующий набор, вы сталкиваетесь с дилеммой. Оператор `GroupJoin` дает вам внешнее объединение; оператор `Join` возвращает плоский результирующий набор. Решение заключается в том, чтобы вначале применить оператор `GroupJoin`, затем вызвать метод `DefaultIfEmpty` для каждой дочерней последовательности, а в конце — применить `SelectMany` к результату:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
```

```
into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null :
cp.Price
};
```

Метод `DefaultIfEmpty` возвращает `null`, если подпоследовательность покупок пуста. Второе предложение `from` транслируется в оператор `SelectMany`. В этой роли он расширяет и делает плоскими все подпоследовательности покупок, конкатенируя их в одну последовательность элементов, соответствующих покупкам.

Объединение и таблицы просмотра

Методы `Join` и `GroupJoin` в классе `Enumerable` выполняют свою работу в два этапа. Вначале они загружают внутреннюю последовательность в *таблицу просмотра*. Затем они опрашивают внешнюю последовательность вместе с этой таблицей.

Таблица просмотра — это последовательность групп, к которым можно обращаться непосредственно по ключу. Если взглянуть на нее под другим углом зрения, это словарь последовательностей, который может содержать несколько элементов под каждым ключом. Таблицы просмотра доступ-

ны только для чтения и определяются следующим интерфейсом:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>,
IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

> ***Примечание***
>
> Операторы объединения, как и прочие операторы, порождающие последовательности, придерживаются принципа отложенного, или "ленивого", выполнения. Это означает, что таблица просмотра не строится, пока вы не начнете перебирать элементы выходной последовательности.

Вы можете создавать и опрашивать таблицы просмотра вручную в качестве стратегии, альтернативной применению операторов объединения, когда имеете дело с локальными коллекциями. Вы можете многократно использовать одну и ту же таблицу просмотра в нескольких запросах.

Метод расширения `ToLookupTT` создает новую таблицу просмотра. В следующем коде все записи о покупках загружаются в таблицу просмотра, а в качестве ключа служит свойство `CustomerID`:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p =>
p);
```

Первый аргумент выбирает ключ; второй — объекты, которые должны быть загружены в таблицу просмотра в виде значений.

Чтение таблицы просмотра аналогично чтению словаря с тем отличием, что индексатор возвращает *последовательность* соответствующих элементов, а не *единственный* элемент. В следующем коде перебираются все покупки, сделанные клиентом, у которого идентификатор ID равен 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

Когда таблица просмотра создана, вы можете писать операторы `SelectMany` или `Select`, которые будут выполняться столь же эффективно, что и операторы `Join` или `GroupJoin`. Оператор `Join` будет эквивалентен оператору `SelectMany`, примененному к таблице просмотра:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };

Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

Добавление вызова `DefaultIfEmpty` заставит этот код выполнить внешнее объединение:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
select new
```

```
{
  c.Name,
  Descript = p == null ? null : p.Description,
  Price = p == null ? (decimal?) null : p.Price
};
```

Оператор `GroupJoin` эквивалентен чтению таблицы просмотра внутри проекции:

```
from c in customers
select new {
              CustName = c.Name,
              CustPurchases = purchLookup [c.ID]
            };
```

Упорядочивание

Метод	Описание	Эквивалентен SQL
`OrderBy`, `ThenBy`	Сортирует последовательность в порядке возрастания элементов	`ORDER BY ...`
`OrderBy Descending`, `ThenBy Descending`	Сортирует последовательность в порядке убывания элементов	`ORDER BY ... DESC`
`Reverse`	Возвращает последовательность с обратным порядком элементов	Возбуждается исключение

Операторы упорядочивания возвращают те же самые элементы в другом порядке.

Операторы *OrderBy*, *OrderByDescending*, *ThenBy* и *ThenByDescending*

Аргументы операторов *OrderBy* и *OrderByDescending*

Аргумент	Тип
Входная последовательность	`IEnumerable<TSource>`
Селектор ключа	`TSource => TKey`

Тип возвращаемого значения:
`IOrderedEnumerable<TSource>`.

Аргументы операторов *ThenBy* и *ThenByDescending*

Аргумент	Тип
Входная последовательность	`IOrderedEnumerable<TSource>`
Селектор ключа	`TSource => TKey`

Синтаксис, облегчающий восприятие

```
orderby выражение1 [descending]
  [, выражение2 [descending] ... ]
```

Описание

Оператор `OrderBy` возвращает отсортированную версию входной последовательности, пользуясь выражением `keySelector` для выполнения сравнений. В следующем запросе порождается последовательность, в которой имена идут в алфавитном порядке:

```
IEnumerable<string> query = names.OrderBy (s =>
s);
```

А этот код упорядочивает имена по длине:

```
IEnumerable<string> query =
  names.OrderBy (s => s.Length);
// Результат: { "Jay", "Tom", "Mary", "Dick",
"Harry" };
```

Относительный порядок элементов с одинаковым ключом сортировки (например, Jay/Tom или Mary/Dick) неопределен, если вы не применяете оператор `ThenBy`:

```
IEnumerable<string> query = names.OrderBy (s =>
s.Length)
                                  .ThenBy (s =>
s);
// Результат: { "Jay", "Tom", "Dick", "Mary",
"Harry" };
```

Оператор `ThenBy` переупорядочивает только элементы, у которых в предыдущей сортировке были одинаковые ключи. Вы можете выстроить цепочку из любого количества операторов `ThenBy`. В следующем коде элементы сортируются вначале по длине, затем — по второму символу и, наконец, по первому:

```
names.OrderBy (s => s.Length)
     .ThenBy (s => s[1]).ThenBy (s => s[0]);
```

В синтаксисе, облегчающем восприятие, эквивалентный код выглядит так:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

В технологии запросов LINQ есть операторы `OrderByDescending` и `ThenByDescending`, которые делают то же самое, но выдают элементы в обратном порядке. В результате следующего запроса LINQ к SQL покупки сортируются в порядке убывания цены, а при равенстве цен они сортируются по алфавиту:

```
dataContext.Purchases.OrderByDescending (p =>
p.Price)
                     .ThenBy (p =>
p.Description);
```

В синтаксисе, облегчающем восприятие, это выглядит так:

```
from p in dataContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Классы, выполняющие сравнение, и сортировка

В локальном запросе объекты-селекторы ключей сами определяют алгоритм выяснения порядка, исходя из их реализации интерфейса `IComparable`. Вы можете переопределить алгоритм сортировки, принятый по умолчанию, указав в качестве аргумента объект `IComparer`. В сле-

дующем коде выполняется сортировка, нечувствительная к регистру:

```
names.OrderBy (n => n,

StringComparer.CurrentCultureIgnoreCase);
```

Передача класса, выполняющего сравнение, в качестве аргумента оператора запроса не поддерживается ни в синтаксисе, облегчающем восприятие, ни в технологии запросов LINQ к SQL. В последней алгоритм сравнения определяется тем, как был упорядочен соответствующий столбец. Если сортировка была чувствительна к регистру, вы можете запросить сортировку без учета регистра, вызвав метод `ToUpper` в селекторе ключа:

```
from p in dataContext.Purchases
orderby p.Description.ToUpper()
select p;
```

Интерфейсы *IOrderedEnumerable* и *IOrderedQueryable*

Операторы упорядочивания возвращают объекты специальных подтипов обобщенного типа `IEnumerable<T>`. Методы класса `Enumerable` возвращают объекты типа, реализующего интерфейс `IOrderedEnumerable`, а методы класса `Queryable` — объекты типа, реализующего `IOrderedQueryable`. Эти подтипы позволяют последующему оператору `ThenBy` уточнять имеющийся порядок объектов, а не заменять его на новый.

Дополнительные члены этих подтипов не являются открытыми, и такие последовательности внешне не

отличаются от обычных. Тот факт, что они имеют разные типы, начинает играть роль только при последовательном построении запросов:

```
IOrderedEnumerable<string> query1 =
  names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 =
  query1.ThenBy (s => s);
```

Если вы укажете для переменной `query1` тип `IEnumerable<string>`, вторая строчка не откомпилируется, потому что оператору `ThenBy` на входе нужна последовательность типа `IOrderedEnumerable<string>`. Вы можете избавить себя от беспокойства по этому поводу с помощью неявного приведения типов переменных запросов:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Впрочем, неявное приведение типов может само по себе создать проблемы. Следующий код не откомпилируется:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);
// Ошибка
```

Компилятор автоматически определяет, что `query` имеет тип `IOrderedEnumerable<string>`, исходя из типа выходной последовательности оператора `OrderBy`. Однако оператор `Where` в следующей строчке возвращает обычную последовательность `IEnumerable<string>`, которую нельзя присвоить переменной `query`. Обойти эту проблему можно либо путем явного преобразования типа, либо с

помощью метода `AsEnumerable()`, вызванного после оператора `OrderBy`:

```
var query = names.OrderBy (s =>
s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);
// OK
```

Эквивалентным решением для интерпретируемых запросов является вызов метода `AsQueryable`.

Группирование

Метод	Описание	Эквивалентен SQL
`GroupBy`	Разбивает последовательность на подпоследовательности	`GROUP BY`

Оператор *GroupBy*

Аргумент	Тип
Входная последовательность	`IEnumerable<TSource>`
Селектор ключа	`TSource => TKey`
Селектор элемента (необязательный)	`TSource => TElement`
Класс, выполняющий сравнение (необязательный)	`IEqualityComparer<TKey>`

Тип возвращаемого значения:
`IEnumerable<IGrouping<TSource,TElement>>`

Синтаксис, облегчающий восприятие

`group` *выражение_для_элемента* `by` *выражение_для_ключа*

Описание

Оператор `GroupBy` реорганизует плоскую входную последовательность в последовательность *групп*. Например, в следующем коде файлы в каталоге c:\temp группируются по расширению:

```
string[] files = Directory.GetFiles
("c:\\temp");
IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension
(file));
```

Или, если вам больше нравится неявное преобразование типов:

```
var query = files.GroupBy
  (file => Path.GetExtension (file));
```

Вот как нужно перебирать элементы результата:

```
foreach (IGrouping<string,string> grouping in
query)
{
  Console.WriteLine ("Extension: " +
grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine ("   - " + filename);
}
```

```
Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Метод `Enumerable.GroupBy` читает входные элементы во временный словарь списков, чтобы все элементы с одинаковыми ключами, в конце концов, оказались в одном подсписке. Затем метод создает последовательность *групп*. Одна группа — это последовательность со свойством `Key`:

```
public interface IGrouping <TKey,TElement>
  : IEnumerable<TElement>, IEnumerable
{
  // Свойство Key применяется к подпоследова-
тельности в целом
  TKey Key { get; }
}
```

По умолчанию элементами каждой группы являются непреобразованные входные элементы, если вы не укажете в качестве аргумента селектор элемента. В следующем коде каждый входной элемент переводится в верхний регистр:

```
files.GroupBy (file =>
  Path.GetExtension (file), file =>
file.ToUpper();
```

Селектор элемента не зависит от селектора ключа.

В нашем случае это означает, что свойство `Key` каждой группы остается в прежнем регистре:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Обратите внимание, что подколлекции генерируются отнюдь не в порядке следования ключей по алфавиту. Оператор `GroupBy` занимается только группировкой; он не выполняет сортировку. То есть, он сохраняет исходный порядок следования элементов. Чтобы отсортировать результат, вы должны добавить оператор `OrderBy`:

```
files
  .GroupBy (file =>
     Path.GetExtension (file), file =>
file.ToUpper())
  .OrderBy (grouping => grouping.Key);
```

Оператор `GroupBy` легко и прямолинейно транслируется в синтаксис, облегчающий восприятие:

`group` *выражение-для-элемента* `by` *выражение-для-ключа*

Наш код в соответствии с этим синтаксисом будет выглядеть так:

```
from file in files
group file.ToUpper() by Path.GetExtension
(file);
```

Подобно конструкции `select` конструкция `group` "заканчивает" запрос, если, конечно, вы не добавляете конструкцию, продолжающую его:

```
from file in files
```

```
group file.ToUpper() by Path.GetExtension (file)
into grouping
orderby grouping.Key
select grouping;
```

Продолжение запроса часто оказывается уместным при группировании. В следующем запросе сквозь фильтр проходят только группы, содержащие менее пяти файлов:

```
from file in files
group file.ToUpper() by Path.GetExtension (file)
into grouping
where grouping.Count() < 5
select grouping;
```

> ***Примечание***
>
> Конструкция `where` после `group by` эквивалентна SQL-оператору `HAVING`. Она применяется к каждой подпоследовательности или группе в целом, а не к отдельным элементам.

Иногда вас интересует результат агрегирования в группе, и тогда вы можете отказаться от подпоследовательности:

```
string[] votes = {
"Bush","Gore","Gore","Bush","Bush" };
IEnumerable<string> query = from vote in votes
                            group vote by vote
into g
                            orderby g.Count()
descending
                            select g.Key;
string winner = query.First();    // Bush
```

Оператор *GroupBy* в запросах LINQ к SQL

В интерпретируемых запросах группирование происходит точно так же. Однако, если в вашем запросе LINQ к SQL установлены ассоциативные свойства, вы заметите, что необходимость в группировании возникает реже, чем при стандартных SQL-запросах. Например, чтобы выделить клиентов, сделавших хотя бы две покупки, вам не нужно прибегать к помощи конструкции `group`. Следующий запрос прекрасно справляется с этой задачей:

```
from c in dataContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count
                             + " purchases";
```

Ситуация, в которой вам, скорее всего, понадобится группирование, — это составление списков всех продаж по группам:

```
from p in dataContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

LINQ-операторы группирования образуют надмножество функциональных возможностей SQL-оператора `GROUP BY`. Еще одним отступлением от традиционного SQL является отсутствие обязательного проецирования переменных или выражений, использованных в группировании или сортировке.

Группирование по нескольким ключам

Вы можете выполнять группирование по составному ключу с использованием анонимного типа:

```
from n in names
group n by new { FirstLetter = n[0], Length =
n.Length };
```

Пользовательские классы для выяснения равенства

В локальном запросе вы можете передать оператору `GroupBy` пользовательский класс для выяснения равенства, изменив тем самым алгоритм сравнения ключей. Впрочем, необходимость в этом возникает редко, потому что смены селектора ключа, как правило, бывает достаточно. Например, в следующем запросе происходит группирование без учета регистра:

```
group name by name.ToUpper()
```

Операции над множествами

Метод	Описание	Эквивалентен SQL
`Concat`	Возвращает конкатенацию отдельных элементов обеих последовательностей	`UNION ALL`

(окончание)

Метод	Описание	Эквивалентен SQL
`Union`	Возвращает конкатенацию отдельных элементов обеих последовательностей, но без повторений	`UNION`
`Intersect`	Возвращает элементы, присутствующие в обеих последовательностях	`WHERE … IN (…)`
`Except`	Возвращает элементы, присутствующие в первой, но отсутствующие во второй последовательности	`EXCEPT` или `WHERE … NOT IN (…)`

Операторы *Concat* и *Union*

Оператор `Concat` возвращает все элементы первой последовательности, за которыми идут все элементы второй. Оператор `Union` делает то же самое, но убирает повторения:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
  concat = seq1.Concat (seq2),    //  { 1, 2, 3,
3, 4, 5 }
  union  = seq1.Union  (seq2);    //  { 1, 2, 3,
4, 5 }
```

Операторы *Intersect* и *Except*

Оператор `Intersect` возвращает элементы, общие для двух последовательностей. Оператор `Except` возвращает элементы первой последовательности, которых *нет* во второй:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
  commonality = seq1.Intersect (seq2),    //  { 3 }
  difference1 = seq1.Except    (seq2),    //  { 1, 2 }
  difference2 = seq2.Except    (seq1);    //  { 4, 5 }
```

Метод `Enumerable.Except` устроен следующим образом: он загружает все элементы первой последовательности в словарь, затем удаляет из словаря элементы, присутствующие во второй последовательности. SQL-эквивалентом является подзапрос `NOT EXISTS` или `NOT IN`:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

Методы преобразования

Технология LINQ работает, в основном, с последовательностями, то есть коллекциями типа `IEnumerable<T>`.

Методы преобразования изменяют типы коллекций:

Метод	Описание
`OfType`	Преобразует `IEnumerable` в `IEnumerable<T>`, отбрасывая элементы, у которых преобразование типа закончилось неуспешно
`Cast`	Преобразует `IEnumerable` в `IEnumerable<T>`, возбуждая исключение при наличии элементов, у которых преобразование типа закончилось неуспешно
`ToArray`	Преобразует `IEnumerable<T>` в `T[]`
`ToList`	Преобразует `IEnumerable<T>` в `List<T>`
`ToDictionary`	Преобразует `IEnumerable<T>` в `Dictionary<TKey,TValue>`
`ToLookup`	Преобразует `IEnumerable<T>` в `ILookup<TKey,TElement>`
`AsEnumerable`	Преобразует к типу `IEnumerable<T>`
`AsQueryable`	Преобразует к типу `IQueryable<T>`

Операторы *OfType* и *Cast*

Операторы `OfType` и `Cast` принимают коллекцию необобщенного типа `IEnumerable` и генерируют коллекцию обобщенного типа `IEnumerable<T>`, к которой вы впоследствии сможете применить запрос:

```
// Тип ArrayList определен в пространстве имен
System.Collections
```

```
ArrayList classicList = new ArrayList();
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 =
classicList.Cast<int>();
```

Эти операторы ведут себя по-разному, когда встречают входной элемент несовместимого типа. Оператор `Cast` возбуждает исключение, а `OfType` игнорирует такой элемент. Продолжим предыдущий пример:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int> sequence2 = classicList
  .OfType<int>(), // OK - игнорирует DateTime
IEnumerable<int> sequence3 = classicList
  .Cast<int>();   // Возбуждает исключение
```

Правила совместимости типов элементов в точности совпадают с теми, по которым работает операция `is` языка C#. В этом можно убедиться, изучив внутреннюю реализацию метода `OfType`:

```
public static IEnumerable<TSource> OfType
<TSource>
  (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

У метода `Cast` практически такая же реализация, если не считать отсутствие проверки на совместимость типов:

```
public static IEnumerable<TSource> Cast
<TSource>
```

```
  (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

Последствием таких реализаций является невозможность использовать оператор `Cast` для преобразования элементов из одного типа-значения в другой (для этого вам придется выполнять оператор `Select`). Иными словами, оператор `Cast` не обладает гибкостью операции приведения типов в C#, которая допускает статические преобразования типов, например:

```
int i = 3;
long l = i;          // Статическое преобразование int->long
int i2 = (int) l;    // Статическое преобразование long->int
```

Мы можем проиллюстрировать сказанное, попытавшись воспользоваться операторами `OfType` или `Cast` для преобразования последовательности чисел `int` в последовательность чисел `long`:

```
int[] integers = { 1, 2, 3 };
IEnumerable<long> test1 =
integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

Когда мы будем перебирать элементы последовательности, запрос `test1` не возвратит ни одного элемента, а `test2` возбудит исключение. Изучение реализации `OfType` дает ответ на вопрос, почему это так. После подстановки `TSource` мы получаем следующее выражение:

```
(element is long)
```

Оно возвращает `false` для `int element` из-за отсутствия отношения наследования.

Как мы уже говорили ранее, решение состоит в использовании обычного оператора `Select`:

```
IEnumerable<long> castLong =
  integers.Select (s => (long) s);
```

Операторы `OfType` и `Cast` полезны также при преобразовании типа элементов обобщенной входной последовательности. Например, если у вас есть входная последовательность типа `IEnumerable<Fruit>` (fruit — фрукты, прим. перев.), то оператор `OfType<Apple>` (apple — яблоко, прим. перев.) будет возвращать только яблоки. Это очень удобно в запросах LINQ к XML.

Операторы *ToArray*, *ToList*, *ToDictionary* и *ToLookup*

Операторы `ToArray` и `ToList` возвращают результаты в виде массива или обобщенного списка. Они вызывают форсированный перебор элементов входной последовательности (если подзапрос или дерево выражений не создали дополнительный уровень косвенности). Примеры приводились ранее в *разд. "Отложенное выполнение"*.

Операторы `ToDictionary` и `ToLookup` принимают следующие аргументы:

Аргумент	Тип
Входная последовательность	`IEnumerable<TSource>`

(окончание)

Аргумент	Тип
Селектор ключа	`TSource => TKey`
Селектор элемента (необязательный)	`TSource => TElement`
Класс, выполняющий сравнение (необязательный)	`IEqualityComparer<TKey>`

Оператор `ToDictionary` тоже форсирует немедленное выполнение и записывает результаты в обобщенный словарь. Выражение-селектор ключа должно возвращать уникальное значение для каждого элемента входной последовательности; в противном случае будет возбуждено исключение. Зато оператор `ToLookup` позволяет иметь много элементов для одного ключа. Мы описали таблицы просмотра ранее в *разд. "Объединение и таблицы просмотра"*.

Операторы *AsEnumerable* и *AsQueryable*

Оператор `AsEnumerable` преобразует тип последовательности "вверх", к типу `IEnumerable<T>`, заставляя компилятор связывать будущие операторы запроса с методами класса `Enumerable`, а не класса `Queryable`. Примеры см. в *разд. "Интерпретируемые запросы"*.

Оператор `AsQueryable` преобразует тип последовательности "вниз" к типу `IQueryable<T>`, если она реализует этот интерфейс. В противном случае он создает оболочку типа `IQueryable<T>` вокруг локального запроса.

Поэлементные операции

Метод	Описание	Эквивалентен SQL
`First`, `FirstOrDefault`	Возвращает первый элемент последовательности, возможно, удовлетворяющий предикату	`SELECT TOP 1 ... ORDER BY ...`
`Last`, `LastOrDefault`	Возвращает последний элемент последовательности, возможно, удовлетворяющий предикату	`SELECT TOP 1 ... ORDER BY ... DESC`
`Single`, `SingleOrDefault`	Эквивалентны методам `First/FirstOrDefault`, но возбуждают исключение при наличии более одного подходящего элемента	

(окончание)

Метод	Описание	Эквивалентен SQL
`ElementAt`, `ElementAtOrDefault`	Возвращает элемент, стоящий в указанной позиции	Возбуждается исключение
`DefaultIfEmpty`	Возвращает `null` или `default(TSource)`, если в последовательности нет элементов	`OUTER JOIN`

Методы, имена которых заканчиваются на "OrDefault", возвращают `default(TSource)` (а не возбуждают исключение), если входная последовательность пуста или ни один ее элемент не удовлетворяет заданному предикату.

Конструкция `default(TSource)` равняется значению `null` для ссылочных типов или значению, смысл которого "пусто" (как правило, ноль), для типов-значений.

Операторы *First*, *Last* и *Single*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат (необязательный)	`TSource => bool`

В следующем примере демонстрируется работа операторов `First` и `Last`:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int first      = numbers.First();
// 1
int last       = numbers.Last();
// 5
int firstEven  = numbers.First  (n => n % 2 ==
0); // 2
int lastEven   = numbers.Last   (n => n % 2 ==
0); // 4
```

А здесь сравниваются операторы `First` и `FirstOrDefault`:

```
// Возбуждается исключение
int firstBigError  = numbers.First     (n => n >
10);
// Результатом будет 0:
int firstBigNumber = numbers.FirstOrDefault (n
=> n > 10);
```

Чтобы не было исключения, методу `Single` требуется ровно один подходящий элемент, а методу `SingleOrDefault` нужно *один или ноль* подходящих элементов:

```
int onlyDivBy3 =
  numbers.Single (n => n % 3 == 0);   // 3
int divBy2Err  =
  numbers.Single (n => n % 2 == 0);   // Ошибка:
2 подходит
int singleError =
  numbers.Single (n => n > 10);       // Ошибка:
нет соответствий
int noMatches   =
   numbers.SingleOrDefault (n => n > 10);
// 0
```

```
int divBy2Error =
  numbers.SingleOrDefault (n => n % 2 == 0);  // Ошибка
```

Оператор `Single` является самым требовательным в этом семействе, а операторы `FirstOrDefault` и `LastOrDefault` — самыми толерантными.

В запросах LINQ к SQL оператор `Single` часто используется для получения строки таблицы по первичному ключу:

```
Customer cust =
  dataContext.Customers.Single (c => c.ID == 3);
```

Оператор *ElementAt*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Индекс возвращаемого элемента	`int`

Оператор `ElementAt` возвращает n-ный элемент последовательности:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);
// 3
int tenthError = numbers.ElementAt (9);
// Ошибка
int tenth      = numbers.ElementAtOrDefault (9);
// 0
```

Метод `Enumerable.ElementAt` написан так, что если входная последовательность реализует интер-

фейс IList<T>, он вызывает индексатор IList<T>. В противном случае перебираются *n* элементов последовательности, и возвращается следующий. Оператор ElementAt не поддерживается технологией LINQ к SQL.

Оператор *DefaultIfEmpty*

Оператор DefaultIfEmpty преобразует пустые последовательности в null или default(). Он применяется при написании плоских внешних объединений; см. *разд. "Внешние объединения с помощью оператора SelectMany"* и *"Плоские внешние объединения"*.

Методы агрегирования

Метод	Описание	Эквивалентен SQL
Count, LongCount	Возвращают количество элементов во входной последовательности, возможно, удовлетворяющих заданному предикату	COUNT()
Min, Max	Возвращают наименьший и, соответственно, наибольший элемент последовательности	MIN(), MAX()
Sum, Average	Вычисляют сумму и среднее арифметическое элементов последовательности	SUM(), AVG()

(окончание)

Метод	Описание	Эквивалентен SQL
`Aggregate`	Выполняет агрегирование, определяемое пользователем	Возбуждается исключение

Операторы *Count* и *LongCount*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат (необязательный)	`TSource => bool`

Оператор `Count` просто перебирает элементы последовательности, возвращая их количество:

```
int fullCount = new int[] { 5, 6, 7 }.Count();
// 3
```

Метод `Enumerable.Count` проверяет, реализует ли входная последовательность интерфейс `ICollection<T>`. Если реализует, он просто вызывает метод `ICollection<T>.Count`. В противном случае он перебирает все элементы, увеличивая счетчик.

При необходимости вы можете указать предикат:

```
int digitCount =
  "pa55w0rd".Count (c => char.IsDigit (c)); // 3
```

Оператор `LongCount` делает то же самое, что и `Count`, но возвращает 64-битовое целое, что позволяет иметь дело с последовательностями из более чем двух миллиардов элементов.

Операторы *Min* и *Max*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата (необязательный)	`TSource => TResult`

Операторы `Min` и `Max` возвращают наименьший и наибольший элементы последовательности:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min();  // 14;
int largest  = numbers.Max();  // 32;
```

Если вы укажете выражение-селектор, каждый элемент вначале будет подвергнут проекции:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

Выражение-селектор обязательно, если элементы сами по себе не подлежат сравнению, то есть не реализуют интерфейс `IComparable<T>`:

```
Purchase runtimeError =
  dataContext.Purchases.Min ();                 //
Ошибка выполнения
decimal? lowestPrice =
  dataContext.Purchases.Min (p => p.Price);  //
OK
```

Выражение-селектор определяет не только то, как сравниваются элементы, но и конечный результат. В предыдущем примере результат представляет собой десятичное значение, а не объект "покупка". Чтобы получить "самую дешевую покупку", вы должны сделать подзапрос:

```
Purchase cheapest = dataContext.Purchases
  .Where (p => p.Price ==
     dataContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

В этом случае вы могли бы сформулировать запрос и без агрегирования, а с помощью операторов `OrderBy` и `FirstOrDefault`.

Операторы *Sum* и *Average*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата (необязательный)	`TSource => TResult`

Операторы `Sum` и `Average` являются операторами агрегирования и используются аналогично операторам `Min` и `Max`:

```
decimal[] numbers  = { 3, 4, 8 };
decimal sumTotal   = numbers.Sum();       // 15
decimal average    = numbers.Average();   // 5
(среднее арифметическое)
```

Следующий код возвращает суммарную длину всех строк массива `names`:

```
int combinedLength = names.Sum (s => s.Length);
// 19
```

Операторы `Sum` и `Average` строги по отношению к типам. Их определения жестко закодированы в каждом числовом типе (`int`, `long`, `float`, `double`, `decimal` и их версии, допускающие значение `null`). В отличие от них операторы `Min` и `Max` могут напрямую работать с чем угодно, если оно реализует интерфейс `IComparable<T>`, например, со строками.

Более того, `Average` всегда возвращает значения `decimal` или `double` в соответствии с такой таблицей:

Тип селектора	Тип результата
`decimal`	`decimal`
`int`, `long`, `float`, `double`	`double`

Это означает, что следующий код не откомпилируется ("невозможно преобразовать `double` в `int`"):

```
int avg = new int[] { 3, 4 }.Average();
```

А этот код успешно пройдет компиляцию:

```
double avg = new int[] { 3, 4 }.Average();   // 3.5
```

Оператор `Average` неявно преобразует входные значения, чтобы избежать потери точности. Мы вычисляли среднее арифметическое двух целых чисел и получили 3.5, не прибегая к приведению типа входных элементов:

```
double avg = numbers.Average (n => (double) n);
```

В запросах LINQ к SQL операторы `Sum` и `Average` транслируются в стандартные SQL-операторы. Следующий запрос возвращает информацию о клиентах, у которых средняя стоимость покупок превышает 500 долларов:

```
from c in dataContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

Оператор *Aggregate*

Оператор `Aggregate` позволяет вам подключать свой алгоритм аккумулирования для реализации нестандартного агрегирования. Он не поддерживается в запросах LINQ к SQL и является довольно специализированным. В следующем коде показано, как `Aggregate` может выполнять работу оператора `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (seed, n) =>
seed + n);
```

Первый аргумент оператора `Aggregate` — это *начальное значение*, с которого начинается аккумулирование. Второй — выражение, обновляющее аккумулируемое значение при поступлении нового элемента. При желании вы можете указать третий аргумент, который будет проецировать аккумулированное значение в окончательный результат.

Сложность работы с оператором `Aggregate` состоит в том, что простой скалярный тип редко подходит на роль аккумулятора. Например, чтобы вы должны следить не только за суммой, но и за те-

кущим количеством элементов. Написание пользовательского типа-аккумулятора решает проблему, но требует больших усилий по сравнению с обычным подходом, который сводится к использованию цикла `foreach` для выполнения агрегирования.

Квантификаторы

Метод	Описание	Эквивалентен SQL
`Contains`	Возвращает `true`, если входная последовательность содержит данный элемент	`WHERE ... IN (...)`
`Any`	Возвращает `true`, если какие-либо элементы удовлетворяют указанному предикату	`WHERE ... IN (...)`
`All`	Возвращает `true`, если все элементы удовлетворяют указанному предикату	`WHERE (...)`
`SequenceEqual`	Возвращает `true`, если элементы второй последовательности идентичны элементам первой	

Операторы *Contains* и *Any*

Оператор `Contains` принимает аргумент типа `TSource`; а `Any` принимает необязательный аргумент-предикат.

Оператор `Contains` возвращает `true`, если указанный элемент присутствует в последовательности:

```
bool isTrue = new int[] { 2, 3, 4 }.Contains
(3);
```

Оператор `Any` возвращает `true`, если указанное выражение истинно хотя бы для одного элемента. Мы можем переписать предыдущий запрос, используя оператор `Any`:

```
bool isTrue = new int[] { 2, 3, 4 }.Any (n => n
== 3);
```

Оператор `Any` может делать все, что делает `Contains`, и даже больше:

```
bool isFalse = new int[] { 2, 3, 4 }.Any (n => n
> 10);
```

Вызов оператора `Any` без предиката возвращает `true`, если последовательность содержит один или несколько элементов. Вот еще один вариант предыдущего запроса:

```
bool isFalse = new int[] { 2, 3, 4 }
               .Where (n => n > 10).Any();
```

Особенно полезен оператор `Any` в подзапросах.

Операторы *All* и *SequenceEqual*

Оператор `All` возвращает `true`, если все элементы последовательности удовлетворяют предикату. Следующий запрос возвращает информацию о покупателях, совершивших покупки меньше, чем на 100 долларов:

```
dataContext.Customers.Where
  (c => c.Purchases.All (p => p.Price < 100));
```

Оператор `SequenceEqual` сравнивает две последовательности. Он возвращает `true`, если последовательности содержат одни и те же элементы, расположенные в одинаковом порядке.

Методы генерирования коллекций

Метод	Описание
`Empty`	Создает пустую последовательность
`Repeat`	Создает последовательность из одинаковых элементов
`Range`	Создает последовательность целых чисел

`Empty`, `Repeat` и `Range` являются статическими методами (не методами расширения), которые генерируют простые локальные последовательности.

Метод *Empty*

Метод `Empty` создает пустую последовательность, и ему требуется единственный аргумент, указывающий тип:

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);                    // <ничего>
```

В комбинации с оператором `??` метод `Empty` выполняет действия, обратные методу `DefaultIfEmpty`.

Предположим, например, что у нас есть ступенчатый массив целых, и мы хотим собрать все числа в один плоский список. Следующий запрос `SelectMany` закончится неудачей, если хотя бы один из внутренних массивов равняется `null`:

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null   // этот null-массив приводит к неуспешному
         // выполнению следующего запроса
};
IEnumerable<int> flat =
  numbers.SelectMany (innerArray => innerArray);
```

Метод `Empty` и оператор `??` совместно решают эту проблему:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray =>
                innerArray ?? Enumerable.Empty
<int>());
foreach (int i in flat)
  Console.Write (i + " ");     // 1 2 3 4 5 6
```

Методы *Range* и *Repeat*

Методы `Range` и `Repeat` работают только с целыми числами. Метод `Range` принимает начальный индекс и количество элементов:

```
foreach (int i in Enumerable.Range (5, 5))
  Console.Write (i + " ");          // 5 6 7 8 9
```

Метод `Repeat` принимает количество повторений и количество шагов:

```
foreach (int i in Enumerable.Repeat (5, 3))
  Console.Write (i + " ");        // 5 5 5
```

Запросы LINQ к XML

Платформа .NET Framework предоставляет ряд API-интерфейсов для работы с XML-данными. Начиная с Framework 3.5, основной технологией неспециализированной обработки XML-документов являются запросы *LINQ к XML*. В эту технологию входят объектная модель XML-документов, удобная для формулирования LINQ-запросов, и набор дополнительных операторов запросов. В большинстве ситуаций этого достаточно для полного отказа от предшествующей объектной модели, удовлетворяющей требованиям W3C и носящей имя `XmlDocument`.

> ***Примечание***
>
> Объектная модель документов, принятая в технологии "LINQ к XML", тщательно продумана и имеет исключительно высокую производительность. Даже в отрыве от LINQ эта объектная модель может служить "облегченным" фасадом для низкоуровневых классов `XmlReader` и `XmlWriter`.

Все типы, необходимые для запросов LINQ к XML, определены в пространстве имен `System.Xml.Linq`.

Обзор архитектуры

Рассмотрим такой XML-файл:

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

Как и все XML-файлы, он начинается с *объявления*, после которого идет *корневой элемент* по имени `customer` (клиент — прим. перев.). Этот элемент имеет два *атрибута*, причем для каждого указано имя (`id` и `status`) и значение (`"123"` и `"archived"`). Внутри элемента `customer` присутствуют два элемента-потомка, `firstname` (имя — прим. перев.) и `lastname` (фамилия — прим. перев.), каждый из которых имеет простое текстовое содержимое (`"Joe"` и `"Bloggs"`).

Все эти конструкции — объявление, элемент, атрибут, значение и текстовое содержимое — могут быть представлены классами. А если у таких классов будут свойства коллекций, позволяющие хранить содержимое узла-потомка, мы сможем построить *дерево* объектов и полностью описать документ. Это и называется *объектной моделью документа*, или DOM.

Технология запросов LINQ к XML состоит из двух частей:

- из объектной модели XML-документов, которую мы будем называть *X-DOM*;

- из комплекта, включающего приблизительно 10 операторов запросов.

Как и следовало ожидать, X-DOM состоит из типов, таких как `XDocument`, `XElement` и `XAttribute`. Интересно, что типы модели X-DOM не привязаны к технологии LINQ; вы можете загружать X-DOM, создавать экземпляры классов, редактировать и сохранять их, не составляя никаких запросов LINQ к XML.

И наоборот, вы можете использовать LINQ-запросы для обращения к модели DOM, созданной на основе старых типов, удовлетворяющих требованиям W3C. Впрочем, это будет трудоемкий процесс, и вы будете постоянно сталкиваться с определенными ограничениями. Отличительной чертой модели X-DOM является ее *дружественность к технологии LINQ*. Это означает, что

- в ней есть методы, порождающие последовательности типа `IEnumerable`, к которым вы можете обращаться с запросами;
- ее конструкторы разработаны так, что вы можете строить дерево X-DOM с помощью LINQ-проекции.

Обзор модели X-DOM

На рис. 11 показаны стандартные типы модели X-DOM. Чаще других используется тип `XElement`. Тип `XObject` является корневым в иерархии *наследования*, а типы `XElement` и `XDocument` — корни в иерархии *членства*.

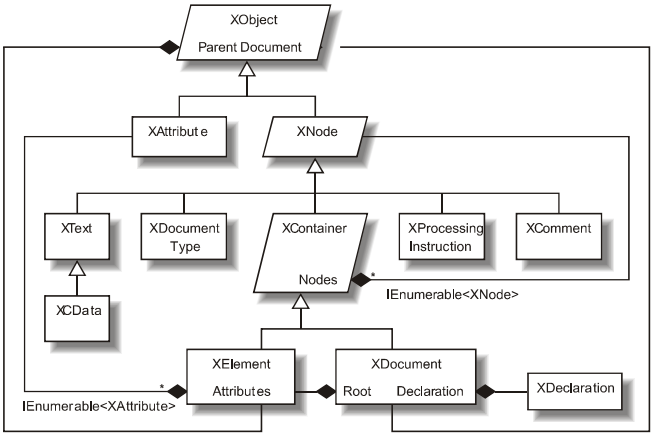

Рис. 11. Стандартные X-DOM

На рис. 12 представлено дерево X-DOM, созданное на основе следующего кода:

```
string xml =
  @"<customer id='123' status='archived'>
    <firstname>Joe</firstname>
    <lastname>Bloggs<!--nice name--></lastname>
</customer>";
XElement customer = XElement.Parse (xml);
```

`XObject` является абстрактным базовым классом для всего XML-содержимого. Он определяет ссылку на родительский элемент в иерархии членства, а также на необязательный элемент `XDocument`.

`XNode` — базовый класс для большей части XML-содержимого, кроме атрибутов. Отличительной чертой этого класса является его способность находиться в упорядоченной коллекции среди других

объектов XNode, имеющих различные типы. В качестве примера рассмотрим следующий XML-код:

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

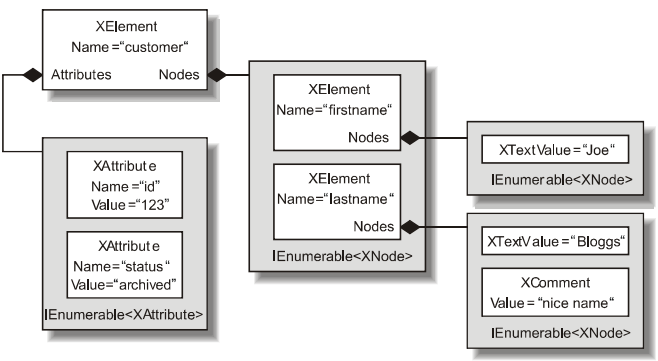

Рис. 12. Простое дерево X-DOM

В родительском элементе <data> присутствуют узел типа XText (с содержимым "Hello world"), узел типа XElement, затем узел типа XComment, после которого стоит второй узел XElement. Что касается типа XAttribute, он "терпит" другие объекты этого типа только в качестве одноранговых узлов.

Хотя объект XNode имеет доступ к своему родителю XElement, он не поддерживает концепцию узлов-потомков; эту работу он перекладывает на свой подкласс XContainer. Класс XContainer обла-

дает средствами для взаимодействия с потомками и является абстрактным базовым классом для `XElement` и `XDocument`.

Класс `XElement` имеет члены для управления атрибутами, в том числе свойства `Name` и `Value`. В довольно распространенной ситуации, когда у элемента есть единственный потомок типа `XText`, свойство `Value` объекта `XElement` инкапсулирует содержимое этого потомка для операций `get` и `set` и излишних переходов по дереву. Благодаря свойству `Value` у вас нет необходимости обращаться к узлам `XText` напрямую.

Класс `XDocument` представляет корень XML-дерева. Точнее говоря, он является *оболочкой* для корневого узла `XElement` и добавляет к нему объект `XDeclaration`, инструкции по обработке узлов и прочее "наполнение" корневого элемента. В отличие от DOM W3C, его присутствие необязательно. Вы можете загружать модель X-DOM, манипулировать ею и сохранять ее, не создавая объект `XDocument`! Отсутствие требования иметь объект `XDocument` позволяет вам легко и эффективно переносить поддерево какого-нибудь узла в другую иерархическую структуру X-DOM.

Загрузка и анализ

Классы `XElement` и `XDocument` предоставляют статические методы `Load` и `Parse` для построения дерева X-DOM по имеющемуся источнику:

- метод `Load` строит дерево X-DOM, получая информацию из файла, URI или от классов `TextReader` и `XmlReader`;

- метод `Parse` строит дерево X-DOM, получая информацию из строки.

> ***Примечание***
>
> Класс `XNode` предоставляет статический метод `ReadFrom`, который создает экземпляр узла любого типа и заполняет его данными, полученными от объекта `XmlReader`. В отличие от метода `Load` он прекращает работу, когда прочитает один (полный) узел. После этого вы можете продолжить чтение данных из объекта `XmlReader` вручную.
>
> А еще вы можете поступить наоборот и воспользоваться методами `CreateReader` и `CreateWriter` объекта `XNode` для создания объектов `XmlReader` и `XmlWriter` и чтения/записи информации в этот объект.

Например:

```
XDocument fromWeb = XDocument.Load
  ("http://albahari.com/sample.xml");
XElement fromFile = XElement.Load
  (@"e:\media\somefile.xml");
XElement config = XElement.Parse (
@"<configuration>
    <client enabled='true'>
      <timeout>30</timeout>
    </client>
  </configuration>");
```

Сохранение и сериализация

Вызов метода `ToString` для любого узла преобразует его содержимое в XML-строку, отформатиро-

ванную переводами строчек и отступами. (Вы можете отключить расстановку переводов строки и отступов, указав `SaveOptions.DisableFormatting` при вызове метода `ToString`.)

Классы `XElement` и `XDocument` предоставляют метод `Save`, который записывает дерево X-DOM в файл, объект `TextWriter` или объект `XmlWriter`. Если вы укажете файл, объявление XML будет записано автоматически. Кроме того, существует метод `WriteTo`, определенный в классе `XNode`, который принимает в качестве аргумента только объект `XmlWriter`.

Обработку XML-объявлений во время сохранения документа мы подробно обсудим в *разд. "Документы и объявления"*.

Создание экземпляра дерева X-DOM

Вы можете обойтись без методов `Load` и `Parse` и построить дерево X-DOM, вручную создавая объекты и добавляя их к родителям с помощью метода `Add` класса `XContainer`.

Чтобы сконструировать объекты `XElement` и `XAttribute`, просто укажите имя и значение:

```
XElement lastName = new XElement ("lastname",
"Bloggs");
lastName.Add (new XComment ("nice name"));
XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
```

```
customer.Add (new XElement ("firstname",
"Joe"));
customer.Add (lastName);
Console.WriteLine (customer.ToString());
```

Результат работы этого кода:

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

При конструировании объекта `XElement` указывать значение необязательно: вы можете указать только имя элемента и добавить содержимое позднее. Обратите внимание, что при указании значения в предыдущем примере было достаточно простой строки. Мы не создавали узел-потомок типа `XText` и не добавляли его в дерево явным образом. Вся эта работа выполняется в модели X-DOM автоматически, а вы имеете дело только со "значениями".

Функциональное конструирование

В предыдущем примере трудно представить себе структуру XML-документа по коду. Модель X-DOM поддерживает другой режим создания XML-элементов, называемый *функциональным конструированием* (по аналогии с функциональным программированием). В этом режиме вы строите все дерево в одном выражении:

```
XElement customer =
  new XElement ("customer", new XAttribute
("id", 123),
```

```
    new XElement ("firstname", "joe"),
    new XElement ("lastname", "bloggs",
      new XComment ("nice name")
    )
  );
```

Такой подход имеет два достоинства. Во-первых, код по форме напоминает XML-документ. Во-вторых, он может быть встроен в предложение `select` в LINQ-запросе. Например, следующий запрос LINQ к SQL напрямую проецирует результаты в дерево X-DOM:

```
XElement query =
  new XElement ("customers",
    from c in dataContext.Customers
    select
      new XElement ("customer",
        new XAttribute ("id", c.ID),
        new XElement ("firstname", c.FirstName),
        new XElement ("lastname", c.LastName,
          new XComment ("nice name")
        )
      )
  );
```

Мы вернемся к этой теме в *разд. "Проецирование в модель X-DOM"*.

Указание содержимого

Функциональное конструирование возможно благодаря тому, что конструкторы классов `XElement`

(и `XDocument`) перегружены и принимают массив объектов `params`:

```
public XElement (XName name, params object[]
content)
```

То же самое справедливо и в отношении метода `Add` класса `XContainer`:

```
public void Add (params object[] content)
```

Поэтому вы можете указывать любое количество объектов-потомков любого типа, когда строите отдельное дерево X-DOM или добавляете его к уже существующему. Здесь *все* считается допустимым содержимым.

Чтобы понять, как это работает, рассмотрим, как объект-содержимое обрабатывается на внутреннем уровне. Вот решения, последовательно принимаемые объектом `XContainer`:

1. Если объект равен `null`, он игнорируется.
2. Если базовым для объекта является класс `XNode` или `XStreamingElement`, он добавляется без изменений в коллекцию `Nodes`.
3. Если объект имеет тип `XAttribute`, он добавляется в коллекцию `Attributes`.
4. Если объект имеет тип `string`, он получает оболочку в виде узла `XText` и добавляется в коллекцию `Nodes`.
5. Если объект реализует интерфейс `IEnumerable`, выполняется перебор его элементов, и эти же правила применяются к каждому элементу в отдельности.
6. Если ни одно из вышеперечисленых условий не выполнено, объект преобразуется в строку, по-

лучает оболочку в виде узла `XText` и добавляется в коллекцию `Nodes`[1].

В конце концов, каждый объект оказывается в одной из двух коллекций, `Nodes` или `Attributes`. Кроме того, любой объект представляет допустимое содержимое, потому что для него всегда можно вызвать метод `ToString` и интерпретировать его как узел `XText`.

> ***Примечание***
>
> Перед вызовом метода `ToString` для произвольного типа объект `XContainer` проверяет, попадает ли этот тип в следующий список:
>
> `float, double, decimal, bool,`
>
> `DateTime, DateTimeOffset, TimeSpan`
>
> Если попадает, вызывается соответственно типизированный метод `ToString` вспомогательного класса `XmlConvert`, а не метод `ToString` самого объекта. Такой подход гарантирует, что данные можно будет пересылать в обоих направлениях и что они удовлетворяют стандартам XML-форматирования.

Автоматическое глубокое клонирование

Когда к элементу добавляется узел или атрибут (будь то посредством функционального конструи-

[1] Модель X-DOM оптимизирует этот шаг, сохраняя в строке простое текстовое содержимое. Узел `XTEXT` фактически не создается, пока вы не вызовете метод `Nodes()` класса `XContainer`.

рования или с помощью метода `Add`), свойство `Parent` добавляемого узла или атрибута устанавливается равным этому элементу. Однако узел может иметь только одного родителя. Если вы добавляете узел, уже имеющий родителя к другому элементу, автоматически выполняется глубокое клонирование узла. Автоматическое копирование узлов позволяет избежать побочных эффектов в дереве X-DOM. Это еще один отличительный признак функционального программирования.

Навигация и отправка запросов

Как и следовало ожидать, классы `XNode` и `XContainer` определяют методы и свойства для обхода дерева X-DOM. Однако, в отличие от обычной модели DOM, эти функции не возвращают коллекцию, реализующую интерфейс `IList<T>`. Вместо этого они возвращают либо одиночное значение, либо последовательность, реализующую интерфейс `IEnumerable<T>`. Предполагается, что вы затем выдадите LINQ-запрос к этой коллекции (либо переберете ее элементы оператором `foreach`). Таким образом можно выполнять сложные запросы или обычную навигацию, пользуясь при этом знакомым синтаксисом запросов LINQ.

> ***Примечание***
>
> В модели X-DOM, как и в самом языке XML, имена элементов и атрибутов чувствительны к регистру.

Навигация по узлам-потомкам

Тип возвращаемого значения	Члены	Работает с классом
XNode	FirstNode	XContainer
	LastNode	XContainer
Ienumerable <XNode>	Nodes()	XContainer*
	DescendantNodes()	XContainer*
	DescendantNodesAndSelf()	XElement*
XElement	Element(XName)	XContainer
Ienumerable <XElement>	Elements()	XContainer*
	Elements(XName)	XContainer*
	Descendants()	XContainer*
	Descendants(XName)	XContainer*
	DescendantsAndSelf()	XElement*
	DescendantsAndSelf(XName)	XElement*
bool	HasElements	XElement

Примечание

Функции, помеченные звездочкой в третьем столбце этой и других таблиц, работают и с *последовательностями* элементов того же типа. Например, вы можете вызвать метод Nodes либо для объекта XContainer, либо для последова-

тельности объекта `XContainer`. Это возможно благодаря методам расширения, определенным в пространстве имен `System.Xml.Linq`, то есть дополнительным операторам запросов, о которых мы уже упоминали.

Методы *FirstNode*, *LastNode* и *Nodes*

Методы `FirstNode` и `LastNode` предоставляют вам прямой доступ к первому и последнему узлу-потомку. Метод `Nodes` возвращает всех потомков в виде последовательности. Все три функции рассматривают только прямых потомков.

Чтение элементов

Метод `Elements` возвращает только узлы-потомки типа `Elements`: Например:

```
var bench = new XElement ("bench",
              new XElement ("toolbox",
                new XElement ("handtool",
"Hammer"),
                new XElement ("handtool",
"Rasp")
              ),
              new XElement ("toolbox",
                new XElement ("handtool",
"Saw"),
                new XElement ("powertool",
"Nailgun")
              ),
              new XComment ("Careful with the
nailgun")
            );
```

```
foreach (XElement e in bench.Elements( ))
  Console.WriteLine (e.Name + "=" + e.Value);
// Результат: toolbox=HammerRasp
              toolbox=SawNailgun
```

Следующий LINQ-запрос ищет элемент `toolbox` (ящик с инструментами — прим. перев.), в котором находится `Nailgun` (пневматический молоток — прим. перев.):

```
IEnumerable<string> query =
  from toolbox in bench.Elements()
  where toolbox.Elements().Any (tool =>
tool.Value == "Nailgun")
  select toolbox.Value;
// Результат: { "SawNailgun" }
```

> ***Примечание***
>
> Метод `Elements` эквивалентен запросу LINQ для метода `Nodes`. Наш предыдущий запрос мог бы начинаться так:
>
> ```
> from toolbox in
> bench.Nodes().OfType<XElement>()
> where ...
> ```

Метод `Elements` также может возвращать элементы с заданным именем. Например:

```
int x = bench.Elements ("toolbox").Count();
// 2
```

Этот код эквивалентен такому:

```
int x = bench.Elements()
              .Where (e => e.Name == "toolbox")
              .Count();  // 2
```

Кроме того, метод `Elements` определен как метод расширения, принимающий аргумент `IEnumerable<XContainer>`. Точнее говоря, этот аргумент имеет следующий тип:

```
IEnumerable<T> where T : XContainer
```

Поэтому метод может работать и с последовательностями элементов. С помощью метода `Elements` мы можем переписать запрос, который ищет подэлементы `handtool` в элементах `toolbox`:

```
from tool in bench.Elements ("toolbox").Elements
("handtool")
select tool.Value.ToUpper();
```

Первый вызов метода `Elements` вызывается к методу экземпляра класса `XContainer`, а второй — к методу расширения.

Чтение одного элемента

Метод `Element` (без "s" на конце) возвращает первый подходящий элемент с данным именем. Метод годится для выполнения простой навигации:

```
var settings = XElement.Load
("databaseSettings.xml");
string cx = settings.Element ("database")
                    .Element ("connectString")
                    .Value;
```

Метод `Element` эквивалентен вызову `Elements()` с последующим применением LINQ-оператора `FirstOrDefault`, у которого предикат задает имя искомого элемента. Если запрошенный элемент не найден, метод `Element` возвращает `null`.

Примечание

Вызов `Element("xyz").Value` закончится возбуждением исключения `NullReferenceException`, если элемент `xyz` отсутствует. Если вы предпочитаете `null` исключению, приведите тип `XElement` к строке, не опрашивая его свойство `Value`. То есть:

```
string xyz = (string) settings.Element
("xyz");
```

Это сработает, потому что `XElement` предусматривает явное преобразование к типу `string` как раз для этой цели!

Рекурсивные функции

Класс `XContainer`, помимо прочего, предоставляет методы `Descendants` и `DescendantNodes`, которые возвращают элементы-потомки или узлы-потомки, работая *рекурсивно*. Метод `Descendants` имеет необязательный параметр — имя элемента. Возвращаясь к предыдущему примеру, воспользуемся методом `Descendants` для поиска всех элементов `handtool`:

```
Console.WriteLine
  (bench.Descendants ("handtool").Count()); // 3
```

В результат включаются и узлы, имеющие и потомков, и узлы-листья. Приведем запрос, который извлекает из дерева X-DOM все комментарии, содержащие слово "careful" (осторожный — прим. перев.):

```
IEnumerable<string> query =
  from c in
bench.DescendantNodes().OfType<XComment>()
```

```
  where c.Value.Contains ("careful")
  orderby c.Value
  select c.Value;
```

Навигация по родительским элементам

Все объекты типа `XNode` имеют свойство `Parent` и методы `AncestorXXX` для навигации по родительским элементам. Родитель всегда имеет тип `XElement`:

Тип возвращаемого значения	Члены	Работает с классом
`XElement`	`Parent { get; }`	`XNode*`
`Enumerable <XElement>`	`Ancestors()`	`XNode*`
	`Ancestors(XName)`	`XNode*`
	`AncestorsAndSelf()`	`XElement*`
	`AncestorsAndSelf (XName)`	`XElement*`

Если `x` имеет тип `XElement`, то следующий код всегда `true`:

```
foreach (XNode child in x.Nodes())
  Console.WriteLine (child.Parent == x);
```

Однако этого нельзя сказать, если `x` имеет тип `XDocument`. Тип `XDocument` особенный: он может иметь потомков, но не может быть ничьим родителем! Чтобы обратиться к элементу типа `XDocument`, вы должны пользоваться свойством

`Document`. Этот подход работает с любым объектом в дереве X-DOM.

Метод `Ancestors` возвращает последовательность, первым элементом которой является `Parent`, вторым — `Parent.Parent` и так далее до корневого элемента.

> ***Примечание***
>
> Вы можете выполнить навигацию к корневому элементу с помощью LINQ-запроса `AncestorsAndSelf().Last()`.
>
> Другим способом добиться того же является вызов метода `Document.Root`, но он годится только для тех случаев, когда присутствует элемент `XDocument`.

Навигация по элементам одного уровня

Тип возвращаемого значения	Члены	Класс
`bool`	`IsBefore(XNode)`	`XNode`
	`IsAfter(XNode)`	`XNode`
`XNode`	`PreviousNode`	`XNode`
	`NextNode`	`XNode`
`Ienumerable <XNode>`	`NodesBeforeSelf()`	`XNode`
	`NodesAfterSelf()`	`XNode`

(окончание)

Тип возвращаемого значения	Члены	Класс
Ienumerable <XElement>	ElementsBeforeSelf()	XNode
	ElementsBeforeSelf (XName)	XNode
	ElementsAfterSelf()	XNode
	ElementsAfterSelf (XName)	XNode

С помощью методов `PreviousNode` и `NextNode` (а также `FirstNode`/`LastNode`) вы можете обходить узлы так, словно они организованы в связный список. На самом деле это не случайно: на внутреннем уровне они хранятся именно в связном списке.

> ***Примечание***
>
> Тип `XNode` пользуется *односвязным* списком, и поэтому метод `PreviousNode` работает очень медленно.

Навигация по атрибутам

Тип возвращаемого значения	Члены	Класс
bool	HasAttributes	XElement
XAttribute	Attribute(XName)	XElement
	FirstAttribute	XElement
	LastAttribute	XElement

(окончание)

Тип возвращаемого значения	Члены	Класс
IEnumerable<XAttribute>	Attributes()	XElement
	Attributes(XName)	XElement

Кроме прочего, тип XAttribute определяет свойства PreviousAttribute, NextAttribute и Parent.

Метод Attributes принимает имя атрибута и возвращает последовательность, содержащую либо ни одного, либо один элемент. В XML у элемента не может быть атрибутов с одинаковыми именами.

Редактирование дерева X-DOM

Вы можете отредактировать элементы и атрибуты следующим способом:

- вызвать SetValue или присвоить значение свойству Value;
- вызвать SetElementValue или SetAttributeValue;
- вызвать один из методов семейства RemoveXXX;
- вызвать один из методов семейства AddXXX или ReplaceXXX и передать ему новое содержимое.

Кроме того, можно присвоить новое значение свойству Name объекта XElement.

Обновление простых значений

Члены	Работает с классом
SetValue(object)	XElement, XAttribute
Value	XElement, XAttribute

Метод SetValue заменяет содержимое элемента или атрибута простым значением. Установка свойства Value делает то же самое, но при этом вы можете указывать только строковое значение. Мы подробно опишем обе функции в *разд. "Работа со значениями"*.

В результате вызова метода SetValue (или установки свойства Value) заменяются все узлы-потомки:

```
XElement settings = new XElement ("settings",
                      new XElement ("timeout",
30)
                    );
settings.SetValue ("blah");
Console.WriteLine (settings.ToString());
// Результат: <settings>blah</settings>
```

Редактирование узлов-потомков и атрибутов

Категория	Члены	Работает с классом
Добавление	Add(params object[])	XContainer
	AddFirst(params object[])	XContainer

(окончание)

Категория	Члены	Работает с классом
Удаление	`RemoveNodes()`	`XContainer`
	`RemoveAttributes()`	`XElement`
	`RemoveAll()`	`XElement`
Обновление	`ReplaceNodes(params object[])`	`XContainer`
	`ReplaceAttributes (params object[])`	`XElement`
	`ReplaceAll(params object[])`	`XElement`
	`SetElementValue(XName, object)`	`XElement`
	`SetAttributeValue(XName, object)`	`XElement`

Самыми удобными методами в этой группе являются последние два: `SetElementValue` и `SetAttributeValue`. Они фактически являются сокращениями для создания экземпляра `XElement` или `XAttribute` и последующего вызова метода `Add` для его родителя. В результате существующий элемент или атрибут с этим именем заменяется на новый:

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30);  // Добавляет узел-потомок
settings.SetElementValue ("timeout", 60);  // Меняет значение
```

Метод `Add` добавляет узел-потомок к элементу или документу. Метод `AddFirst` делает то же самое, но вставляет узел в начало коллекции, а не в конец.

Вы можете удалить все узлы-потомки или атрибуты за один шаг, вызвав метод `RemoveNodes` или `RemoveAttributes`. Метод `RemoveAll` эквивалентен вызову обоих этих методов.

Методы `ReplaceXXX` эквивалентны последовательному вызову методов `Remove` и `Add`. Они временно сохраняют входные данные, так что вызов `e.ReplaceNodes(e.Nodes())` работает без сюрпризов.

Обновление узла через его родителя

Члены	Работает с классом
`AddBeforeSelf (params object[])`	`XNode`
`AddAfterSelf (params object[])`	`XNode`
`Remove()`	`XNode*, XAttribute*`
`ReplaceWith (params object[])`	`XNode`

Методы `AddBeforeSelf`, `AddAfterSelf`, `Remove` и `ReplaceWith` не работают с потомками узла. Они манипулируют с коллекцией, в которой этот узел находится. Для этого нужно, чтобы у узла был родитель; в противном случае будет возбуждено исключение. Методы `AddBeforeSelf` и

`AddAfterSelf` полезны при вставке узла в произвольную позицию:

```
XElement items = new XElement ("items",
                   new XElement ("one"),
                   new XElement ("three")
                 );
items.FirstNode.AddAfterSelf (new XElement
("two"));
```

Вот что получится:

```
<items><one /><two /><three /></items>
```

Вставка узла в произвольную позицию в длинной последовательности элементов довольно эффективна, потому что узлы хранятся в виде связного списка.

Метод `Remove` удаляет текущий узел из списка узлов его родителя. Метод `ReplaceWith` делает то же самое, а затем вставляет на это место какое-то другое содержимое. Например:

```
XElement items = XElement.Parse
  ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith
  (new XComment ("One was here"));
```

Результат:

```
<items><!--one was here--><two /><three
/></items>
```

Удаление последовательности узлов и атрибутов

Благодаря методам расширения в пространстве имен `System.Xml.Linq` вы можете вызвать метод

`Remove` и для *последовательности* узлов и атрибутов. Рассмотрим такое дерево X-DOM:

```
XElement contacts = XElement.Parse (
@"<contacts>
    <customer name='Mary'/>
    <customer name='Chris' archived='true'/>
    <supplier name='Susan'>
      <phone archived='true'>
         012345678
         <!--confidential-->
      </phone>
    </supplier>
  </contacts>");
```

Следующий код удалит всех клиентов ("customer"):

```
contacts.Elements ("customer").Remove();
```

А следующий оператор удалит все архивные контакты (то есть, "Chris" исчезнет):

```
contacts.Elements()
  .Where (e => (bool?) e.Attribute ("archived")
== true)
  .Remove();
```

> ***Примечание***
>
> На внутреннем уровне методы `Remove` реализованы так: вначале они читают все подходящие элементы, занося их во временный список, а затем перебирают этот список, чтобы выполнить удаление. Так исключаются ошибки, которые могли бы возникнуть при одновременном удалении элементов и обращении к ним с запросом.

Если мы вместо `Elements()` напишем `Descendants()`, исчезнут все архивные элементы по всему дереву, и получится такой результат:

```
<contacts>
  <customer name="Mary" />
  <supplier name="Susan" />
</contacts>
```

В следующем коде удаляются все контакты, имеющие комментарий "confidential" (конфиденциальный — прим. перев.):

```
contacts.Elements()
  .Where (e => e.DescendantNodes()
                .OfType<XComment>()
                .Any (c => c.Value ==
"confidential")
          ).Remove();
```

Результат:

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

И, наконец, простой запрос, убирающий с дерева все узлы-комментарии:

```
contacts.DescendantNodes().OfType<XComment>().
Remove();
```

Работа со значениями

Классы `XElement` и `XAttribute` имеют свойство `Value` типа `string`. Если у элемента есть один узел-

потомок типа `XText`, свойство `Value` класса `XElement` оказывается удобным средством обращения к содержимому этого узла. У класса `XAttribute` свойство `Value` является всего лишь свойством атрибута.

Несмотря на различия в способах хранения, дерево X-DOM обеспечивает единый набор операций для работы со значениями элементов и атрибутов.

Установка значений

Существуют два способа присваивания значений: вызов метода `SetValue` или обращение к свойству `Value`. Метод `SetValue` более гибок, потому что принимает не только строки, но и данные других простых типов:

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value);
// Результат: 2007-03-02T16:39:10.734375+09:00
```

Вместо всего этого можно было просто установить у элемента свойство `Value`, но тогда нам пришлось бы вручную преобразовать `DateTime` в строку. При этом мы не смогли бы ограничиться вызовом метода `ToString`, а должны были бы воспользоваться методом `XmlConvert` для получения результата, удовлетворяющего стандарту XML.

Когда вы передаете *значение* конструктору `XElement` или `XAttribute`, происходит аналогичное преобразование в нестроковый тип. Это гарантирует корректное форматирование объектов `DateTime`,

то есть, слово `true` будет всегда написано в нижнем регистре, а вместо `double.NegativeInfinity` будет написано "-INF".

Чтение значений

Чтобы пройти в обратном направлении и преобразовать значение свойства `Value` в базовый тип, вы просто приводите к этому типу элемент `XElement` или `XAttribute`. На первый взгляд кажется, что такое приведение типа не сработает, но у вас все получится! Пример:

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;
XAttribute a = new XAttribute ("resolution",
1.234);
double res = (double) a;
```

Элемент или атрибут не хранятся в виде объекта `DateTime` или числа; они всегда сохраняются в текстовом виде, а в нужный момент выполняется их анализ. Кроме того, они не "помнят" свой оригинальный тип, и вы должны преобразовать их тип корректно, чтобы избежать ошибки на этапе выполнения. Для повышения устойчивости кода вы можете поместить приведение типа в блок `try/catch` и отслеживать исключение `FormatException`).

Типы `XElement` и `XAttribute` могут быть явно приведены к следующим типам:

- ко всем стандартным числовым типам;
- к типам `string`, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`;

- к Nullable<>-версиям всех вышеперечисленных типов-значений.

Приведение к типам, допускающим значение null, полезно в сочетании с методами Element и Attribute, потому что приведение типа будет работать даже при отсутствии запрошенного имени. Например, если у элемента x нет элемента timeout, то первая строка возбудит сообщение об ошибке на этапе выполнения, а вторая — нет:

```
int timeout = (int) x.Element ("timeout");
// Ошибка
int? timeout = (int?) x.Element ("timeout");
// OK
```

Вы можете убрать тип, допускающий значение null, из окончательного результата с помощью оператора ??. Следующий код возвращает значение 1.0, если атрибут resolution не существует:

```
double resolution =
  (double?) x.Attribute ("resolution") ?? 1.0;
```

Впрочем, приведение к типу, допускающему значение null, не избавит вас от проблем, если элемент или атрибут существует и имеет пустое (или некорректно отформатированное) значение. В этом случае вам придется "ловить" исключение FormatException.

Приведение типов можно применять и в LINQ-запросах. Следующий код возвращает "John":

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100'
/>
      <customer id='2' name='John' credit='150'
/>
```

```
      <customer id='3' name='Anne' />
    </data>");
IEnumerable<string> query =
  from cust in data.Elements()
  where (int?) cust.Attribute ("credit") > 100
  select cust.Attribute ("name").Value;
```

Приведение к типу `int`, допускающему значение `null`, позволяет избежать исключения `NullReferenceException` для клиента 'Anne', у которого нет атрибута `credit`. Другое решение состоит в добавлении предиката в предложение `where`:

```
where cust.Attributes ("credit").Any()
&& (int) cust.Attribute...
```

Те же принципы действуют и при опросе значений элементов.

Значения и узлы со смешанным содержимым

Имея в своем распоряжении значение свойства `Value`, вы, возможно, задаете себе вопрос, а придется ли вам вообще напрямую обращаться к узлам типа `XText`. Ответ на него такой: придется, когда у вас появится смешанное содержимое. Например:

```
<summary>
  An XAttribute is <bold>not</bold> an XNode
</summary>
```

Здесь простого свойства `Value` недостаточно для получения содержимого элемента `summary`. Он имеет три потомка: узел `XText`, затем узел

XElement, и еще один узел XText. Вот как он конструируется:

```
XElement summary = new XElement ("summary",
                       new XText ("An XAttribute
is "),
                       new XElement ("bold",
"not"),
                       new XText (" an XNode")
                     );
```

Интересно, что мы все равно можем опросить свойство Value элемента summary, и никакое исключение не возникнет. Мы просто получим конкатенацию всех значений потомков:

```
An XAttribute is not an XNode
```

Здесь допустимо присвоить новое значение свойству Value элемента summary. В результате все потомки будут заменены на один узел типа XText.

Автоматическая конкатенация элементов *XText*

Когда вы добавляете к элементу XElement простое содержимое, новый потомок не создается. Вместо этого к существующему потомку XText добавляется новое поддерево X-DOM. В следующем примере элементы e1 и e2 получают по одному потомку XText, значением которого является текст HelloWorld:

```
var e1 = new XElement ("test", "Hello");
    e1.Add ("World");
var e2 = new XElement ("test", "Hello",
"World");
```

Если же вы будете специально создавать узлы `XText`, то появятся несколько потомков:

```
var e = new XElement ("test",
                      new XText ("Hello"),
                      new XText ("World"));
Console.WriteLine (e.Value);             //
HelloWorld
Console.WriteLine (e.Nodes().Count());   // 2
```

`XElement` не станет выполнять конкатенацию двух узлов `XText`, поэтому идентичности их объектов сохраняются.

Документы и объявления

Класс *XDocument*

Класс `XDocument` представляет собой оболочку для корневого элемента `XElement` и позволяет вам добавлять к последнему объект типа `XDeclaration`, указывать инструкции по обработке и тип документа, а также сопровождать его комментариями. Объект класса `XDocument` не является обязательным; его можно опустить или проигнорировать. В отличие от документа в модели W3C DOM, он не служит цементирующей основой для остальных элементов.

Класс `XDocument` предоставляет те же функциональные конструкторы, что и класс `XElement`. Кроме того, поскольку базовым для него является класс `XContainer`, он поддерживает методы `AddXXX`, `RemoveXXX` и `ReplaceXXX`. В отличие от

класса XElement, класс XDocument накладывает ограничения на свое содержимое. Это могут быть:

- один объект типа XElement ("корень");
- один объект типа XDeclaration;
- один объект типа XDocumentType (для ссылки на DTD);
- любое количество объектов типа XProcessingInstruction;
- любое количество объектов типа XComment.

> ***Примечание***
>
> Из перечисленных объектов обязателен только корень XElement. Он делает объект XDocument корректным. Объект XDeclaration можно опустить, и в этом случае при сериализации будут действовать настройки по умолчанию.

Простейший допустимый объект XDocument состоит только из корневого элемента:

```
var doc = new XDocument (
            new XElement ("test", "data")
          );
```

Обратите внимание на отсутствие объекта XDeclaration. Файл, сгенерированный методом doc.Save, тем не менее будет содержать объявление XML, сгенерированное по умолчанию.

В следующем примере создается простой, но корректный XHTML-файл, иллюстрирующий все конструкции, которые может принимать объект XDocument:

```
var styleInstruction = new
XProcessingInstruction (
```

```
  "xml-stylesheet", "href='styles.css'
type='text/css'");
var docType = new XDocumentType ("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd",
  null);
XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement (ns + "html",
    new XElement (ns + "head",
      new XElement (ns + "title", "An XHTML
page")),
    new XElement (ns + "body",
      new XElement (ns + "p", "This is the
content"))
  );
var doc =
  new XDocument (
    new XDeclaration ("1.0", "utf-8", "no"),
    new XComment ("Reference a stylesheet"),
    styleInstruction,
    docType,
    root);
doc.Save ("test.html");
```

Файл по имени test.html будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8"
standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css'
type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"
```

```
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
  <body>
    <p>This is the content</p>
  </body>
</html>
```

Класс `XDocument` имеет свойство `Root`, которое служит средством для обращения к единственному элементу `XElement`. Обратную ссылку обеспечивает свойство `Document` класса `XObject`, и это справедливо в отношении всех объектов в дереве:

```
Console.WriteLine (doc.Root.Name.LocalName);
// html
XElement bodyNode = doc.Root.Element (ns +
"body");
Console.WriteLine (bodyNode.Document == doc);
// True
```

> ***Примечание***
>
> Объект `XDeclaration` не является объектом `XNode` и поэтому не находится в коллекции `Nodes` объекта-документа (в отличие от комментариев, инструкций по обработке и корневого элемента). Он присвоен специальному свойству, называемому `Declaration`. Именно поэтому слово "True" в предыдущем примере повторяется четыре раза, а не пять.

Вспомним, что у потомков объекта-документа нет свойства `Parent`:

```
Console.WriteLine (doc.Root.Parent == null); //
True
```

```
foreach (XNode node in doc.Nodes())
  Console.Write (node.Parent == null);       //
TrueTrueTrueTrue
```

XML-объявления

Стандартный XML-файл начинается с объявления, например, такого:

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?>
```

XML-объявление гарантирует, что файл будет корректно разобран и интерпретирован классом-читателем. При генерировании XML-объявлений классы `XElement` и `XDocument` придерживаются следующих правил:

- вызов метода `Save` с именем файла всегда приводит к созданию объявления;
- вызов метода `Save` с классом `XmlWriter` в качестве аргумента приводит к созданию объявления, если класс `XmlWriter` не проинструктирован иначе;
- вызов метода `ToString` никогда не приводит к созданию XML-объявления.

> ***Примечание***
>
> Вы можете сообщить классу `XmlWriter` о том, что не нужно создавать объявление, установив свойства `OmitXmlDeclaration` и `ConformanceLevel` объекта `XmlWriterSettings` при конструировании класса `XmlWriter`.

Наличие или отсутствие объекта `XDeclaration` не влияет на то, как написано XML-объявление. Цель

этого объекта в том, чтобы *сообщать, как должна происходить XML-сериализация*, уточняя два аспекта:

- какая текстовая кодировка должна быть применена;
- какие значения должны быть у атрибутов `encoding` и `standalone` в XML-объявлении (если оно должно быть создано).

Конструктор класса `XDeclaration` принимает три аргумента, которые соответствуют атрибутам `version`, `encoding` и `standalone`. В следующем примере файл test.xml имеет кодировку UTF-16:

```
var doc = new XDocument (
            new XDeclaration ("1.0", "utf-16",
"yes"),
            new XElement ("test", "data")
          );
doc.Save ("test.xml");
```

> ***Примечание***
>
> Класс, создающий XML-документ, проигнорирует все, что вы напишете в качестве версии XML, и подставит `"1.0"`.

Указывая кодировку, вы должны соблюдать код IETF и писать, например, `"utf-16"`, — как это должно быть в XML-объявлении.

Имена и пространства имен

Аналогично тому как типы .NET имеют пространства имен, свои пространства имен есть у XML-элементов и атрибутов.

Пространства имен XML позволяют достичь двух целей. Во-первых, они, как и пространства имен C#, помогают избежать коллизий имен. Это важно, когда вы сливаете данные двух XML-файлов. Во-вторых, пространства имен придают именам *абсолютный* смысл. Например, имя "nil" может обозначать все, что угодно. Однако в пространстве имен **http://www.w3.org/2001/XMLSchema-instance** оно имеет смысл, близкий к "null" в терминах C#, и существуют определенные правила по его употреблению.

В XML пространство имен определяется с атрибутом `xmlns`:

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

Здесь `xmlns` — это специальный зарезервированный атрибут. Использованный таким образом, он выполняет две функции:

- указывает пространство имен для данного элемента;
- указывает пространство имен, принимаемое по умолчанию для всех потомков этого элемента.

Вы также можете указать пространство имен с помощью *префикса*. Префикс — это псевдоним, который вы можете присвоить пространству имен, чтобы избежать повторения. Применение префикса состоит из двух шагов: из его *объявления* и *использования*. Их можно выполнить за одну операцию:

```
<nut:customer
xmlns:nut="OReilly.Nutshell.CSharp"/>
```

Здесь происходят два разных события. Конструкция, стоящая справа (`xmlns:nut="..."`), определяет префикс `nut` и делает его доступным данному

элементу и его потомкам. Конструкция слева (`nut:customer`) присваивает только что созданный префикс элементу `customer`.

Элемент, снабженный префиксом, *не определяет* пространство имен по умолчанию для своих потомков. В следующем XML-коде у элемента `firstname` пространство имен пусто:

```
<nut:customer
nut:xmlns="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

Чтобы дать префикс `OReilly.Nutshell.CSharp` элементу `firstname`, мы должны будем написать:

```
<nut:customer
xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

XML позволяет определять префиксы исключительно для элементов-потомков, не присваивая префиксы родителю. В следующем коде определяются префиксы `i` и `z`, но сам элемент `customer` остается в пустом пространстве имен:

```
<customer
  xmlns:i="http://www.w3.org/2001/XMLSchema-
instance"

xmlns:z="http://schemas.microsoft.com/Serializat
ion/">
  ...
</customer>
```

(Оба префикса в нашем примере представляют собой URI. Использование ваших собственных URI

гарантирует уникальность пространств имен в вашем коде и являются стандартной практикой.)

Вы можете присваивать пространства имен и атрибутам тоже. Основное отличие при этом будет состоять в необходимости всегда указывать префикс. Например:

```
<customer
  xmlns:nut="OReilly.Nutshell.CSharp"
nut:id="123" />
```

Еще одно отличие заключается в том, что неквалифицированный атрибут всегда имеет пустое пространство имен; он не наследует пространство имен от родительского элемента.

Указание пространства имен в модели X-DOM

До сих пор в этой книге мы пользовались простыми строками для указания имен элементов `XElement` и `XAttribute`. Простая строка соответствует XML-имени с пустым пространством имен. Здесь уместна аналогия с .NET-типом, определенном в глобальном пространстве имен.

Существуют два способа уточнения XML-пространства имен. Первый состоит в заключении его в фигурные скобки и помещении перед локальным именем. Например:

```
var e = new XElement
  ("{http://domain.com/xmlspace}customer",
"Bloggs");
Console.WriteLine (e.ToString());
```

В результате получится такой XML-код:

```
<customer xmlns="http://domain.com/xmlspace">
  Bloggs
</customer>
```

Второй, более производительный, подход заключается в использовании типов `XNamespace` и `XName`. Вот их определения:

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}
public sealed class XName
{
  public string LocalName { get; }
  public XNamespace Namespace { get; }   // Необязательно
}
```

Оба типа определяют неявное приведение к ним от типа `string`, так что следующий код корректен:

```
XNamespace ns   = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  =
"{http://domain.com/xmlspace}customer";
```

Кроме того, тип `XName` перегружает операцию `+`, позволяя вам объединять пространство имен и имя, не пользуясь фигурными скобками:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName);
// Результат:
{http://domain.com/xmlspace}customer
```

Все конструкторы и методы модели X-DOM, принимающие в качестве аргумента имя элемента или атрибута, на самом деле принимают объект типа `XName`, а не `string`. Во всех предыдущих примерах мы могли передавать строки исключительно благодаря неявному приведению типов.

Пространство имен указывается одинаково и для элемента, и для атрибута:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
              new XAttribute (ns + "id", 123)
           );
```

X-DOM и пространства имен по умолчанию

Модель X-DOM игнорирует пространства имен, принятые по умолчанию, вплоть до момента фактического вывода XML-документа. Это означает, что, когда вы конструируете элемент-потомок типа `XElement`, вы должны явно указывать пространство имен в случае необходимости. Помните, что оно *не будет* унаследовано от родителя:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer",
"Bloggs"),
             new XElement (ns + "purchase",
"Bicycle")
           );
```

Зато при чтении и выводе XML-документа модель X-DOM учитывает пространства имен по умолчанию:

```
Console.WriteLine (data.ToString());
```

Результат:

```
<data xmlns="http://domain.com/xmlspace">
  <customer>Bloggs</customer>
  <purchase>Bicycle</purchase>
</data>
```

```
Console.WriteLine (data.Element (ns +
"customer").ToString());
```

Результат:

```
<customer
xmlns="http://domain.com/xmlspace">Bloggs
</customer>
```

Если вы сконструируете потомки элемента `XElement`, не указывая пространства имен, то есть так:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement ("customer",
"Bloggs"),
             new XElement ("purchase",
"Bicycle")
           );
Console.WriteLine (data.ToString());
```

вы получите другой результат:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

Еще одна ловушка подстерегает вас, когда вы не указываете пространство имен при навигации по дереву X-DOM:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
             new XElement (ns + "customer",
"Bloggs"),
             new XElement (ns + "purchase",
"Bicycle")
           );
XElement x = data.Element (ns + "customer");
// ok
XElement y = data.Element ("customer");
// null
```

Если вы построите дерево X-DOM, не задавая пространства имен, вы впоследствии сможете присвоить каждому элементу одно пространство имен:

```
foreach (XElement e in
data.DescendantsAndSelf())
  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

Префиксы

Модель X-DOM относится к префиксам точно так же, как к пространствам имен и пользуется ими исключительно для сериализации. Сказанное означает, что вы вполне можете обойтись без префиксов и больше о них не думать! Единственное, что может заставить вас вспомнить о префиксах — это забота об эффективности вывода в XML-файл. В качестве примера рассмотрим такой код:

```
XNamespace ns1 = "http://test.com/space1";
XNamespace ns2 = "http://test.com/space2";
```

```
var mix = new XElement (ns1 + "data",
            new XElement (ns2 + "element",
"value"),
            new XElement (ns2 + "element",
"value"),
            new XElement (ns2 + "element",
"value")
          );
```

По умолчанию класс `XElement` сериализует его следующим образом:

```
<data xmlns="http://test.com/space1">
  <element
xmlns="http://test.com/space2">value</element>
  <element
xmlns="http://test.com/space2">value</element>
  <element
xmlns="http://test.com/space2">value</element>
</data>
```

Здесь легко заметить необязательные повторения. Чтобы избавиться от них, *не нужно* менять способ конструирования дерева X-DOM. Просто дайте сериализатору соответствующие указания до вывода XML-документа. Для этого добавьте атрибуты, определяющие те префиксы, которые вы хотите применить. Обычно такие атрибуты придаются корневому элементу:

```
mix.SetAttributeValue (XNamespace.Xmlns + "ns1",
ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2",
ns2);
```

Здесь префикс "ns1" присваивается переменной `ns1`, имеющей тип `XNamespace`, а префикс "ns2" — переменной `ns2`. Дерево X-DOM автоматически выберет эти атрибуты при сериализации и "ужмет"

результирующий XML-документ. Вот что вы получите, вызвав `ToString` для `mix`:

```
<ns1:data xmlns:ns1="http://test.com/space1"
          xmlns:ns2="http://test.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Префиксы не влияют на то, как вы конструируете, опрашиваете и редактируете дерево X-DOM. Занимаясь этими видами деятельности, вы можете игнорировать префиксы и пользоваться полными именами. Префиксы "вступают в игру", только когда вы преобразуете дерево в XML-файлы или потоки и обратно.

Префиксы также учитываются при сериализации атрибутов. В следующем примере мы описываем дату рождения клиента и его кредит как `"nil"`, используя атрибут, соответствующий стандарту W3C. Строчка, выделенная полужирным шрифтом, гарантирует, что префикс будет сериализован без излишнего повторения пространства имен:

```
XNamespace xsi =
  "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);
var cust =
  new XElement ("customers",
    new XAttribute (XNamespace.Xmlns + "xsi",
xsi),
    new XElement ("customer",
    new XElement ("lastname", "Bloggs"),
    new XElement ("dob", nil),
```

```
    new XElement ("credit", nil)
    )
  );
```

Результирующий XML-документ:

```
<customers
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

Для краткости мы заранее объявили, что `nil` имеет тип `XAttribute`, чтобы два раза воспользоваться им в построении DOM. Вы можете сослаться на один атрибут дважды, потому что он автоматически копируется в случае необходимости.

Проецирование в модель X-DOM

Вы можете использовать LINQ-запросы для проецирования в X-DOM. Источником может служить все, к чему допустимо обратиться с LINQ-запросом, то есть

- классы сущностей в технологии запросов LINQ к SQL;
- локальные коллекции;
- другие деревья X-DOM.

Каким бы ни был источник, стратегия использования LINQ порождения дерева X-DOM всегда одна и та же. Вначале вы пишете выражение для *функционального конструирования*, которое создает дерево X-DOM требуемой формы, а затем строите LINQ-запрос к этому выражению.

Предположим в качестве примера, что нам нужно получить из базы данных информацию о клиентах и записать ее в такой XML-документ:

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

Начнем с составления выражения для функционального конструирования дерева X-DOM и воспользуемся простыми литералами:

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute
("id", 1),
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
    )
  );
```

Теперь превратим этот код в проекцию и построим вокруг него LINQ-запрос:

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
```

```
    select
      new XElement ("customer", new XAttribute
("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
```

Получится такой результат:

```
<customers>
  <customer id="1">
    <name>Tom</firstname>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</firstname>
    <buys>2</buys>
  </customer>
    ...
</customers>
```

Внешний запрос определяет границу, после которой запрос превращается из удаленного запроса LINQ к SQL в локальный LINQ-запрос к перечисляемой последовательности. Конструктор класса `XElement` ничего не знает об интерфейсе `IQueryable<>` и поэтому форсирует перебор элементов для запроса LINQ к SQL и, следовательно, выполнение SQL-оператора.

Исключение пустых элементов

Вернемся к предыдущему примеру и предположим, что мы, кроме прочего, интересуемся инфор-

мацией о последней дорогостоящей покупке, сделанной клиентом. Можно написать такой код:

```
var customers =
  new XElement ("customers",
    from c in dataContext.Customers
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault()
    select
      new XElement ("customer",
        new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys",
c.Purchases.Count),
        new XElement ("lastBigBuy",
          new XElement ("description",
            lastBigBuy == null
              ? null : lastBigBuy.Description),
          new XElement ("price",
            lastBigBuy == null
              ? 0m : lastBigBuy.Price)
        )
      )
  );
```

Однако он породит пустые элементы у тех клиентов, которые не покупали дорогие товары. (Если бы это был локальный запрос, а не запрос LINQ к SQL, он возбудил бы исключение `NullReferenceException`.) В таких случаях было бы разумно вовсе удалить узел `lastBigBuy`. Мы можем добиться этого, создав

для конструктора элемента `lastBigBuy` оболочку в виде условного оператора:

```
    select
      new XElement ("customer",
        new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys",
c.Purchases.Count),
        lastBigBuy == null ? null :
          new XElement ("lastBigBuy",
            new XElement ("description",
              lastBigBuy.Description),
            new XElement ("price",
lastBigBuy.Price)
```

Для клиентов, не совершавших дорогие покупки, будут сгенерированы элементы `null`, а не пустые элементы `XElement`. Именно этого мы и хотели, потому что содержимое `null` будет просто проигнорировано.

Проецирование в поток

Если вы проецируете данные в дерево X-DOM только для того, чтобы сохранить его методом `Save` (или вызвать для него метод `ToString`), вы можете эффективнее использовать память с помощью класса `XStreamingElement`. Он является "урезанной" версией класса `XElement` и реализует семантику *отложенной загрузки* по отношению к содержимому узла-потомка. Чтобы воспользовать-

ся этим классом, просто замените внешние элементы `XElement` на элементы `XStreamingElement`:

```
var customers =
  new XStreamingElement ("customers",
    from c in dataContext.Customers
    select
      new XStreamingElement ("customer",
        new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
    );
customers.Save ("data.xml");
```

Запросы, переданные конструктору класса `XStreamingElement`, не подвергаются перебору, пока вы не вызовете метод `Save`, `ToString` и `WriteTo` для элемента. Это позволяет обойтись без загрузки в память сразу всего дерева X-DOM. Обратной стороной медали является повторное выполнение запросов при повторном вызове метода `Save`. Кроме того, у вас не будет возможности обойти содержимое узла-потомка элемента `XStreamingElement`, поскольку он не предоставляет методы, аналогичные `Elements` или `Attributes`.

Класс `XStreamingElement` не базируется на классе `XObject` (и ни на каком другом), потому что имеет ограниченный набор членов. Кроме членов `Save`, `ToString` и `WriteTo`, у него еще есть только два:

- метод `Add`, принимающий содержимое, как конструктор;
- свойство `Name`.

Класс `XStreamingElement` не позволяет вам *читать* содержимое из потока; для этого вы должны воспользоваться классом `XmlReader` применительно к дереву X-DOM.

Преобразование дерева X-DOM

Вы можете преобразовать дерево X-DOM, выполнив его повторное проецирование. Предположим, нам нужно преобразовать XML-файл msbuild, используемый компилятором C# и средой Visual Studio для описания проекта, в файл простого формата, пригодный для генерирования отчета. Файл msbuild выглядит так:

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/dev...>
  <PropertyGroup>
    <Platform Condition=" '$(Platform)' == '' ">
      AnyCPU
    </Platform>
    <ProductVersion>9.0.11209</ProductVersion>
    ...
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="ObjectGraph.cs" />
    <Compile Include="Program.cs" />
    <Compile
Include="Properties\AssemblyInfo.cs" />
    <Compile Include="Tests\Aggregation.cs" />
    <Compile
Include="Tests\Advanced\RecursiveXml.cs" />
  </ItemGroup>
```

```
  <ItemGroup>
    ...
  </ItemGroup>
   ...
</Project>
```

Пусть нас интересуют только файлы:

```
<ProjectReport>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <File>Properties\AssemblyInfo.cs</File>
  <File>Tests\Aggregation.cs</File>
  <File>Tests\Advanced\RecursiveXml.cs</File>
</ProjectReport>
```

Следующий запрос выполняет необходимое преобразование:

```
XElement project = XElement.Load
("myProjectFile.csproj");
XNamespace ns = project.Name.Namespace;
var query =
  new XElement ("ProjectReport",
    from compileItem in
      project.Elements (ns + "ItemGroup")
             .Elements (ns + "Compile")
    let include = compileItem.Attribute
("Include")
    where include != null
    select new XElement ("File", include.Value)
  );
```

Запрос вначале извлекает все элементы `ItemGroup`, а затем вызывает метод расширения `Elements` для получения плоской последовательности подэлементов `Compile`.

Обратите внимание, что нам пришлось указать XML-пространство имен (так как все содержимое оригинального файла наследует пространство имен, определяемое элементом `Project`), и поэтому имя локального элемента, например, `ItemGroup`, не будет работать без префикса. Затем мы извлекли значение атрибута `Include` и спроецировали его как элемент.

Предметный указатель

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

А

В

Г

Д

З

И

К

Л

М

Н

О

П

С

Т

У

Ф

Ц

Э

Джозеф Албахари, Бен Албахари

LINQ. Карманный справочник

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Перевод с английского	*Сергея Иноземцева*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.09.08.
Формат $84\times100^{1}/_{32}$. Печать офсетная. Усл. печ. л. 11,625.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12