Reactive Extensions入门(2)：LINQ操作符

yycoding.xyz/post/2012/5/18/introducting-linq-and-reactiveextensions-core-linq

LINQ和Reactive Extension(Rx)之间有很深的联系。本文将简要介绍LINQ中的一些比较重要的操作符和方法，因为这些方法在LINQ和Rx中的作用类似。

首先以一个例子展示LINQ的语法，然后讨论一些比较重要的LINQ操作，最后以两个例子展示了这些方法的综合运用。

1. LINQ语法

LINQ有许多分支，每一个分支对应操作不同的数据类型。LINQ to Object用来操作内存中的对象，而LINQ to SQL用来和数据库中的数据进行交互。虽然LINQ to Object更好的能揭示LINQ本质,但是他们本质上都是相同的。

考虑下面的代码。

```
var primes = new List<int>() { 1, 2, 3, 5, 7, 11, 13, 17, 19 };
var query = from num in primes
            where num < 7
            select num;
foreach (var i in query)
{
    Console.WriteLine(i);
}
```

中间三行代码组成了LINQ查询表达式。查询表达式以from开头，VS中支持LINQ的类型检查和智能提示。

LINQ中还有另外一种称之为方法表达式的语法，如上面的句子用方法表达式改写则为：

```
var query = primes
                  .Where(num => num < 7)
                  .Select(num=>num);
```

上面语句中的第二行称之为过滤（filter）操作，最后一行称之为投影(projection)操作。过滤操操作用来减少结果集，投影操作用来从现有的集合中提取某些数据组成新的IEnumerable集合。

因为LINQ表达式返回的是IEnumerable对象，所以我们可以通过foreach语句来遍历结果集。

1.1查询操作

LINQ中有许多操作，在后面将会逐个介绍，现在作为开始，我们使用LINQ来查找int类型中的所有方法，并将这些方法进行排序和分组。LINQ的只能提示也使用反射来列出我们想使用的方法。

```
var query = from method in typeof(int).GetMethods()
            orderby method.Name
            group method by method.Name into groups
            select new { MethodName = groups.Key, MethodOverloads = groups.Count() };

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

输出结果为：

```
{ MethodName = CompareTo, MethodOverloads = 2 }
{ MethodName = Equals, MethodOverloads = 2 }
{ MethodName = GetHashCode, MethodOverloads = 1 }
{ MethodName = GetType, MethodOverloads = 1 }
{ MethodName = GetTypeCode, MethodOverloads = 1 }
{ MethodName = Parse, MethodOverloads = 4 }
{ MethodName = ToString, MethodOverloads = 4 }
{ MethodName = TryParse, MethodOverloads = 2 }
```

1.2 延迟执行

LINQ执行过程的一个重要特征是延迟执行，就是知道要获取数据时，才回去进行计算。考虑上面例子中的代码。你可能认为执行完query语句后，所有的值都会存到query中了。实际上，这条语句会延迟到foreach调用时才会执行。这种特性对我们编码有好处也有坏处。

优点：

下面的代码演示了一个延迟执行的例子。

```
var q = Enumerable.Range(0, 1000 * 1000)
        .Select(x => {
            Thread.Sleep(1000);
            return x * 10;
        });

foreach (var num in q)
{
    Console.WriteLine(num);
}
```

这段代码对1000000个数据进行操作，对于每一个数据，暂停1秒，然后返回计算后的值。如果在执行foreach之前，就要将这些值算都算出来的话，我们需要等待大概11.57天。延迟执行的优点就发挥出来了。在foreach方法执行时，执行一遍循环就计算一个值，然后将值打印出来。Foreach的执行方式可以参考这篇文章。在类似这样的场景中，LINQ的延迟执行很有帮助。

缺点：

LINQ的延迟执行机制有时候也有缺点。下面的例子中，我们循环打印两次结果，输出的结果和我们想象的可能不同。

```
int counter = 0;
var evenNumersInSeries = Enumerable.Range(1, 10).Select(x =>
{
    int result = counter + x;
    counter++;
    return result;

});

Console.WriteLine("第一次循环打印结果:");
foreach (int i in evenNumersInSeries)
{
    Console.Write(i + " ");
}

Console.Write(Environment.NewLine);

Console.WriteLine("第二次循环打印结果:");
foreach (int i in evenNumersInSeries)
{
    Console.Write(i + " ");
}
```

运行结果为：可以看到第二次执行和第一次执行的结果有关，这不是我们想要的结果，我们想两次执行的结果一样。

```
第一次循环打印结果:
1 3 5 7 9 11 13 15 17 19
第二次循环打印结果:
11 13 15 17 19 21 23 25 27 29
```

我们有两种方法可以解决这一问题。一种方法是在进行第二次执行时重置一下计数器。另一种方式是强制LINQ立即执行语句。我们可以通过LINQ将结果转换为Array对象来达到这一目的。我们只需要在之前的查询表达式后面加上.ToArray即可达到这一目的。

```
int counter = 0;
var evenNumersInSeries = Enumerable.Range(1, 10).Select(x =>
{
    int result = counter + x;
    counter++;
    return result;

}).ToArray();

Console.WriteLine("第一次循环打印结果:");
foreach (int i in evenNumersInSeries)
{
    Console.Write(i + " ");
}

Console.Write(Environment.NewLine);

Console.WriteLine("第二次循环打印结果:");
foreach (int i in evenNumersInSeries)
{
    Console.Write(i + " ");
}
```

运行结果如下:

```
第一次循环打印结果:
1 3 5 7 9 11 13 15 17 19
第二次循环打印结果:
1 3 5 7 9 11 13 15 17 19
```

2. LINQ比较重要的操作符

使用LINQ感觉很好的一个很重要的原因是他有很多很好用的操作，这些操作很多，这里只挑选一些值得关注的操作符。

2.1 Any

Any操作符用来判断序列是否为空或者谭端序列是否包含某个特定的值。

```
var firstList = Enumerable.Empty<Int32>();
var secondList = Enumerable.Range(1, 10);
Console.WriteLine("The first list has member? {0}, The second list has member? {1}",
                firstList.Any(), secondList.Any());
```

代码输出结果为:

```
The first list has member? False, The second list has member? True
```

现在添加第二个测试:

```
var firstList = Enumerable.Empty<Int32>();
var secondList = Enumerable.Range(1, 10);
Console.WriteLine("Is 6 in the second list?{0}, Is 12 in the second list?{1}",
                secondList.Any(x => x == 6), secondList.Any(x => x == 12));
```

现在运行结果为：

```
Is 6 in the second list?True, Is 12 in the second list?False
```

值得注意的是Any操作符在满足第一个条件时就会返回。比如说如果在第二个list里面6出现了三个，那么在遇到第一个时就会返回值了。

2.2 Contains

判断某个值是否在一个集合中的第二种方法是使用Contains操作符。以上面的两个集合为例。我们用Contains改写如下：

```
var firstList = Enumerable.Empty<Int32>();
var secondList = Enumerable.Range(1, 10);
Console.WriteLine("SecondList contains 6? {0},SecondList contains 12? {1}",
                    secondList.Contains(6),secondList.Contains(12));
```

结果如下：

```
SecondList contains 6? True,SecondList contains 12? False
```

上面的语法可能有些简单，Contains操作符有很多重载，你可以传入自己的比较方法来为不同的类型进行比较。

如下代码，首先创建一个Person类，然后创建一个继承自Person的Student类。

```
class Person
{
    public Int32 ID { get; set; }
    public String FullName { get; set; }
}
class Student : Person
{
    public String Major { get; set; }
}
```

然后创建一个实现IEqualityComparer的类，这个类用来判断两个People对象是否相等。这个相等性可以自己定义，可以是FullName一样或者ID相等就认为相等。这里我们只比较ID对象。

```
class StudentToPersonEquals : IEqualityComparer<Person>
{

    public bool Equals(Person x, Person y)
    {
        if (x.ID == y.ID)
            return true;
        else
            return false;
    }

    public int GetHashCode(Person obj)
    {
        return obj.GetHashCode();
    }
}
```

然后，实例化Person和Student对象，将他们放在一个集合中。

```
static void Main(string[] args)
{
    var people = new List<Person>();
    var Zhangsan = new Student() { FullName = "Zhangsan", ID = 1, Major = "Computer
Science" };

    people.Add(Zhangsan);
    people.Add(new Student() { ID = 2, FullName = "Lisi" });
    people.Add(new Student() { ID = 3, FullName = "Wangwu", Major = "Chemistry" });
    people.Add(new Student() { ID = 4, FullName = "Zhaoliu", Major = "History" });
    Console.WriteLine("People contains Zhangsan? {0}",
            people.Contains(Zhangsan,new StudentToPersonEquals()));

}
```

2.3 Take

Take操作使得我们可以确定从一个大的集合返回序列开头指定个数的元素，来进行操作，如下：

```
var input = Enumerable.Range(1, 20);
var output = input.Take(5).Select(x => x * 10);
foreach (var item in output)
{
    Console.Write("{0} ", item);
}
```

输出结果为： 10 20 30 40 50。

在这个例子中，input对象有60个元素，如果没有Take操作，将会返回10到200。Take(5)操作使得只返回集合中的前五个元素。一般滴，Take经常和Skip操作一起使用来完成分页逻辑。如Skip((page-1)*numberPerpage).Take(numerPerPage);

2.4 Distinct

通常我们的集合中有重复数据，Distinct操作会去除这些重复操作，然后返回唯一的值。

```
var input = new[] { 1, 2, 3, 2, 1, 2, 3, 2, 1, 2, 3, 2, 1 };
var output = input.Distinct().Select(x => x * 10);
```

输出结果为：10 20 30

2.5 Zip

Zip操作比较有意思，他将两个list对象拼合到一起。使用Zip的最简单方法是传入两个list对象，然后传入一个lambda表达式只是如何将这两个对象拼合到一起。结果是一个Enumerable对象。

```
String[] codes = { "SH", "BJ","GZ" ,"WH", "CQ"  };
String[] cities = { "上海", "北京", "广州", "武汉", "重庆" };
var codewithCity = codes.Zip(cities, (code, city) => code + ":" + city);
foreach (var item in codewithCity)
{
    Console.WriteLine(item);
}
```

codewithCity是一个Enumerable对象，这个对象将这两个list合成到了一起，输出结果为：

```
SH:上海
BJ:北京
GZ:广州
WH:武汉
CQ:重庆
```

Zip操作并不需要两个lsit对象的类型和长度相同。

```
Int32[] codes = Enumerable.Range(1,100).ToArray();
String[] cities = { "上海", "北京", "广州", "武汉", "重庆" };
var codewithCity = codes.Zip(cities, (code, city) => code + ":" + city);
foreach (var item in codewithCity)
{
    Console.WriteLine(item);
}
```

在这个例子中，第一个集合为100个Int32类型的值，Zip操作在达到某一个list的长度时就停止，因为第二个list只有5个元素，所以如下输出结果只有5个元素：

```
1:上海
2:北京
3:广州
4:武汉
5:重庆
```

2.6 SelectMany

SelectMany是一个功能强大，但是也不太好理解的操作符。使用SelectMany的一个最主要的目的是将集合中的内嵌集合提取到一个集合中来。当然SelectMany还有其他用途。例如如果我们有这样一个集合[1,2,3],[4],[5,6]，对这个集合进行SelectMany的话，就会返回[1,2,3,4,5,6]这样一个单一集合。

理解SelectMany方法非常重要，因为Reactive Framework中该方法能够帮助我们链式的调用异步方法。下面两个例子来说明SelectMany的用法，第一个展示了如何将多级数组展平为一个数组，第二个演示了递归遍历元素。

将多级对象展平

为了理解SelectMany操作的将多级对象展平的用法，我们举了两个例子。在第一个例子中，我们创建一个Book对象，该对象中包含一系列作者。

```
class Book
{
    public String Title { get;set;}
    public List<Author> Authors { get; set; }
}

class Author
{
    public String FullName { get; set; }
}
```

然后我们需要准备数据源，一般的数据源可以从数据库或者WebService获取，现在为了演示，我们通过一个静态的方法提供一个数据源，在这个数据源中，包括两本书，每一本书有一个或者多个作者。

```
public static List<Book> GetBooks()
{
    var books = new List<Book>()
    {
        new Book{
            Title="Introduction to Algorithms",
            Authors=new List<Author>()
            {
                new Author{FullName="Thomas H. Cormen"},
                new Author{FullName="Charles E. Leiserson"},
                new Author{FullName="Ronald L. Rivest"},
                new Author{FullName="Clifford Stein"}
            }
        },
        new Book
        {
            Title="Design Patterns: Elements of Reusable Object-Oriented Software",
            Authors=new List<Author>()
            {
                new Author{FullName="Erich Gamma"},
                new Author{FullName="Richard Helm"},
                new Author{FullName="Ralph Johnson"},
                new Author{FullName="John Vlissides"}
            }
        }
    };
    return books;
}
```

数据准备好了之后，我们想要获取所有这两本书的所有作者信息。

```
var books = GetBooks();
var q = from b in books
        select b.Authors;
```

第一句获取所有的数据，第二句使用LINQ语句来获取集合中的所有作者信息。

如果我们想打印出所有的作者信息，该怎么办呢。我们可以使用for循环，但是q中并不包含一个包含所有作者的集合。它包含了一个书本的集合，每一本书中包含一个作者的集合（select b.Authors表示我们只选择Authors而不关心Title），所以要获取所有的作者信息，我们需要循环嵌套两次，如下：

```
foreach (var book in q)
{
    foreach (var auth in book)
    {
        Console.WriteLine(auth.FullName);
    }
}
```

输出结果如下：

```
Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
```

我们可以通过SelectMany方法来完成这个需要循环两次的操作，原来的集合中有两个Book对象，每个对象中只有一个List的Authors对象，现在可以将这两个对象中的两个List对象通过SelectMany方法一次合并到一个集合中来，这就是SelectMany将“**多级对象展平**”(flattening hierarchies)的功能，代码如下：

```
var q = books.SelectMany(book => book.Authors);
foreach (var auth in q)
{
    Console.WriteLine(auth.FullName);
}
```

输出结果和之前的相同。

递归遍历层级对象

为了演示SelectMany的功能，上面的例子可能有点简单。另一个思考SelectMany的方法是“对于集合中的每一个元素，我们可以将其替换为空，一个单一对象，或者另外一个集合对象”，下面我们通过SelectMany来获取我的电脑桌面上的所有文件，包括文件夹中的文件。

```
static IEnumerable<String> GetFilesInAllSubdirectories(String root)
{
    var di = new System.IO.DirectoryInfo(root);
    return di.GetDirectories().SelectMany(x => GetFilesInAllSubdirectories(x.FullName))
                              .Concat(di.GetFiles().Select(x => x.FullName));
}

var allFilesOnDesktop = GetFilesInAllSubdirectories(
    System.Environment.GetFolderPath(Environment.SpecialFolder.Desktop));

foreach (var file in allFilesOnDesktop)
{
    Console.WriteLine(file);
}
```

核心代码在GetFilesInAllSubdirectories的第二条语句中：

.SelectMany(x => GetFilesInAllSubdirectories(x.FullName))

我们从头开始看，我们将桌面的文件夹路径传进去。我们知道所有的循环迭代完成后我们再返回所有的文件名。

在GetFilesInAllSubdirectories迭代方法中，我们首先通过传进去的文件夹构造一个DirectoryInfo对象。当调用该对象的GetDirectories方法时，会返回该文件夹下所有的子文件夹，然后再每一个子文件夹上调用SelectMany方法迭代相同的方法，传进去每一个子文件夹。

当最底层的文件夹迭代完之后，我们使用Concat操作符来将所有文件夹通过GetFiles方法获得的当前文件夹下的文件，然后通过select方法获取文件的文件名，最后把所有文件名拼成一个集合。

关键在于GetDirectories方法返回的是一个关于文件夹的层级对象。SelectMany对象将所有文件夹的每一个文件夹下的子文件夹对象展开到当前Directories集合中，使得我们可以直接对这个集合进行迭代。

理解LINQ这些比较重要的操作很重要，下面我们以两个实际的例子来展示LINQ的用处。

3. 例子

3.1 解析以特定符号分隔的数据文件

在现实编程中，我们有时会对一些文本文件进行解析，这些文件中的数据是以某个标识符分隔的，比如Tab键或者分号，最典型的就是.csv文件了，通常用来进行数据交换。下面来掩饰如何对这种文件进行解析。

下面是某个以Tab分隔的csv文本文件，第一行是标题，后面的是内容。

name id beer_style first_brewed alcohol_content original_gravity final_gravity ibu_scale country brewery_brand color_srm from_region containers

Miller Genuine Draft /m/02hv39w

Hakim Stout /m/059q7h1 Stout 5.8 Harar Beer Factory Ethiopia

Wellington County Dark Ale /m/04dqvym 5.0 Wellington Canada /m/04dr5xv

De Regenboog 't Smisje Honingbier /m/04dqbhm 6.0 Regenboog Brewery Belgium /m/04dqxph

Hop Back Entire Stout /m/04dqf8w 4.5 Hop Back Brewery United Kingdom /m/04dqzb7

呵呵，有点乱。第一行是头，每一个字段以tab分隔。接下来是数据部分。每一行的数据并不是完整的，有的列并没有数据。不完全的列有多个tab分隔。如果屏幕分辨率够大的话，数据显示出来可能更好看一点。

我们首先需要打开文件，我们原始的数据是一行一行存储的，我们使用LINQ来解析这些数据并进行统计。

```
var lines = File.ReadAllLines(@"beer.tsv");
```

首先，我们需要获取文件中的第一行来产生数据列字段。如果不用LINQ，我们可能需要像下面的代码那样书写，首先定义一个Dictionary对象，key是列名称，值是改列处于第一列，如("from_region => 6"))。

```
int i = 0;
var headers = new Dictionary<String, Int32>();
foreach (var header in lines[0].Split('\t'))
{
    headers[header] = i;
    i++;
}
```

输出结果如下：

```
[name, 0]
[id, 1]
[beer_style, 2]
[first_brewed, 3]
[alcohol_content, 4]
[original_gravity, 5]
[final_gravity, 6]
[ibu_scale, 7]
[country, 8]
[brewery_brand, 9]
[color_srm, 10]
[from_region, 11]
[containers, 12]
```

现在我们尝试用LINQ方法完成上面的任务。使用LINQ来解决问题时的代码显得更有函数式的编程风格。首先我们使用Zip方法来将我们生成的0……100和表格头进行合并。

```
var headers = Enumerable.Zip(lines[0].Split('\t'),Enumerable.Range(0, 100),(header,
index) => new { header, index })
                        .ToDictionary(k => k.header, k => k.index);
```

输出结果和之前的一样。

和之前的使用zip方法不同，之前是一个集合对象使用zip对象和另外一个集合对象进行合并，现在是调用Enumerable对象的zip方法，然后传入两个集合对象，以及一个组合的方式来产生新的集合，达到的效果是一样的。

对比之上的两种实现方式，我们可以看到LINQ方式使得代码更加具有函数式风格，代码更加简单明了和易于维护。

LINQ以及Rx的一个主要特征是数据的组合性。第一个查询的结果可以被第二个查询使用。简单的来说，这种操作运行数据经过一系列连贯的操作，然后转换成我们想要的数据。举例来说，我们想跳过表头，直接读取数据。这只需要一条语句即可完成。

```
IEnumerable<String[]> allRecords=lines.Skip(1).Select(line=>line.Split('\t'));
```

上面的句子中,skip(1)跳过第一行的表头部分，然后把后面的每一行进行分割，最后获取到的allRecord对象就是我们要的数据了，allRecord对象中，每一行数据是一个string的数组。

有了这些数据后，我们就可以对这些数据进行统计了。现在我们来打印出数据中from_region字段对应的所有国家。

```
var allCounrties = allRecords.Select(fileds =>
fileds[headers["from_region"]]).Distinct();
```

结果如下：

```
Ethiopia
Canada
Belgium
United Kingdom
Kenya
Netherlands
United States of America
Germany
Spain
South Korea
……
```

下一步，我们要从这些数据中找出前5个酿造时间最老的酒。使用LINQ或者Rx能够很容易的完成这一操作，而且如何操作的可以直观的从代码中看出来，代码更富有表现力。

```
var oldestBeers = allRecords.Select(x => new { Name = x[headers["name"]], FirstBrewed =
x[headers["first_brewed"]] })
                            .Where(x => !String.IsNullOrWhiteSpace(x.FirstBrewed))
                            .OrderBy(x =>
                            {
                                int ret = Int32.MaxValue;
                                return (Int32.TryParse(x.FirstBrewed, out ret) ? ret :
Int32.MaxValue);
                            })
                            .Take(5);
```

结果如下：

```
{ Name = Budweiser Budvar, FirstBrewed = 1265 }
{ Name = Franziskaner Hefe-Weisse Hell, FirstBrewed = 1516 }
{ Name = Grolsch, FirstBrewed = 1615 }
{ Name = Kilkenny, FirstBrewed = 1710 }
{ Name = Guinness, FirstBrewed = 1759 }
```

最后来一个最复杂的操作，我们想计算一下每一个国家酿造的酒的烈度平均值，并按期降序排列。

首先我们将各种啤酒以原产国分组，对于每一个国家，我们队该国的酒的烈度去平均值。我们可以使用Aggregate来计算平均值。使用where语句来忽略缺少数据的行。

```
var strongestbeer = byRegeion.Select(group => new
                                {
                                    Country = group.Key,
                                    Strength = group
                                                .Select(g =>
g[headers["alcohol_content"]])
                                                .Where(x =>
!String.IsNullOrWhiteSpace(x))
                                                .Aggregate(0.0, (acc, x) => acc +
Double.Parse(x) / group.Count())
                                })
                                    .Where(x => x.Strength > 0.0)
                                    .OrderByDescending(x => x.Strength)
                                    .ThenBy(x => x.Country);
```

运行结果如下：

```
{ Country = Sri Lanka, Strength = 8 }
{ Country = Kosovo, Strength = 7 }
{ Country = Belgium, Strength = 6.98925619834712 }
{ Country = Switzerland, Strength = 6.9 }
{ Country = Denmark, Strength = 6.75333333333333 }
{ Country = Poland, Strength = 6.64814814814815 }
{ Country = Netherlands, Strength = 6.55142857142856 }
{ Country = Nigeria, Strength = 6.55 }
……
```

从上面的例子中可以看出，LINQ可以极大的方便我们对数据进行操作和处理。

3.2 一个题目

这个例子来源于网上的一个面试题，题目是这样的：

一个文件中有一千多条记录。每一行记录包含一个字符以及若干个数字，设计一个算法，算法接受若干个数字，然后打印出最能匹配这些数字所在行的那个字符。

以下文件格式

input file

---

aa 3 4 10 2

bb 9 14 15 21 3

cc 12 1024 200 3 9 4

下面是测试数据：

input: 3 4 10

output: aa

input: 12 3 4

output: cc

input: 3 9

output: bb

input: 3 9

output: cc

input: 3 4 12

output: cc

解决这一问题可以使用LINQ，代码如下：

```
var invertedIndex = File.ReadAllLines("input.txt").
            SelectMany(line => {
                var fs = line.Split();
                return Enumerable.Range(1, fs.Length - 1).
                        Select(i => new { Num = int.Parse(fs[i]), Word = fs[0] }); })
                        .Distinct()
                        .ToLookup(x => x.Num, x => x.Word);

foreach (var item in invertedIndex)
    Console.WriteLine(item);

var results=new int[][] { new int[] { 3, 4, 10 },
                          new int[] { 12, 3, 4 },
                          new int[] { 3, 9 },
                          new int[] { 3, 4, 12 } }
                          .Select(x => x.SelectMany(xx => invertedIndex[xx])
                          .GroupBy(xx => xx)
                          .OrderByDescending(xx => xx.Count())
                          .First().Key);
```

4. 结语

本文介绍了LINQ中IEnumerable对象以及其的一些列比较重要的操作符。并通过一些简单的例子展现了这些操作符的用法，最后一两个现实编程中的例子演示了这些操作符的综合运用。因为LINQ和Rx中的一些操作符功能都是相似的，所以了解LINQ中的这些主要操作符对您使用LINQ和了解Rx都会有用处。希望本文对于您了解LINQ及Rx有所帮助。