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.

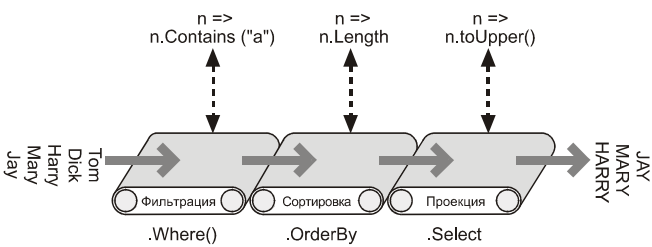

Рис. 1. Цепочка операторов запроса

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.

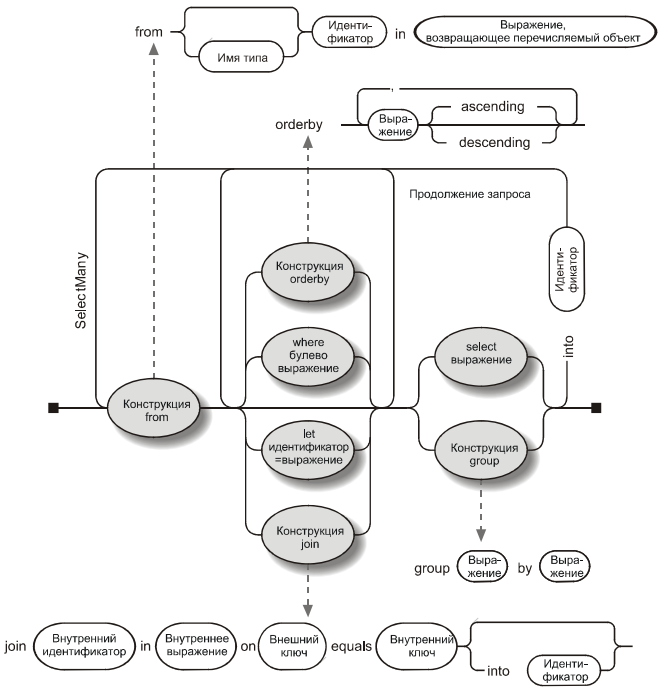

Рис. 2. Синтаксис, облегчающий восприятие запроса

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-