Язык интегрированных запросов LINQ

LINQ

- LINQ (Microsoft **L**anguage **IN**tegrated **Q**uery) – предназначен для поддержки запросов к данным всех типов на уровне языка. Эти типы включают массивы и коллекции в памяти, базы данных, документы XML и многое другое.
- LINQ ориентирован на запросы, который могут возвращать набор объектов, единственный объект или множество полей из объекта или набора объектов.
- В LINQ возвращенный набор называется *последовательностью.*
- Большинство последовательностей в LINQ имеют тип **IEnumerable<T>**, где T тип данных объектов- элементов последовательности.
- LINQ позволяет выполнять не только запросы, но и итерации по данным, совместно с их обработкой.

Отложенное выполнение

- Многие запросы LINQ выполняются не в момент конструирования выражения LINQ а в момент обращения к результатам запроса, т.е. при их перечислении.

Способы записи запросов LINQ

- Стандартная точечная нотация C# с вызовом методов на объектах и классах
- Выражения запросов – SQL подобный синтаксис – расширение языка C#

Грамматика выражений запросов

1. Выражение начинается с конструкции from.
2. Далее – ноль или более конструкций from, let или where
   1. **from** — это генератор, который объявляет переменную(ые) диапазона, перечисляющих последовательность или соединение нескольких последовательностей.
   2. **let** представляет переменную диапазона и присваивает ей значение.
   3. **where** фильтрует элементы из входной последовательности или соединения нескольких входных последовательностей в выходную последовательность.
3. Далее может идти конструкция **orderby**, содержащая поле (поля) сортировки с необязательным направлением упорядочивания. Направлением может быть **ascending** (по возрастанию) или **descending** (по убыванию).
4. Далее могут идти конструкция **select** или **group**.
5. Наконец, в оставшейся части может следовать конструкция продолжения: либо **into**, ноль или более конструкций **join**, или же другая повторяющаяся последовательность перечисленных элементов, начиная с конструкций из правила 2. Конструкция into направляет результаты запроса в воображаемую выходную последовательность, которая служит конструкцией from для последующих выражения запросов, начиная с конструкций из правила 2.

LINQ to Objects

IEnumerable<T>

- IEnumerable<T> — это интерфейс, реализуемый всеми классами обобщенных коллекций C#, как это делают массивы. Этот интерфейс позволяет выполнять перечисление элементов коллекций.
  Единственный его метод GetEnumerator возвращает перечислитель Ienumerator<T>(итератор), выполняющий перебор элементов в коллекции.
- Последовательность — это термин для обозначения коллекции, реализующей интерфейс IEnumerable<T> .
- Например, IEnumerable<string> означает последовательность строк.

Расширяющие методы

- Большинство стандартных операций запросов представляют собой расширяющие методы в статическом классе System.Linq.Enumerable, прототипированые с IEnumerable<T> в качестве первого аргумента.
- Поскольку они являются расширяющими методами, предпочтительно вызывать их на переменной типа IEnumerable<T> , что позволяет синтаксис расширяющих методов, а не передавать переменную типа IEnumerable<T> в первом аргументе.
- Методы стандартных операций запросов класса System.Linq.Enumerable , не являющиеся расширяющими методами — это просто статические методы, которые должны быть вызваны на классе System.Linq.Enumerable . Комбинация этих методов стандартных операций запросов дает возможность выполнять сложные запросы данных на последовательности IEnumerable<T>
- Чтобы получить доступ к стандартным операциям запросов, необходимо иметь в коде директиву **using System.Linq;**

Демонстрация отложенного выполнения

```
int[] intArray = new int[] { 1, 2, 3 };
IEnumerable<int> ints = intArray.Select(i => i);
// Отобразить результаты.
foreach (int i in ints)
    Console.WriteLine(i);
// Изменить элемент в источнике данных.
intArray[0] = 5;
Console.WriteLine("---------");
// Снова отобразить результат.
foreach (int i in ints)
    Console.WriteLine(i);
```

- Запрос вернул объект ints, который при перечислении (foreach) выполнил запрос и получает последовательность по одному элементу за раз.

Немедленное выполнение

- Для немедленного выполнения запроса следует использовать операций преобразования **ToArray , ToList , ToDictionary** или **ToLookup,** которые возвращают не IEnumerable<T> , а различные структуры данных с кэшированными результатами, не изменяющимися с изменением источника данных.

```
// Создать массив целых чисел.
int[] intArray = new int[] { 1, 2, 3 };
List<int> ints = intArray.Select(i => i).ToList();
// Отобразить результаты.
foreach (int i in ints) Console.WriteLine(i);
// Изменить элемент в исходном массиве.
intArray[0] = 5;
Console.WriteLine("---------");
// Снова отобразить результаты.
foreach (int i in ints) Console.WriteLine(i);
```

Делегаты Func

- Некоторые стандартные операции запросов прототипированы на прием делегата Func в качестве аргумента. Это предотвращает явное объявление типов делегатов.
- Ниже приведены объявления делегата Func :
  - public delegate TR Func<TR>();
  - public delegate TR Func<T0, TR>(T0 a0);
  - public delegate TR Func<T0, T1, TR>(T0 a0, T1 a1);
  - public delegate TR Func<T0, T1, T2, TR>(T0 a0, T1 a1, T2 a2);
  - public delegate TR Func<T0, T1, T2, T3, TR>(T0 a0, T1 a1, T2 a2, T3 a3);
- Здесь TR ссылается на возвращаемый тип данных. Другие параметры типа — T0 , T1 , T2 и T3 — ссылаются на входные параметры, переданные методу.
- Объявлений несколько, потому что некоторые стандартные операции запросов имеют аргументы-делегаты, требующие больше параметров, чем другие.

Один из прототипов операции Where

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

- Отсюда видно, что метод-предикат или лямбда-выражение должны принимать один аргумент — параметр T и возвращать bool.

```
// Создать массив целых чисел.
int[] ints = new int[] { 1, 2, 3, 4, 5, 6 };
// Объявление делегата.
Func<int, bool> GreaterThanTwo = i => i > 2;
IEnumerable<int> intsGreaterThanTwo =
ints.Where(GreaterThanTwo);
// Отобразить результаты.
foreach (int i in intsGreaterThanTwo) Console.WriteLine(i);
```

LINQ to Object
Отложенные операции

Необходимые для работы

- Пространства имён
  - using System.Linq;
  - using System.Collections;
  - using System.Collections.Generic;
  - using System.Data.Linq;
- Сборки
  - System.Data.Linq.dll

Общие для всех примеров классы

```
public class Employee
{
    public int id;
    public string name;
    public string surname;

    public static List<Employee> GetEmployeesList()
    {
        var list = new List<Employee>()
        {
            new Employee { id = 1, name = "Joe", surname = "Rattz" },
            new Employee { id = 2, name = "William", surname = "Gates" },
            new Employee { id = 3, name = "Anders", surname = "Hejlsberg" },
            new Employee { id = 4, name = "David", surname = "Lightman" },
            new Employee { id = 101, name = "Kevin", surname = "Flynn" }
        };
        return list;
    }

    public static Employee[] GetEmployeesArray()
    {
        return GetEmployeesList().ToArray();
    }
}
```

```
class EmployeeOptionEntry
{
    public int id;
    public long optionsCount;
    public DateTime dateAwarded;
    public static EmployeeOptionEntry[] GetEmployeeOptionEntries()
    {
        var empOptions = new EmployeeOptionEntry[] {
            new EmployeeOptionEntry {
            id = 1, optionsCount = 2,
            dateAwarded = DateTime.Parse("1999/12/31") },
            new EmployeeOptionEntry {
            id = 2, optionsCount = 10000,
            dateAwarded = DateTime.Parse("1992/06/30") },
            new EmployeeOptionEntry {
            id = 2, optionsCount = 10000,
            dateAwarded = DateTime.Parse("1994/01/01") },
            new EmployeeOptionEntry {
            id = 3, optionsCount = 5000,
            dateAwarded = DateTime.Parse("1997/09/30") },
            new EmployeeOptionEntry {
            id = 2, optionsCount = 10000,
            dateAwarded = DateTime.Parse("2003/04/01") },
            ...
};
        return empOptions;
    }
}
```

Ограничение

- Операции ограничения (restriction) используются для включения или исключения элементов из входной последовательности.
- Операция **Where** используется для фильтрации элементов в последовательность.
- Первый прототип Where:

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

- Этот прототип Where принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает true .

- Второй прототип Where

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

- Отличие второго прототип Where от первого: делегат метода-предиката принимает дополнительный целочисленный аргумент.
- Этот аргумент будет индексом элемента во входной последовательности.
- Нумерация индекса начинается с нуля, поэтому индексом первого элемента будет 0.

Проекция

- Операции проекции возвращают выходную последовательность элементов, которая сгенерирована за счет выбора элементов или путем создания совершенно новых элементов, содержащих части элементов из входной последовательности.
- Тип данных элементов выходной последовательности может отличаться от типа элементов входной последовательности.

Select. Первый прототип

- Операция **Select** используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа.
- Первый прототип Select

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

- При вызове Select делегат метода-селектора передается в аргументе selector. Метод-селектор должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа S .
- Операция Select вызовет метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Метод-селектор выберет интересующую часть входного элемента, создаст новый элемент — возможно, другого типа (даже анонимного) — и вернет его.

Select. Второй прототип

- Второй прототип Select

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, int, S> selector);
```

- В этом прототипе операции Select методу-селектору передается дополнительный целочисленный параметр. Это индекс, начинающийся с нуля, входного элемента во входной последовательности.

SelectMany. Первый прототип

- Операция SelectMany используется для создания выходной последовательности с проекцией “один ко многим” из входной последовательности.
- В то время как операция Select возвращает один выходной элемент для каждого входного элемента, SelectMany вернет ноль или более выходных элементов для каждого входного.
- Первый прототип SelectMany

```
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<S>> selector);
```

- Этот прототип операции возвращает объект, который проходит по входной последовательности, получая каждый элемент индивидуально из входной последовательности и передавая его в метод-селектор. Последний затем возвращает объект, который во время перечисления выдает ноль или более элементов типа S в промежуточную выходную последовательность.
- Операция SelectMany вернет конкатенированную выходную последовательность при каждом вызове метода-селектора.

Пример для SelectMany

```
Employee[] employees = Employee.GetEmployeesArray();
            EmployeeOptionEntry[] empOptions =
EmployeeOptionEntry.GetEmployeeOptionEntries();
            var employeeOptions =
                employees.SelectMany(e => empOptions
                    .Where(eo => eo.id == e.id)
                    .Select(eo =>
                        new { id = eo.id, optionsCount = eo.optionsCount }));
            foreach (var item in employeeOptions)
                Console.WriteLine(item);
```