Linq

Ссылка для изучения: <https://msdn.microsoft.com/ru-ru/library/bb397933.aspx>

# Вступление

Запрос представляет собой выражение, извлекающее данные из источника данных. Запросы обычно выражаются на специальном языке запросов. Со временем были разработаны различные языки для различных типов источников данных, например SQL для реляционных баз данных и XQuery для XML. Таким образом, разработчики вынуждены изучать новый язык запросов для каждого типа источника данных или формата данных, который они должны поддерживать. LINQ упрощает ситуацию, предлагая единообразную модель для работы с данными в различных видах источников и форматов данных. В запросе LINQ работа всегда осуществляется с объектами. Для запросов и преобразований данных в XML-документах, базах данных SQL, наборах данных ADO.NET, коллекциях .NET и любых других форматах, для которых доступен поставщик LINQ, используются одинаковые базовые шаблоны кодирования.

Все операции запроса LINQ состоят из трех различных действий.

1. Получение источника данных.
2. Создание запроса.
3. Выполнение запроса.

class IntroToLINQ

{

static void Main()

{

// The Three Parts of a LINQ Query:

// 1. Data source.

int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Query creation.

// numQuery is an IEnumerable<int>

var numQuery =

from num in numbers

where (num % 2) == 0

select num;

// 3. Query execution.

foreach (int num in numQuery)

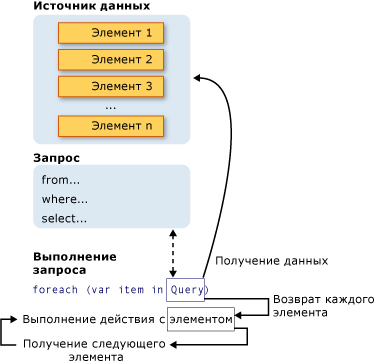
{

Console.Write("{0,1} ", num);

}

}

}



# Источник данных

В предыдущем примере источником данных является массив, поэтому он неявно поддерживает универсальный интерфейс IEnumerable<T>. Это значит, что к нему можно выполнять запросы с LINQ. Запрос выполняется в операторе foreach, и для foreach требуется IEnumerable или IEnumerable<T>. Типы, поддерживающие IEnumerable<T> или производные интерфейсы, такие как универсальный IQueryable<T>, называются запрашиваемыми типами.

# Запрос

Запрос указывает, какую информацию нужно извлечь из источника или источников данных. При необходимости, запрос также указывает способ сортировки, группировки и формирования этих сведений перед возвращением. Запрос хранится в переменной запроса и инициализируется выражением запроса. Чтобы упростить написание запросов, в C# появился новый синтаксис запроса.

Запрос из предыдущего примера возвращает все четные числа из массива целых чисел. Выражение запроса содержит три предложения: from, where и select. (Если вы знакомы с SQL, обратите внимание, что порядок предложений противоположен порядку в SQL.) Предложение from указывает источник данных, предложение where применяет фильтр, а предложение select указывает тип возвращаемых элементов. Эти и другие предложения запросов подробно описаны в разделе Выражения запросов LINQ. Важно, что в LINQ сама переменная запроса не предпринимает действий и не возвращает никаких данных. Она просто хранит сведения, необходимые для предоставления результатов при последующем выполнении запроса.

# Выполнение запроса

### Отложенное выполнение

Как уже говорилось ранее, сама переменная запроса только хранит команды запроса. Фактическое выполнение запроса откладывается до выполнения итерации переменной запроса в операторе foreach. Эту концепцию называют отложенным выполнением, она показана в следующем примере.

// Query execution.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

### Принудительное немедленное выполнение

Запросы, выполняющие статистические функции над диапазоном исходных элементов, должны сначала выполнить итерацию этих элементов. Примерами таких запросов являются Count, Max, Average и First. Они выполняются без явного оператора foreach, поскольку сам запрос должен использовать foreach для возвращения результата. Обратите внимание, что такой тип запросов возвращает одиночное значение, а не коллекцию IEnumerable. Следующий запрос возвращает количество четных чисел в исходном массиве.

var evenNumQuery =

from num in numbers

where (num % 2) == 0

select num;

int evenNumCount = evenNumQuery.Count();

Чтобы принудительно вызвать немедленное выполнение любого запроса и кэшировать его результаты, можно вызвать метод ToList<TSource> или ToArray<TSource>.

List<int> numQuery2 =

(from num in numbers

where (num % 2) == 0

select num).ToList();

// or like this:

// numQuery3 is still an int[]

var numQuery3 =

(from num in numbers

where (num % 2) == 0

select num).ToArray();

# Основы выражения запроса

##### Получение источника данных

//queryAllCustomers is an IEnumerable<Customer>

var queryAllCustomers = from cust in customers

select cust;

##### Фильтрация

var queryLondonCustomers = from cust in customers

where cust.City == "London"

select cust;

where cust.City=="London" && cust.Name == "Devon"

##### Порядок

Часто целесообразно отсортировать возвращенные данные. Предложение orderby сортирует элементы возвращаемой последовательности в зависимости от компаратора по умолчанию для сортируемого типа. Например, следующий запрос может быть расширен для сортировки результатов на основе свойства Name. Поскольку Name является строкой, сравнение по умолчанию выполняется в алфавитном порядке от А до Я.

var queryLondonCustomers3 =

from cust in customers

where cust.City == "London"

orderby cust.Name ascending

select cust;

##### Группировка

Предложение group позволяет группировать результаты на основе указанного ключа. Например, можно указать, что результаты должны быть сгруппированы по City так, чтобы все заказчики из Лондона или Парижа оказались в отдельных группах. В этом случае ключом является cust.City.

// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>

var queryCustomersByCity =

from cust in customers

group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>

foreach (var customerGroup in queryCustomersByCity)

{

Console.WriteLine(customerGroup.Key);

foreach (Customer customer in customerGroup)

{

Console.WriteLine(" {0}", customer.Name);

}

}

Когда запрос завершается предложением group, результаты представляются в виде списка из списков. Каждый элемент в списке является объектом, имеющим член Key и список элементов, сгруппированных по этому ключу. При итерации запроса, создающего последовательность групп, необходимо использовать вложенный цикл foreach. Внешний цикл выполняет итерацию каждой группы, а внутренний цикл — итерацию членов каждой группы.

Если необходимо ссылаться на результаты операции группировки, можно использовать ключевое слово into для создания идентификатора, который можно будет запрашивать. Следующий запрос возвращает только те группы, которые содержат более двух заказчиков.

// custQuery is an IEnumerable<IGrouping<string, Customer>>

var custQuery =

from cust in customers

group cust by cust.City into custGroup

where custGroup.Count() > 2

orderby custGroup.Key

select custGroup;

##### Соединение

var innerJoinQuery =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorName = dist.Name };

В LINQ нет необходимости использовать join так часто, как в SQL, поскольку внешние ключи в LINQ представлены в объектной модели свойствами, содержащими коллекцию элементов. Например, объект Customer содержит коллекцию объектов Order. Вместо выполнения соединения, доступ к заказам можно получить с помощью точечной нотации.

from order in Customer.Orders...

# Пример

public class Student

{

public string First { get; set; }

public string Last { get; set; }

public int ID { get; set; }

public List<int> Scores;

}

// Create a data source by using a collection initializer.

static List<Student> students = new List<Student>

{

new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},

new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},

new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},

new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},

new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},

new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},

new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},

new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},

new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},

new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},

new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},

new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91} }

};

Создание запроса:

// Create the query.

// The first line could also be written as "var studentQuery ="

IEnumerable<Student> studentQuery =

from student in students

where student.Scores[0] > 90

select student;

Выполнение:

// Execute the query.

// var could be used here also.

foreach (Student student in studentQuery)

{

Console.WriteLine("{0}, {1}", student.Last, student.First);

}

// Output:

// Omelchenko, Svetlana

// Garcia, Cesar

// Fakhouri, Fadi

// Feng, Hanying

// Garcia, Hugo

// Adams, Terry

// Zabokritski, Eugene

// Tucker, Michael

Изменение запроса:

where student.Scores[0] > 90 && student.Scores[3] < 80

orderby student.Last ascending

orderby student.Scores[0] descending

Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);

Группировка:

// studentQuery2 is an IEnumerable<IGrouping<char, Student>>

var studentQuery2 =

from student in students

group student by student.Last[0];

// studentGroup is a IGrouping<char, Student>

foreach (var studentGroup in studentQuery2)

{

Console.WriteLine(studentGroup.Key);

foreach (Student student in studentGroup)

{

Console.WriteLine(" {0}, {1}",

student.Last, student.First);

}

}

// Output:

// O

// Omelchenko, Svetlana

// O'Donnell, Claire

// M

// Mortensen, Sven

// G

// Garcia, Cesar

// Garcia, Debra

// Garcia, Hugo

// F

// Fakhouri, Fadi

// Feng, Hanying

// T

// Tucker, Lance

// Tucker, Michael

// A

// Adams, Terry

// Z

// Zabokritski, Eugene

Упорядочивание по значению ключа:

var studentQuery4 =

from student in students

group student by student.Last[0] into studentGroup

orderby studentGroup.Key

select studentGroup;

foreach (var groupOfStudents in studentQuery4)

{

Console.WriteLine(groupOfStudents.Key);

foreach (var student in groupOfStudents)

{

Console.WriteLine(" {0}, {1}",

student.Last, student.First);

}

}

// Output:

//A

// Adams, Terry

//F

// Fakhouri, Fadi

// Feng, Hanying

//G

// Garcia, Cesar

// Garcia, Debra

// Garcia, Hugo

//M

// Mortensen, Sven

//O

// Omelchenko, Svetlana

// O'Donnell, Claire

//T

// Tucker, Lance

// Tucker, Michael

//Z

// Zabokritski, Eugene

Идентификаторы в запросе:

// studentQuery5 is an IEnumerable<string>

// This query returns those students whose

// first test score was higher than their

// average score.

var studentQuery5 =

from student in students

let totalScore = student.Scores[0] + student.Scores[1] +

student.Scores[2] + student.Scores[3]

where totalScore / 4 < student.Scores[0]

select student.Last + " " + student.First;

foreach (string s in studentQuery5)

{

Console.WriteLine(s);

}

// Output:

// Omelchenko Svetlana

// O'Donnell Claire

// Mortensen Sven

// Garcia Cesar

// Fakhouri Fadi

// Feng Hanying

// Garcia Hugo

// Adams Terry

// Zabokritski Eugene

// Tucker Michael