Введение в LINQ

Что такое LINQ?

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Основная часть функциональности LINQ сосредоточена в пространстве имен **System.LINQ**. В проектах под .NET 6 (и выше) данное пространство имен подключается по умолчанию.

Например, если у вас есть массив чисел, вы можете использовать LINQ для выбора всех чисел, которые больше 80, следующим образом:

```
int[] scores = { 97, 92, 81, 60 };
IEnumerable<int> scoreQuery = scores.Where(x => x > 80);
foreach(var i in scoreQuery) Console.Write(i + " ");
// Output: 97 92 81
```

Разновидности LINQ

1) **LINQ to Objects**: применяется для работы с массивами и коллекциями
2) **LINQ to Entities**: используется при обращении к базам данных через технологию Entity Framework
3) **LINQ to XML**: применяется при работе с файлами XML
4) **LINQ to DataSet**: применяется при работе с объектом DataSet
5) **Parallel LINQ (PLINQ)**: используется для выполнения параллельных запросов

Для работы с коллекциями можно использовать два способа:

1) Операторы запросов LINQ
2) Методы расширений LINQ

Операторы запросов LINQ

Операторы запросов LINQ - это специальные операторы, которые позволяют создавать запросы к данным в C# с помощью синтаксиса, похожего на SQL. Они позволяют выполнять различные операции с данными, такие как выборка, фильтрация, сортировка, группировка и объединение. Операторы запросов LINQ могут быть использованы для работы с различными типами данных, такими как массивы, коллекции, базы данных и XML-документы.

Например, оператор where используется для фильтрации данных, а оператор select - для выборки определенных полей данных.

```
List<Person> people = new List<Person>()
{
    new Person { Name = "John", Age = 25 },
    new Person { Name = "Jane", Age = 20 },
    new Person { Name = "Bob", Age = 30 }
};

IEnumerable<string> query = from x in people
                            where x.Age > 21
                            select x.Name;

foreach (var name in query) Console.Write(name + " ");
// Output: John Bob
```

```
public class Person
{
    0 references
    public string Name { get; set; } = default!;
    0 references
    public int Age { get; set; }
}
```

Методы расширения LINQ

Методы расширения LINQ - это методы, которые расширяют функциональность классов, реализующих интерфейс IEnumerable<T>. Они позволяют выполнять запросы к данным, используя цепочки методов, которые могут быть более читаемыми и эффективными для выполнения сложных запросов.

```
IEnumerable<string> query = people.Where(x => x.Age > 21).Select(x => x.Name);
```

Проекция Данных

Проекция Данных

Проекция позволяет преобразовать объект одного типа в объект другого типа. Для проекции используется метод **Select.**

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

Примеры использования метода Select

```
List<Person> people = new List<Person>()
{
    new Person { Name = "John", Age = 25 },
    new Person { Name = "Jane", Age = 20 },
    new Person { Name = "Bob", Age = 30 }
};

var names = people.Select(x => x.Name);

foreach (string n in names)
    Console.Write(n + " ");
// Output: John Jane Bob
```

```
var query = people.Select(p => new
{
    FirstName = p.Name,
    Year = DateTime.Now.Year - p.Age,
});

foreach (var i in query)
    Console.WriteLine($"{i.FirstName} - {i.Year}");
// Output:
// John - 1998
// Jane - 2003
// Bob - 1993
```

Метод SelectMany

Метод **SelectMany** позволяет свести набор коллекций в одну коллекцию.

```
var companies = new List<Company>
{
    new Company("Microsoft", new List<Person> {new Person("Tom"), new Person("Bob")}),
    new Company("Google", new List<Person> {new Person("Sam"), new Person("Mike")}),
};

IEnumerable<Person>? employees = companies.SelectMany(c => c.Staff);

foreach (var i in employees)
    Console.Write($"{i.Name} ");
// Output: Tom Bob Sam Mike
```

```
3 references
record class Company(string Name, List<Person> Staff);
8 references
record class Person(string Name);
```

Фильтрация Коллекций

Фильтрация коллекции

Для выбора элементов из некоторого набора по условию используется метод **Where**. Этот метод принимает делегат **Func<TSource,bool>**, который в качестве параметра принимает каждый элемент последовательности и возвращает значение bool. Если элемент соответствует некоторому условию, то возвращается true, и тогда этот элемент передается в коллекцию, которая возвращается из метода Where.

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

Пример использования метода Where

```
List<Person> people = new List<Person>()
{
    new Person { Name = "John", Age = 25 },
    new Person { Name = "Jane", Age = 20 },
    new Person { Name = "Bob", Age = 30 }
};

IEnumerable<Person> query = people.Where(p => p.Age == 20 || p.Name == "John");

foreach (var i in query)
    Console.Write($"{i.Name} ");
// Output: John Jane
```

Сортировка

Метод OrderBy

Для сортировки набора данных в LINQ можно применять метод **OrderBy**. Первая версия метода получает делегат, который через параметр получает элемент коллекции и который возвращает значение, применяемое для сортировки. Вторая версия позволяет также задать принцип сортировки через реализацию интерфейса **IComparer**.

```
OrderBy (Func<TSource,TKey> keySelector)
OrderBy (Func<TSource,TKey> keySelector, IComparer<TKey>? comparer);
```

```
int[] numbers = { 3, 12, 4, 10 };
var orderedNumbers = numbers.OrderBy(n => n);
foreach (int i in orderedNumbers)
    Console.WriteLine(i);
// Output: 3 4 10 12
```

Для сортировки по убыванию можно применять метод **`OrderByDescending()`**, который работает аналогично `OrderBy` за исключением направления сортировки:

```
int[] numbers = { 3, 12, 4, 10 };
var orderedNumbers = numbers.OrderByDescending(n => n);
foreach (int i in orderedNumbers)
    Console.WriteLine(i);
// Output: 12 10 4 3
```

Множественные критерии сортировки

В наборах сложных объектов иногда встает ситуация, когда надо отсортировать не по одному, а сразу по нескольким полям. С помощью методов расширения то же самое можно сделать через метод **ThenBy()**(для сортировки по возрастанию) и **ThenByDescending()** (для сортировки по убыванию)

```
var people = new List<Person>
{
    new Person("Tom", 37),
    new Person("Sam", 28),
    new Person("Tom", 22),
    new Person("Bob", 41),
};

var query = people.OrderBy(x => x.Name)
                  .ThenByDescending(x => x.Age);
// Bob - 41
// Sam - 28
// Tom - 37
// Tom - 22
```

Объединение, пересечение и разность коллекций

Разность коллекций

С помощью метода **`Except()`** можно получить разность двух коллекций. В данном случае из массива `foo` убираются все элементы, которые есть в массиве `bar`.

```
string[] foo = { "Microsoft", "Google", "Apple"};
string[] bar = { "Apple", "IBM", "Samsung"};

// разность коллекций
var result = foo.Except(bar);

foreach (string s in result)
    Console.WriteLine(s);
// Output: Microsoft Google
```

Пересечение коллекций

Для получения пересечения последовательностей, то есть общих для обоих наборов элементов, применяется метод **Intersect:**

```
string[] foo = { "Microsoft", "Google", "Apple"};
string[] bar = { "Apple", "IBM", "Samsung"};

var result = foo.Intersect(bar);

foreach (string s in result)
    Console.WriteLine(s);
// Output: Apple
```

Удаление дубликатов

Для удаления дубликатов в наборе используется метод **Distinct:**

```
string[] foo = { "Microsoft", "Google", "Apple", "Microsoft", "Google" };

var result = foo.Distinct();

foreach (string s in result)
    Console.WriteLine(s);
// Output: Microsoft Google Apple
```

Объединение коллекций

Для объединения двух последовательностей используется метод **Union**. Его результатом является новый набор, в котором имеются элементы, как из первой, так и из второй последовательности. Повторяющиеся элементы добавляются в результат только один раз:

```
string[] foo = { "Microsoft", "Google", "Apple"};
string[] bar = { "Apple", "IBM", "Samsung"};

var result = foo.Union(bar);

foreach (string s in result)
    Console.WriteLine(s);
// Output: Microsoft Google Apple IBM Samsung
```

Если же нам нужно простое объединение двух наборов, то мы можем использовать метод **Concat**:

```
string[] foo = { "Microsoft", "Google", "Apple"};
string[] bar = { "Apple", "IBM", "Samsung"};

var result = foo.Concat(bar);

foreach (string s in result)
    Console.WriteLine(s);
// Output: Microsoft Google Apple Apple IBM Samsung
```

Методы Skip и Take

Метод Skip

Метод **Skip()** пропускает определенное количество элементов. Количество пропускаемых элементов передается в качестве параметра в метод:

```
string[] people = { "Tom", "Sam", "Bob", "Mike", "Kate" };
// пропускаем первые два элемента
var result1 = people.Skip(2); // "Bob", "Mike", "Kate"
var result2 = people.SkipLast(2); // "Tom", "Sam", "Bob"
```

Метод SkipWhile

Метод **SkipWhile()** пропускает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. В метод передается делегат, который представляет условие, он получает каждый элемент коллекции и возвращает значение `true`, если элемент соответствует условию.

```
//  SkipWhile(Func<TSource, bool> predicate);

string[] people = { "Tom", "Sam", "Bob", "Mike", "Kate" };

// пропускаем первые элементы, длина которых равна 3
var result = people.SkipWhile(p=> p.Length == 3);    // "Mike", "Kate", "Bob"
```

Методы Take и TakeLast

Метод **Take()** извлекает определенное число элементов. Количество извлекаемых элементов передается в метод в качестве параметра. Метод **TakeLast()** извлекает определенное количество элементов с конца коллекции. Например, извлечем три последних элемента:

```
string[] people = { "Tom", "Sam", "Mike", "Kate", "Bob" };
// извлекаем последние 3 элемента
var result = people.TakeLast(3);    // "Mike", "Kate", "Bob"
```

Метод TakeWhile

Метод **TakeWhile()** выбирает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. В метод передается делегат, который представляет условие, он получает каждый элемент коллекции и возвращает значение `true`, если элемент соответствует условию. Например:

```
string[] people = { "Tom", "Sam", "Mike", "Kate", "Bob" };
// извлекаем первые элементы, длина которых равна 3
var result = people.TakeWhile(p=> p.Length == 3);    // "Tom", "Sam"
```

Проверка наличия и получение элементов

Метод All

Метод **All()** проверяет, соответствуют ли все элементы условию. Если все элементы соответствуют условию, то возвращается true.
Например:

```
string[] people = { "Tom", "Tim", "Bob", "Sam" };
// проверяем, все ли элементы имеют длину в 3 символа
bool allHas3Chars = people.All(s => s.Length == 3); // true
Console.WriteLine(allHas3Chars);

// проверяем, все ли строки начинаются на T
bool allStartsWithT = people.All(s => s.StartsWith("T")); // false
Console.WriteLine(allStartsWithT);
```

Метод Any

Метод **Any()** действует подобным образом, только возвращает `true`, если хотя бы один элемент коллекции удовлетворяет определенному условию:

```
string[] people = { "Tom", "Tim", "Bob", "Sam" };

// проверяем, все ли элементы имеют длину больше 3 символов
bool allHasMore3Chars = people.Any(s => s.Length > 3);     // false
Console.WriteLine(allHasMore3Chars);

// проверяем, все ли строки начинаются на T
bool allStartsWithT = people.Any(s => s.StartsWith("T"));   // true
Console.WriteLine(allStartsWithT);
```

Метод Contains

Метод **Contains()** возвращает `true`, если коллекция содержит определенный элемент.

```
string[] people = { "Tom", "Tim", "Bob", "Sam" };

// проверяем, есть ли строка Tom
bool hasTom = people.Contains("Tom");     // true
Console.WriteLine(hasTom);

// проверяем, есть ли строка Mike
bool hasMike = people.Contains("Mike");     // false
Console.WriteLine(hasMike);
```

Метод First

Метод **First()** возвращает первый элемент последовательности. Также в метод First можно передать предикат. В этом случае метод возвращает первый элемент, который соответствует условию

Стоит учитывать, что если коллекция пуста или в коллекции нет элементов, который соответствуют условию, то будет сгенерировано исключение.

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первая строка, длина которой равна 4 символам
var firstWith4Chars = people.First(s=> s.Length == 4);  // Kate
Console.WriteLine(firstWith4Chars);
```

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первая строка, длина которой равна 5 символам
var firstWith5Chars = people.First(s => s.Length == 5);  // ! исключение
Console.WriteLine(firstWith5Chars);

var first = new string[] {}.First();  // ! исключение
Console.WriteLine(first);
```

FirstOrDefault

Метод **FirstOrDefault()** также возвращает первый элемент и также может принимать условие, только если коллекция пуста или в коллекции не окажется элементов, которые соответствуют условию, то метод возвращает значение по умолчанию.

Стоит учитывать, что для коллекций ссылочных типов FirstOrDefault возвращает значение типа T?, то есть значение, которое может быть равно null, а значение по умолчанию - null. Для коллекций числовых типов возвращается непосредственно значение типа T, а значение по умолчанию - 0.

Но мы можем настроить значение по умолчанию, передав его в качестве одного из аргументов.

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первый элемент
var first = people.FirstOrDefault();  // Tom
Console.WriteLine(first);

// первая строка, длина которой равна 4 символам
var firstWith4Chars = people.FirstOrDefault(s => s.Length == 4);  // Kate
Console.WriteLine(firstWith4Chars);

// первый элемент из пустой коллекции
var Default1 = new string[] {}.FirstOrDefault();
Console.WriteLine(Default1);  // null

string? firstWith5Chars = people.FirstOrDefault(s => s.Length == 5, "Undefined");
Console.WriteLine(firstWith5Chars); // Undefined

// первый элемент из пустой коллекции строк
string? Default2 = new string[] {}.FirstOrDefault("hello"); // hello - значение по умолчанию
Console.WriteLine(Default2);  // hello
```

Методы Last и LastOrDefault

Метод **Last()** аналогичен по работе методу First, только возвращает последний элемент. Если коллекция не содержит элемент, который соответствуют условию, или вообще пуста, то метод генерирует исключение. Метод **LastOrDefault()** возвращает последний элемент или значение по умолчанию, если коллекция не содержит элемент, который соответствуют условию, или вообще пуста.