Синтаксис запроса, описанный в предыдущих разделах, применяется при формировании

большинства запросов в С#. Он удобен, эффективен и компактен, хотя и не

является единственным способом формирования запросов. Другой способ состоит

в использовании *методов запроса,* которые могут вызываться для любого перечислимого

объекта, например массива.

**Основные методы запроса**

Методы запроса определяются в классе System.Linq.Enumerable и реализуются

в виде *методов расширения* функций обобщенной формы интерфейса IEnumerable<T>.

(Методы запроса определяются также в классе System.Linq.Queryable, расширяющем

функции обобщенной формы интерфейса IQueryable<T>, но этот интерфейс

в настоящей главе не рассматривается.) Метод расширения дополняет функции другого

класса, но без наследования.

В классе Enumerable предоставляется немало методов запроса, но основными считаются

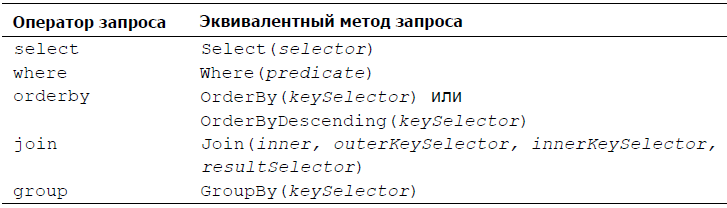
те методы, которые соответствуют описанным ранее операторам запроса. Эти

методы перечислены ниже вместе с соответствующими операторами запроса. Следует,

однако, иметь в виду, что эти методы имеют также перегружаемые формы, а здесь они

представлены лишь в самой простой своей форме. Но именно эта их форма используется

чаще всего.



За исключением метода Join(), остальные методы запроса принимают единственный

аргумент, который представляет собой объект некоторой разновидности

обобщенного типа Func<T, TResult>. Это тип встроенного делегата, объявляемый

следующим образом:

delegate TResult Func<in Т, out TResult>(T arg)

где TResult обозначает тип результата, который дает делегат, а Т — тип элемента.

В методах запроса аргументы *selector, predicate* или *keySelector* определяют

действие, которое предпринимает метод запроса. Например, в методе Where() аргумент

*prediсаte* определяет порядок отбора данных в запросе. Каждый метод запроса

возвращает перечислимый объект. Поэтому результат выполнения одного метода запроса

можно использовать для вызова другого, соединяя эти методы в цепочку.

Метод Join() принимает четыре аргумента. Первый аргумент (*inner*) представляет

собой ссылку на вторую объединяемую последовательность, а первой является последовательность,

для которой вызывается метод Join(). Селектор ключа для первой последовательности

передается в качестве аргумента *outerKeySelector,* а селектор ключа для

второй последовательности — в качестве аргумента *LnnerKeySelector.* Результат объединения

обозначается как аргумент *resultSelector.* Аргумент *outerKeySelector*

имеет тип Func<TOuter, ТКеу>, аргумент *innerKeySelector —* тип Func<TInner,

ТКеу>, тогда как аргумент *resultSelector* — тип Func<TOuter, Tinner, TResult>,

где TOuter — тип элемента из вызывающей последовательности; Tinner — тип элемента

из передаваемой последовательности; TResult — тип элемента из объединяемой

в итоге последовательности, возвращаемой в виде перечислимого объекта.

Аргумент метода запроса представляет собой метод, совместимый с указываемой

формой делегата Func, но он не обязательно должен быть явно объявляемым методом.

На самом деле вместо него чаще всего используется лямбда-выражение. Как пояснялось

в главе 15, лямбда-выражение обеспечивает более простой, но эффективный

способ определения того, что, по существу, является анонимным методом, а компилятор

C# автоматически преобразует лямбда-выражение в форму, которая может быть

передана в качестве параметра делегату Func. Благодаря тому что лямбда-выражения

обеспечивают более простой и рациональный способ программирования, они используются

во всех примерах, представленных далее в этом разделе.

**Формирование запросов с помощью методов запроса**

Используя методы запроса одновременно с лямбда-выражениями, можно формировать

запросы, вообще не пользуясь синтаксисом, предусмотренным в C# для запросов.

Вместо этого достаточно вызвать соответствующие методы запроса. Обратимся

сначала к простому примеру. Он представляет собой вариант первого примера программы

из этой главы, переделанный с целью продемонстрировать применение методов

запроса Where() и Select() вместо соответствующих операторов.

(***glava19\_16***)

class SimpQuery

{

static void Main()

{

int[] nums = { 1, -2, 3, 0, -4, 5 };

//use methods Where() and Select() to use

//simple query

var posNums = nums.Where(n => n > 0).Select(r => r);

Console.Write("Positive numbers from nums: ");

foreach (var i in posNums) Console.Write(i + " ");

Console.WriteLine();

}

}

Обратите особое внимание в данной программе на следующую строку кода.

var posNums = nums.Where(n => n > 0).Select(r => r);

В этой строке кода формируется запрос, сохраняемый в переменной posNums. По

этому запросу, в свою очередь, формируется последовательность положительных значений,

извлекаемых из массива nums. Для этой цели служит метод Where(), отбирающий

запрашиваемые значения, а также метод Select(), избирательно формирующий

из этих значений окончательный результат.

Формально метод Select() в рассматриваемом здесь примере не нужен, поскольку

это простой запрос. Ведь последовательность, возвращаемая методом Where(), уже

содержит конечный результат. Но окончательный выбор можно сделать и по более

сложному критерию, как это было показано ранее на примерах использования синтаксиса

запросов.

Как и следовало ожидать, в цепочку можно объединять и другие операции над данными,

получаемыми по запросу. Например, по следующему запросу выбираются положительные

значения, которые затем сортируются по убывающей и возвращаются в

виде результирующей последовательности:

var posNums = nums.Where(n => n > 0).OrderByDescending(j => j);

где выражение j => j обозначает, что упорядочение зависит от входного параметра,

который является элементом данных из последовательности, получаемой из метода

Where().

В приведенном ниже примере демонстрируется применение метода запроса

GroupBy(). Это измененный вариант представленного ранее примера.

(***glava19\_17***)

class GroupByDemo

{

static void Main()

{

string[] websites =

{

"hsNameA.com", "hsNameB.net", "hsNameC.net",

"hsNameD.com", "hsNameE.org", "hsNameF.org",

"hsNameG.tv", "hsNameH.net", "hsNameI.tv"

};

//query methods

var webAddrs = websites.Where(w => w.LastIndexOf('.') != -1).

GroupBy(x => x.Substring(x.LastIndexOf('.')));

foreach(var sites in webAddrs)

{

Console.WriteLine("Web - sites: " + sites.Key);

foreach (var site in sites)

Console.WriteLine(" " + site);

Console.WriteLine();

}

}

}

Рассмотрим другой пример. Но сначала приведем еще раз запрос из представленного

ранее примера применения оператора join.

var inStockList = from item in items

join entry in statusList

on item.ItemNumber equals entry.ItemNumber

select new Temp(item.Name, entry.InStock);

По этому запросу формируется последовательность, состоящая из объектов, инкапсулирующих

наименование товара и состояние его запасов на складе. Вся эта информация

получается путем объединения двух источников данных: items и statusList.

Ниже приведен переделанный вариант данного запроса, в котором вместо синтаксиса,

предусмотренного в C# для запросов, используется метод запроса Join().

// Использовать метод запроса Join() для составления списка

// наименований товаров и состояния их запасов на складе.

var inStockList = items.Join(statusList,

k1 => k1.ItemNumber,

k2 => k2.ItemNumber,

(k1, k2) => new Temp(k1.Name, k2.InStock));

В данном варианте именованный класс Temp используется для хранения результирующего

объекта, но вместо него можно воспользоваться анонимным типом.

**Синтаксис запросов и методы запроса**

Как пояснялось в предыдущем разделе, запросы в C# можно формировать двумя

способами, используя синтаксис запросов или методы запроса. Любопытно, что оба

способа связаны друг с другом более тесно, чем кажется, глядя на исходный код программы.

Дело в том, что синтаксис запросов компилируется в вызовы методов запроса.

Поэтому код

where х< 10

будет преобразован компилятором в следующий вызов.

Where(х => х < 10)

Таким образом, оба способа формирования запросов в конечном итоге сходятся на

одном и том же.

Но если оба способа оказываются в конечном счете равноценными, то какой из них

лучше для программирования на С#? В целом, рекомендуется чаще пользоваться синтаксисом

запросов, поскольку он полностью интегрирован в язык С#, поддерживается

соответствующими ключевыми словами и синтаксическим конструкциями.

**Дополнительные методы расширения, связанные с запросами**

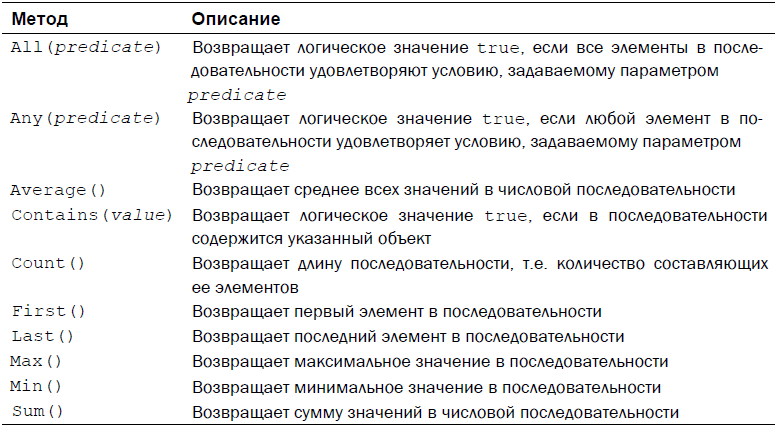
Помимо методов, соответствующих операторам запроса, поддерживаемым в С#, имеется

ряд других методов расширения, связанных с запросами и зачастую оказывающих

помощь в формировании запросов. Ниже приведены

наиболее часто используемые методы расширения, связанные с запросами. Многие

из них могут перегружаться, поэтому они представлены лишь в самой общей форме.



Метод Count() уже демонстрировался ранее в этой главе. А в следующей программе

демонстрируются остальные методы расширения, связанные с запросами.

(***glava19\_18***)

class ExtMethods

{

static void Main()

{

int[] nums = { 3, 1, 2, 5, 4 };

Console.WriteLine("Min value: " + nums.Min());

Console.WriteLine("Max value: " + nums.Max());

Console.WriteLine("First value: " + nums.First());

Console.WriteLine("Last value: " + nums.Last());

Console.WriteLine("Sum value: " + nums.Sum());

Console.WriteLine("Average value: " + nums.Average());

if (nums.All(n => n > 0))

Console.WriteLine("All values more than 0.");

if (nums.All(n => (n % 2) == 0))

Console.WriteLine("At least one value is even.");

if (nums.Contains(3))

Console.WriteLine("Array contains value 3.");

}

}

Методы расширения, связанные с запросами, можно также использовать в самом

запросе, основываясь на синтаксисе запросов, предусмотренном в С#. И в действительности

это делается очень часто. Например, метод Average() используется в приведенной

ниже программе для получения последовательности, состоящей только из тех

значений, которые оказываются меньше среднего всех значений в массиве.

(***glava19\_19***)

class ExtMethods2

{

static void Main()

{

int[] nums = { 1, 2, 3, 8, 6, 9, 10, 3, 6, 7 };

var ltAvg = from n in nums

let x = nums.Average()

where n < x

select n;

Console.WriteLine("Average: " + nums.Average());

Console.Write("Values less then avg: ");

foreach (int i in ltAvg) Console.Write(i + " ");

Console.WriteLine();

}

}

Как видите, переменной x в операторе let присваивается среднее всех значений

в массиве nums. Это значение получается в результате вызова метода Average() для

массива nums.

**Режимы выполнения запросов: отложенный и немедленный**

В LINQ запросы выполняются в двух разных режимах: немедленном и отложенном.

Как пояснялось ранее в этой главе, при формировании запроса определяется ряд правил,

которые не выполняются вплоть до оператора цикла foreach. Это так называемое

*отложенное выполнение.*

Но если используются методы расширения, дающие результат, отличающийся от

последовательности, то запрос должен быть выполнен для получения этого результата.

Рассмотрим, например, метод расширения Count(). Для того чтобы этот метод

возвратил количество элементов в последовательности, необходимо выполнить запрос,

и это делается автоматически при вызове метода Count(). В этом случае имеет

место *немедленное выполнение,* когда запрос выполняется автоматически для получения

требуемого результата. Таким образом, запрос все равно выполняется, даже если он не

используется явно в цикле foreach.

Ниже приведен простой пример программы для получения количества положительных

элементов, содержащихся в последовательности.

(***glava19\_20***)

class ImmediateExec

{

static void Main()

{

int[] nums = { 1, -2, 3, 0, -4, 5 };

int len = (from n in nums

where n > 0

select n).Count();

Console.WriteLine("Positive values in nums: " + len);

}

}

Обратите внимание на то, что цикл foreach не указан в данной программе явным

образом. Вместо этого запрос выполняется автоматически благодаря вызову метода

расширения Count().

Любопытно, что запрос из приведенной выше программы можно было бы сформировать

и следующим образом.

var posNums = from n in nums

where n > 0

select n;

int len = posNums.Count(); // запрос выполняется здесь

В данном случае метод Count() вызывается для переменной запроса. И в этот момент

запрос выполняется для получения подсчитанного количества.

К числу других методов расширения, вызывающих немедленное выполнение запроса,

относятся методы ТоАrray() и ToList(). Оба этих метода расширения определены

в классе Enumerable. Метод ToArray() возвращает результаты запроса в массиве,

а метод ToList() — результаты запроса в форме коллекции List.

**Деревья выражений**

Еще одним средством, связанным с LINQ, является *дерево выражений,* которое представляет

лямбда-выражение в виде данных. Это означает, что само лямбда-выражение

нельзя выполнить, но можно преобразовать в исполняемую форму. Деревья выражений

инкапсулируются в классе System.Linq.Expressions.Expression<TDelegate>.

Они оказываются пригодными в тех случаях, когда запрос выполняется вне программы,

например средствами SQL в базе данных. Если запрос представлен в виде данных,

то его можно преобразовать в формат, понятный для базы данных. Этот процесс выполняется,

например, средствами LINQ to SQL в интегрированной среде разработки

Visual Studio. Таким образом, деревья выражений способствуют поддержке в C# различных

баз данных.

Для получения исполняемой формы дерева выражений достаточно вызвать метод

Compile(), определенный в классе Expression. Этот метод возвращает ссылку, которая

может быть присвоена делегату для последующего выполнения. А тип делегата

может быть объявлен собственным или же одним из предопределенных типов делегата

Func в пространстве имен System. Две формы делегата Func уже упоминались

ранее при рассмотрении методов запроса, но существует и другие его формы.

Деревьям выражений присуще следующее существенное ограничение: они могут

представлять только одиночные лямбда-выражения. С их помощью нельзя представить

блочные лямбда-выражения.

Ниже приведен пример программы, демонстрирующий конкретное применение

дерева выражений. В этой программе сначала создается дерево выражений, данные

которого представляют метод, определяющий, является ли одно целое число множителем

другого. Затем это дерево выражений компилируется в исполняемый код. И наконец,

в этой программе демонстрируется выполнение скомпилированного кода.

(***glava19\_21***)

using System.Linq.Expressions;

class SimpleExpTree

{

static void Main()

{

//lambda func as data type

Expression<Func<int, int, bool>>

IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;

//compile expressions in code

Func<int, int, bool> IsFactor = IsFactorExp.Compile();

//use expr

if (IsFactor(10, 5))

Console.WriteLine("Number 5 is multiplier of 10.");

if (!IsFactor(10, 7))

Console.WriteLine("Number 7 is not multiplier of 10.");

Console.WriteLine();

}

}

Данный пример программы наглядно показывает два основных этапа применения

дерева выражений. Сначала в ней создается дерево выражений с помощью следующего

оператора.

Expression<Func<int, int, bool>>

IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;

В этом операторе конструируется представление лямбда-выражения в оперативной

памяти. Как пояснялось выше, это представление доступно по ссылке, присваиваемой

делегату IsFactorExp. А в следующем операторе данные выражения преобразуются

в исполняемый код.

Func<int, int, bool> IsFactor = IsFactorExp.Compile();

Обратите также внимание на то, что <Func<int, int, bool> обозначает тип

делегата. В этой форме делегата Func указываются два параметра типа int и возвращаемый

тип bool. В рассматриваемой здесь программе использована именно эта

форма делегата Func, совместимая с лямбда-выражениями, поскольку для выражения

требуются два параметра. Для других лямбда-выражений могут подойти иные

формы делегата Func в зависимости от количества требуемых параметров. Вообще

говоря, конкретная форма делегата Func должна удовлетворять требованиям лямбда-

выражения.

**Методы расширения**

Как упоминалось выше, методы расширения предоставляют средства для расширения

функций класса, не прибегая к обычному механизму наследования. Методы расширения

создаются нечасто, поскольку механизм наследования, как правило, предлагает

лучшее решение. Тем не менее знать, как они действуют, никогда не помешает.

Ведь они имеют существенное значение для LINQ.

Метод расширения является статическим и поэтому должен быть включен в состав

статического, необобщенного класса. Тип первого параметра метода расширения

определяет тип объектов, для которых этот метод может быть вызван. Кроме того,

первый параметр может быть указан с модификатором this. Объект, для которого

вызывается метод расширения, автоматически передается его первому параметру. Он

не передается явным образом в списке аргументов. Следует, однако, иметь в виду, что

метод расширения может по-прежнему вызываться для объекта аналогично методу

экземпляра, несмотря на то, что он объявляется как статический.

Ниже приведена общая форма метода расширения.

static возращаемый\_тип имя(this тип\_вызывающего\_объекта ob, список\_параметров)

Очевидно, что *список\_параметров* окажется пустым в отсутствие аргументов, за

исключением аргумента, неявно передаваемого вызывающим объектом *ob.* Не следует,

однако, забывать, что первым параметром метода расширения является автоматически

передаваемый объект, для которого вызывается этот метод. Как правило, метод

расширения становится открытым членом своего класса.

В приведенном ниже примере программы создаются три простых метода

расширения.

(***glava19\_22***)

static class MyExtMeths

{

//return back of number

public static double Reciprocal(this double v)

{

return 1.0 / v;

}

//change to another register and return result

public static string RevCase(this string str)

{

string temp = "";

foreach (char ch in str)

{

if (Char.IsLower(ch)) temp += Char.ToUpper(ch, CultureInfo.CurrentCulture);

else temp += Char.ToLower(ch, CultureInfo.CurrentCulture);

}

return temp;

}

//return absolute value n / d

public static double AbsDivdeBy(this double n, double d)

{

return Math.Abs(n / d);

}

}

class ExtDemo

{

static void Main()

{

double val = 8.0;

string str = "Alpha Beta Gamma";

//method reciprocal

Console.WriteLine("Reversed num {0} = {1}", val, val.Reciprocal());

//method revcase

Console.WriteLine(str + " after change register: " + str.RevCase());

//method absdivideby

Console.WriteLine("Result of method AbsDivideBy(-2): " + val.AbsDivdeBy(-2));

}

}

В данном примере программы каждый метод расширения содержится в статическом

классе MyExtMeths. Как пояснялось выше, метод расширения должен быть

объявлен в статическом классе. Более того, этот класс должен находиться в области

действия своих методов расширения, чтобы ими можно было пользоваться.

Объявленные методы расширения вызываются для объекта таким же образом, как

и методы экземпляра. Главное отличие заключается в том, что вызывающий объект

передается первому параметру метода расширения.