Интеграция LINQ с C#

Чтобы обеспечить гладкую интеграцию LINQ с C#, в язык C# понадобилось внести существенные усовершенствования. Хотя все эти средства ценны и сами по себе, на самом деле они являются частями общего вклада в LINQ, который делает расширения C# столь замечательными.

Чтобы действительно понять большую часть синтаксиса LINQ, необходимо сначала разобраться в некоторых новых средствах языка C# и только затем приступать к работе с компонентами LINQ.

Лямбда-выражения

Начиная с версии 3, язык C# поддерживает **лямбда-выражения (lambda expressions)**. Лямбда-выражения использовались в языках программирования вроде LISP с давних времен, а впервые их концепция была сформулирована в 1936 г. американским математиком Алонзо Черчем (Alonzo Church). Эти выражения представляют сокращенный синтаксис для определения алгоритма.

Но, прежде чем обратиться непосредственно к лямбда-выражениям, давайте взглянем на эволюцию способов указания алгоритма как аргумента метода, поскольку именно в этом и состоит назначение лямбда-выражений.

Использование именованных методов

Ранее, когда метод или переменная была типизирована так, что требовала делегата (delegate) (../../../csharp/charp_theory/level10/10_1.php), разработчик должен был создавать именованный метод и передавать его имя туда, где требовался делегат.

В качестве примера рассмотрим следующую ситуацию. Предположим, что есть два разработчика, один из которых занимается кодом общего назначения, а другой - прикладным. Не обязательно, чтобы это были два разных разработчика, нам просто нужно разграничить две разных роли. Разработчик общего кода может создавать код

общего назначения, который может быть использован многократно во всем проекте. Прикладной разработчик будет потреблять код общего назначения, чтобы создавать приложение.

В этом случае разработчик общего кода желает создать метод для фильтрации массивов целых чисел, но с возможностью указания алгоритма, применяемого для фильтрации. Для начала должен быть объявлен делегат. Этот делегат будет прототипирован для приема параметра int и возврата значения true, если данный int должен быть включен в отфильтрованный массив.

Итак, он создает служебный класс и добавляет делегат и метод фильтрации. Вот этот код общего назначения:

C#

```
public class Common
    {
        public delegate bool IntFilter(int i);
        public static int[] FilterArrayOfInts(int[] ints, IntFilter filter)
        {
            ArrayList aList = new ArrayList();
            foreach (int i in ints)
            {
                if (filter(i))
                {
                    aList.Add(i);
                }
            }
            return ((int[])aList.ToArray(typeof(int)));
        }
    }
```

Разработчик общего кода поместит и объявление делегата, и FilterArrayOfInts в общую библиотечную сборку — динамически подключаемую библиотеку (DLL) — чтобы его можно было использовать во многих приложениях.

Приведенный выше метод FilterArrayOfInts позволяет прикладному разработчику передавать массив целых чисел и делегат его метода фильтрации, получая обратно отфильтрованный массив.

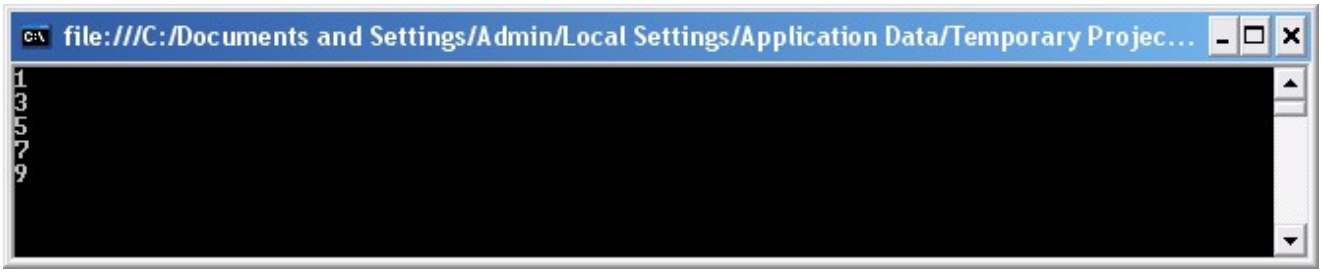
Теперь предположим, что прикладной разработчик желает отфильтровать только нечетные числа. Вот его метод фильтрации, который объявлен в прикладном коде:

```
public class Application
    {
        public static bool IsOdd(int i)
        {
            return ((i & 1) == 1);
        }
    }
```

На основе кода метода FilterArrayOfInts этот метод будет вызван для каждого значения int в массиве, переданном ему. Фильтр вернет true, если переданное значение является нечетным. Ниже показан пример использования метода FilterArrayOfInts, за которым представлен результат:

C#

```
class Program
    {
        static void Main()
        {
            int[] nums = {1,2,3,4,5,6,7,8,9,10 };
            int[] oddNums = Common.FilterArrayOfInts(nums, Application.IsOdd);
            foreach (int i in oddNums)
                Console.WriteLine(i);
        }
    }
```



Обратите внимание, что для передачи делегата во втором параметре FilterArrayOfInts прикладной разработчик передает имя метода. Просто создав другой фильтр, он может фильтровать числа иначе. Он может иметь фильтр для четных чисел, простых чисел или отобранных в соответствии с любым нужным критерием. Делегаты позволяют создавать в высшей степени многократно используемый код.

Пройди тесты

Использование анонимных методов

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

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

Так или иначе, но написание всех этих методов фильтрации или любых других методов delegate может оказаться довольно утомительным. Многие из этих методов будут использованы лишь однократно, и скучно создавать именованные методы для таких случаев. Начиная с версии C# 2.0, у разработчиков появилась возможность создавать экземпляр делегата за счет предоставления встроенного кода как анонимного метода. Анонимные методы позволяют разработчику указывать код практически везде, где обычно должен передаваться делегат. Вместо создания метода IsOdd он может написать код фильтрации прямо в точке, где обычно передается делегат:

C#

```
int[] nums = {1,2,3,4,5,6,7,8,9,10 };
int[] oddNums = Common.FilterArrayOfInts(nums, delegate(int i)
     { return ((i & 1) == 1); });
foreach (int i in oddNums)
      Console.WriteLine(i);
```

Совсем неплохо. Прикладной разработчик более не обязан где-то объявлять метод. Это замечательно для кода логики фильтрации, вероятность многократного использования которого не высока. Как и ожидалось, вывод программы не отличается от предыдущего.

С применением анонимных методов связан один недостаток. Получаемый в результате код довольно громоздкий и трудно читаемый. Должен существовать более удобный способ написания кода метода.

Использование лямбда-выражений

Лямбда-выражения определяются как разделенный запятыми список параметров, за которым следует лямбда-операция, а за ней — выражение или блок операторов. Если параметров более одного, входные параметры помещаются в скобки. В C# лямбда-операция записывается как =>. Таким образом, лямбда-выражение в C# выглядит подобно следующему:

```
(параметр1, параметр2,  параметр3) => выражение
```


Или, когда требуется более высокая сложность, может применяться блок операторов:

```
(параметр1, параметр2,  параметр3) =>
{
   оператор1;
   оператор2;
   оператор3;
   return(тип_возврата_лямбда_выражения);
}
```

В этом примере возвращаемый тип данных в конце блока операторов должен соответствовать типу возврата, указанному делегатом. Вот пример лямбда-выражения:

C#

```
x => x
```

Это лямбда-выражение может быть прочитано, как "x идет к x" или, возможно, "ввод x возвращает x". Это значит, что при входной переменной x выражение вернет x. Это выражение просто возвращает то, что оно получило. Поскольку здесь только единственный параметр x, нет необходимости заключать его в скобки. Важно знать, что этот делегат диктует тип входного параметра x и тип возврата.

Например, если делегат определен как принимающий string, но возвращающий bool, тогда выражение x => x не может использоваться, потому что если входной x будет иметь тип string, то возвращаемый x также должен относиться к типу string, но делегат определен как возвращающий bool. Поэтому с delegate, определенным таким образом, часть выражения справа от лямбда-операции (=>) должна вычисляться для возврата bool, как показано в следующем примере:

C#

```
x => x.Length > 0
```

Это лямбда-выражение может быть прочитано как "x идет в x.Length > 0", или, возможно, "ввод x возвращает x.Length > 0". Поскольку правая часть выражения вычисляется как bool, делегат должен указывать, что метод возвращает bool, иначе компилятор сообщит об ошибке.

Следующее лямбда-выражение пытается вернуть длину входного аргумента. Значит, делегат должен определять int в качестве типа возврата.

C#

```
s => s.Length
```

Если лямбда-выражению передается несколько параметров, отделяйте их запятыми и помещайте в скобки, как показано ниже:

C#

```
(x, y) => x == y
```

Сложные лямбда-выражения могут даже включать блок операторов:

C#

```
(x, y) =>
{
   if(x > y)
      return(x);
   else
      return(y);
}
```

Важно помнить, что делегат определяет, какими должны быть типы входных параметров и каким - тип возврата. Поэтому удостоверьтесь, что лямбда-выражение соответствует определению делегата.

Удостоверьтесь, что лямбда-выражение рассчитано на прием входных типов, указанных в определении делегата, и возвращает тип, определенный как возвращаемый тип делегата.

Чтобы освежить память, ниже приведено объявление делегата, определенное разработчиком общего кода:

C#

```
delegate bool IntFilter(int i);
```

Пройди тесты

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

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

Лямбда-выражение разработчика прикладного кода должно поддерживать int, переданный в параметре, и возвращать bool. Это может быть выведено из вызываемого им метода и назначения метода фильтрации, но важно помнить, что это диктуется делегатом.

Предыдущий пример, в котором на этот раз используется лямбда-выражение, должен выглядеть так:

C#

```
int[] nums = {1,2,3,4,5,6,7,8,9,10 };
int[] oddNums = Common.FilterArrayOfInts(nums, i => ((i & 1) == 1) );
foreach (int i in oddNums)
      Console.WriteLine(i);
```

Вот такой удобный код получился. Поначалу это может показаться забавным, но как только вы привыкнете, то оцените читабельность и сопровождаемость такого кода. Как и можно было ожидать, результат будет тем же, что у предыдущих примеров.

Использовать именованные методы, анонимные методы или лямбда-выражения - выбор за разработчиком. Применяйте то, что имеет смысл в каждой конкретной ситуации.

Преимуществами лямбда-выражений часто пользуются при передаче их как аргументов в вызовы операций запросов LINQ. Поскольку каждый запрос LINQ, скорее всего, будет иметь уникальное или очень ограничено используемое лямбда-выражение, это обеспечивает гибкость указания логики операции без необходимости постоянного создания методов почти для каждого запроса.

Деревья выражений

Дерево выражения (expression tree) - эффективное представление в древовидной форме данных лямбда-выражения операции запроса. Эти представления деревьев выражений могут быть вычислены все сразу, так что единственный запрос может быть построен и выполнен на одном источнике данных, таком как база данных.

В большинстве примеров, рассмотренных до сих пор, операции выражений выполнялись в линейной манере. Рассмотрим следующий код:

Пройди тесты

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

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

```
int[] nums = new int[] { 6, 2, 7, 1, 9, 3 };
IEnumerable<int> numsLessThanFour = nums
   .Where (i => i < 4)
   .OrderBy(i => i);
```

Этот запрос содержит две операции - Where и OrderBy, которые ожидают делегаты в качестве своих аргументов. В результате его компиляции генерируется код на промежуточном языке .NET (Intermediate Language — IL) (../../../csharp/assembly/level4/4_1.php), который идентичен IL-коду анонимного метода для каждого лямбда-выражения операции запроса.

Когда выполняется этот запрос, сначала вызывается операция Where, за ней — операция OrderBy. Такое линейное выполнение операций кажется оправданным для данного примера, но давайте подумаем о запросе к очень большому источнику, такому как база данных. Имеет ли смысл для SQL-запроса сначала обратиться к базе данных только с конструкцией Where, чтобы изменить порядок последующих вызовов? Естественно, это не реально для запросов к базе данных, как потенциально и для других типов запросов. Именно здесь приходят на помощь деревья выражений. Поскольку дерево выражений допускает параллельное вычисление и выполнение всех операций в запросе, может быть произведен единственный общий запрос вместо отдельных запросов для каждой операции.

Итак, теперь есть две разные вещи, которые может генерировать компилятор для лямбда-выражения операции — IL-код и дерево выражения. Что определяет, будет лямбда-выражение операции компилироваться в IL-код или в дерево выражения? Какое из этих двух действий предпримет компилятор определяется прототипом операции. Если операция объявлена для приема делегата метода, будет сгенерирован IL-код. Если же операция объявлена для приема выражения делегата, будет создано дерево выражения.

В качестве примера давайте рассмотрим две разных реализации операции Where. Первая — стандартная операция запроса, присутствующая в API - интерфейсе LINQ to Objects и определенная в классе System.Linq.Enumerable:

C#

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


Вторая реализация операции Where находится в API-интерфейсе LINQ to SQL и принадлежит классу System.Linq.Queryable:

C#

```
public static IQueryable<T> Where<T>(
   this IQueryable<T> source,
   System.Linq.Expressions.Expression(Func<int, bool> predicate);
```

Как видите, первая операция Where объявлена как принимающая делегат, на что указывает делегат Func, и компилятор для этого лямбда-выражения операции сгенерирует IL-код. Имейте в виду, что делегат Func определяет сигнатуру делегата, передаваемого в качестве аргумента предиката. Вторая операция Where объявлена для приема дерева выражения (Expression), поэтому здесь компилятор сгенерирует древовидное представление лямбда-выражения.

Расширяющие методы на последовательностях IEnumerable<T> имеют IL-код сгенерированный компилятором. Расширяющие методы на IQueryable<T> имеют сгенерированные компилятором деревья выражений.

Назад (1_2.php)	2	3	4	Вперед (1_4.php)

Пройди тесты

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

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