Обзор LINQ to Objects

Отчасти то, что делает LINQ настолько мощным и удобным в применении, заключается в его тесной интеграции с языком C#. Вместо того, чтобы иметь дело с полностью новым набором средств в форме классов, можно применять все те же самые привычные коллекции и массивы с существующими классами. Это значит, что все преимущества запросов LINQ можно получить с минимальными модификациями существующего кода или же вовсе без них. Функциональность LINQ to Objects обеспечивается интерфейсом IEnumerable<T>, последовательностями и стандартными операциями запросов.

Например, если нужно отсортировать массив целых чисел, можно выполнить запрос LINQ для упорядочивания результатов — почти так же, как если бы это был запрос SQL. Может существовать список ArrayList объектов Customer, в котором требуется найти определенный объект Customer. В этом случае LINQ to Object будет наилучшим выбором.

Интерфейс IEnumerable<T>

IEnumerable<T> — это интерфейс, реализуемый всеми классами обобщенных коллекций C#, как это делают массивы. Этот интерфейс позволяет выполнять перечисление элементов коллекций.

Последовательность — это термин для обозначения коллекции, реализующей интерфейс IEnumerable<T>. Если есть переменная типа IEnumerable<T>, то можно сказать, что имеется последовательность элементов типа T. Например, IEnumerable<string> означает последовательность строк. Любая переменная, объявленная как IEnumerable<T> для типа T, рассматривается как последовательность типа T.

Большинство стандартных операций запросов представляют собой расширяющие методы в статическом классе System.Linq.Enumerable и прототипированы 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>.

Унаследованные коллекции, не являющиеся обобщенными, которые существовали до C# 2.0, поддерживают интерфейс IEnumerable, а не IEnumerable<T>. Это значит, что нельзя непосредственно вызывать эти расширяющие методы с первым аргументом типа IEnumerable<T> на унаследованных коллекциях. Однако можно выполнять запросы LINQ на унаследованных коллекциях, вызывая стандартную операцию запроса Cast или OfType на унаследованной коллекции, чтобы произвести последовательности реализующие IEnumerable<T>, а это откроет доступ к полному арсеналу стандартных операций запросов.

Чтобы получить доступ к стандартным операциям запросов, добавьте в код директиву *using System.Linq;*, если ее еще там нет. Добавлять ссылку на сборку не понадобится, потому что необходимый код содержится в сборке System.Core.dll, которая автоматически добавляется к проекту средой Visual Studio 2010.

Важно помнить, что хотя многие из стандартных операций запросов прототипированы на возврат IEnumerable<T>, и IEnumerable<T> воспринимается как последовательность, на самом деле операции не возвращают последовательность в момент их вызова. Вместо этого операции возвращают объект, который при перечислении выдает (yield) очередной элемент последовательности. Во время перечисления возвращенного объекта запрос выполняется, и выданный элемент помещается в выходную последовательность. Таким образом, выполнение запроса откладывается.

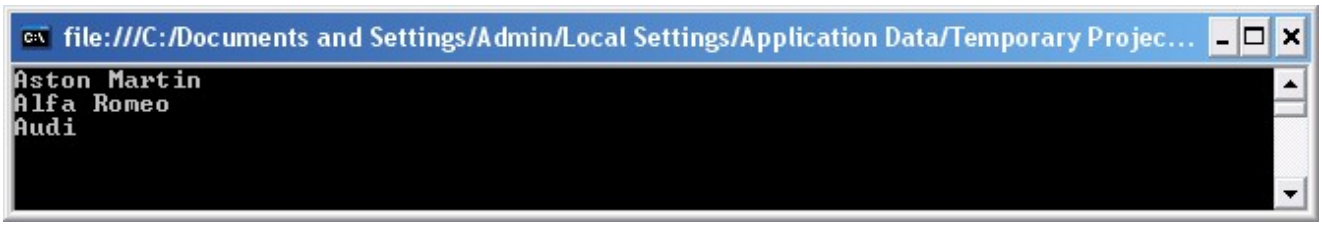
Понятие выдачи (yield) связано с ключевым словом **yield**, которое было добавлено к языку C# для облегчения написания перечислителей. Например, рассмотрим код:

C#

```
string[] cars = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge",
"BMW",
                            "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toy
ota", "Volvo", "Subaru", "Жигули :)"};
IEnumerable<string> items = cars.Where(p => p.StartsWith("A"));
foreach (string s in items)
    Console.WriteLine(s);
```

Запрос, использующий операцию Where, на самом деле не запускается, когда выполняется строка, содержащая запрос. Вместо этого возвращается объект. И только во время перечисления элементов возвращенного объекта этот запрос Where выполняется. Это значит, что ошибка, возникающая в самом запросе, может быть не обнаружена до тех пор, пока не начнется перечисление!

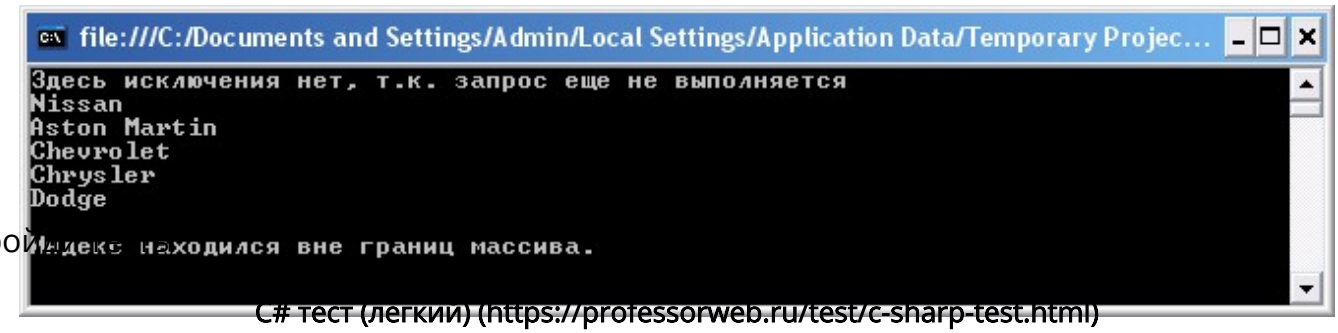
Результат предыдущего запроса выглядит следующим образом:



Запрос выполнился, как и ожидалось. Теперь внесем в него ошибку. В следующем коде предпринимается попытка проиндексировать пятый символ каждой машины. Когда перечисление достигает элемента, длина которого меньше 5 символов, возникает исключение. Однако помните, что исключение не произойдет до тех пор, пока не начнется перечисление выходной последовательности:

C#

```
IEnumerable<string> items = cars.Where(s => Char.IsLower(s[4]));
Console.WriteLine("Здесь исключения нет, т.к. запрос еще не выполняется");

try
{
    foreach (string s in items)
         Console.WriteLine(s);
}
catch (Exception ex)
{
     Console.WriteLine("\n" + ex.Message);
}
```



Прой

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)

Обратите внимание на вывод "Здесь исключения нет, т.к. запрос еще не выполняется". Он не появляется до тех пор, пока в перечислении не дойдет очередь до четвертого элемента — Alfa Romeo, где и возникает исключение. Урок, который можно отсюда извлечь, состоит в том, факт успешной компиляции запроса и кажущееся отсутствие проблем при его выполнении еще не говорит о том, что он свободен от ошибок.

Вдобавок, поскольку такого рода запросы, возвращающие IEnumerable<T>, являются отложенными, код определения запроса может быть вызван однажды и затем использован многократно, с перечислением его результатов несколько раз. В случае изменения данных при каждом перечислении результатов будут выдаваться разные результаты. Ниже показан пример отложенного запроса, где результат не кэшируется и может изменяться от одного перечисления к другому:

C#

```
//  Создать массив целых чисел.
      int[] intArray = new int[] { 1, 2, 3 };
      IEnumerable<int> ints = intArray.Select(i => i);

      foreach (int i in ints)
        Console.WriteLine(i);

      // Изменить элемент, в источнике данных
      intArray[0] = 5;

      Console.WriteLine("---------");

      foreach (int i in ints)
        Console.WriteLine(i);
```

Давайте более подробно рассмотрим, что здесь происходит. Когда вызывается операция Select, возвращается объект, хранящийся в переменной ints типа, реализующего интерфейс IEnumerable<int>. В этой точке запрос в действительности еще не выполняется, но хранится в объекте по имени ints. Другими словами, поскольку запрос еще не выполнен, последовательность целых чисел пока не существует, но этот объект ints знает, как получить последовательность, выполнив присвоенный ему запрос, которым в этом случае является операция Select.

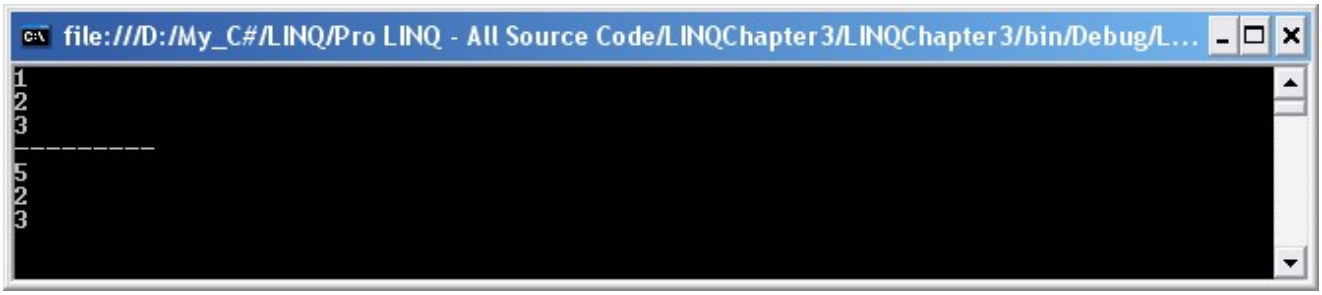
Когда оператор foreach выполняется на ints в первый раз, объект ints производит запрос и получает последовательность по одному элементу за раз.

После этого в исходном массиве целых чисел изменяется один элемент. Затем снова запускается оператор foreach. Это заставляет ints снова выполнить запрос. Поскольку элемент в исходном массиве был изменен, а запрос выполнен снова, т.к.

заново запущено перечисление ints, на этот раз возвращается измененный элемент.

Вызванный запрос вернул объект, реализующий IEnumerable<int>. Однако в большинстве случаев при обсуждении LINQ, обычно говорится, что запрос вернул последовательность целых чисел. Логически это верно, и в конечном счете так оно и есть. Но важно понимать, что происходит в действительности.

Ниже показан результат запуска этого кода:

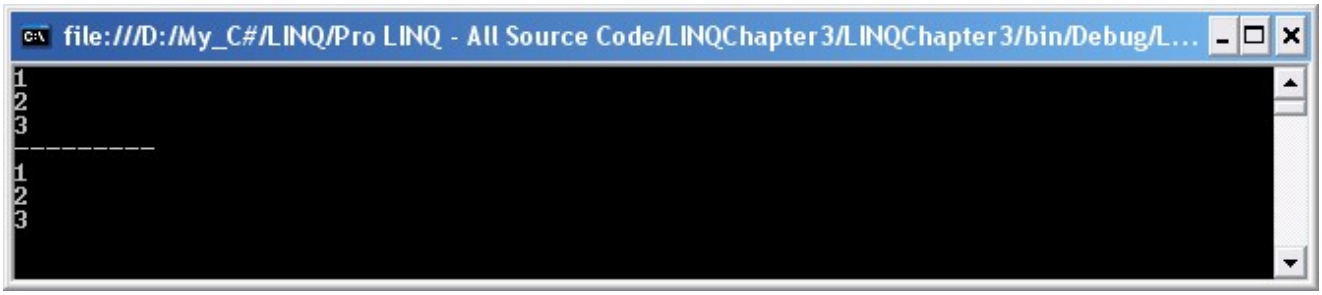


Обратите внимание, что несмотря на однократный вызов запроса, результаты двух перечислений отличаются. Это еще одно доказательство того, что запрос является отложенным. Если бы это было не так, то результаты двух перечислений совпали бы. Это может рассматриваться как преимущество, так и недостаток. Если не хотите, чтобы в таких ситуациях результаты отличались, воспользуйтесь одной из операций преобразования, которые не возвращают IEnumerable<T>, так что запрос получается не отложенным, а возвращают ToArray, ToList, ToDictionary или ToLookup, создавая различные структуры данных с кэшированными результатами, не изменяющимися с изменением источника данных.

Ниже показан тот же код, что и в предыдущем примере, но запрос возвращает не IEnumerable<T>, а List<int> — за счет вызова операции ToList:

C#

```
//  Создать массив целых чисел.
      int[] intArray = new int[] { 1, 2, 3 };

      List<int> ints = intArray.Select(i => i).ToList();

      foreach (int i in ints)
        Console.WriteLine(i);

      // Изменить элемент, в источнике данных
      intArray[0] = 5;

      Console.WriteLine("---------");

      foreach (int i in ints)
        Console.WriteLine(i);
```



Обратите внимание, что результаты, полученные от двух перечислений, одинаковы. Причина в том, что метод ToList не является отложенным, и запрос на самом деле выполняется в тот момент, когда он был вызван.

Возвращаясь к дискуссии об отличиях между этим примером и предыдущим, где операция Select отложена, следует отметить, что операция ToList отложенной не является. Когда ToList вызывается в операторе запроса, она немедленно перечисляет объект, возвращенный оператором Select, в результате чего весь запрос перестает быть отложенным.

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:

C#

```
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);
```

В каждом объявлении TR ссылается на возвращаемый тип данных. Обратите внимание, что тип возвращаемого аргумента TR находится в конце шаблона типов параметров для каждой перегрузки делегата Func. Другие параметры типа — TO, T1, T2 и T3 — ссылаются на входные параметры, переданные методу. Существует множество объявлений, потому что некоторые стандартные операции запросов имеют аргументы-делегаты, требующие больше параметров, чем другие. Взглянув на объявления, можно заметить, что ни одна из стандартных операций запросов не имеет аргумента-делегата, требующего более четырех входных параметров.

Давайте взглянем на другие пример операции Select.

C#

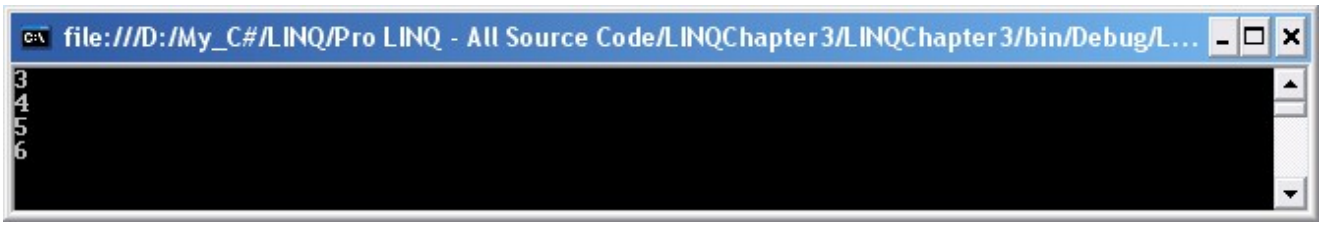
```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Аргумент-предикат указан как Func<T, bool>. Отсюда видно, что метод-предикат или лямбда-выражение должны принимать один аргумент — параметр T и возвращать bool. Вы знаете это потому, что известен тип возврата, указанный в конце списка параметров шаблона. Давайте посмотрим использование делегата Func на примере:

C#

```
int[] ints = new int[] { 1, 2, 3, 4, 5, 6 };

//  Объявление делегата
Func<int, bool> GreaterThanTwo = i => i > 2;

//  Выполнить запрос... Не забывайте об отложенных запросах!
IEnumerable<int> intsGreaterThanTwo = ints.Where(GreaterThanTwo);

foreach (int i in intsGreaterThanTwo)
  Console.WriteLine(i);
```

Этот код вернет следующие результаты:



Назад (1_5.php)	4	5	6

Пройди тесты