介绍LINQ之前先介绍一下枚举器
Iterator：枚举器
如果你正在创建一个表现和行为都类似于集合的类，允许类的用户使用foreach语句对集合中的成员进行枚举将会是很方便的。
我们将以创建一个简单化的List Box作为开始，它将包含一个8字符串的数组和一个整型，这个整型用于记录数组中已经添加了多少字符串。
构造函数将对数组进行初始化并使用传递进来的参数填充它。

```
/// <summary> /// Iterator：枚举器 /// 测试枚举器，继承IEnumerable，实现IEnumerator<T> GetEnumerator() /// </summary> /// <typeparam name="T"></typeparam> public class ListBox<T> : IEnumerable<T> { public List<T> Strings = new List<T>(); public ListBox(params T[] initialStrings) { foreach (T s in initialStrings) { Strings.Add(s); } } IEnumerator<T> IEnumerable<T>.GetEnumerator()//这个方法和下面的方法都必须实现 { foreach (T s in Strings) { yield return s; } } //GetEnumerator方法使用了新的 yield 语句。yield语句返回一个表达式。yield语句仅在迭代块中出现， //并且返回foreach语句所期望的值。那也就是，对GetEnumerator的每次调用都//将会产生集合中的下一个字符串；所有的状态管理已经都为你做好了！ IEnumerator IEnumerable.GetEnumerator()//这个方法和上面的方法都必须实现 { throw new NotImplementedException(); } }
```

语言集成查询 (LINQ) 是一组技术的名称，这些技术建立在将查询功能直接集成到 C# 语言（以及 Visual Basic
和可能的任何其他 .NET 语言）的基础上。 借助于 LINQ，查询现在已是高级语言构造，就如同类、方法、事件等等。
对于编写查询的开发人员来说，LINQ 最明显的“语言集成”部分是查询表达式。
查询表达式是使用 C# 3.0 中引入的声明性查询语法编写的。
通过使用查询语法，您甚至可以使用最少的代码对数据源执行复杂的筛选、排序和分组操作。
您使用相同的基本查询表达式模式来查询和转换 SQL 数据库、ADO.NET 数据集、XML 文档和流以及 .NET 集合中的数据。
LINQ的基本格式如下所示： var <变量> = from <项目> in <数据源> where <表达式> orderby <表达式>
group分组子句 语句格式：var str = from p in PersonList group p by p.age group子句将数据源中的数据进行分组，在遍历数据元素时，并不像前面的章节那样直接对元素进行遍历，因为group子句返回的是元素类型为IGrouping<TKey,TElement>的对象序列，必须在循环中嵌套一个对象的循环才能够查询相应的数据元素。 在使用group子句时，LINQ查询子句的末尾并没有select子句，因为group子句会返回一个对象序列，通过循环遍历才能够在对象序列中寻找到相应的对象的元素，如果使用group子句进行分组操作，可以不使用select子句。 orderby排序子句 语句格式：var str = from p in PersonList orderby p.age select p; orderby子句中使用descending关键字进行倒序排列 示例代码如下： var str = from p in PersonList orderby p.age descending select p; orderby子句同样能够进行多个条件排序，只需要将这些条件用“，”号分割即可 示例代码如下： var str = from p in PersonList orderby p.age descending,p.name select p; join连接子句 在LINQ中同样也可以使用join子句对有关系的数据源或数据

对象进行查询，但首先这两个数据源必须要有一定的联系 var str = from p in PersonList join car in CarList on p.cid equals car.cid select p;
下面的这个是LInq中常用的类

Enumerable 类 (System.Linq)

```
public class LinqClass { public static void Test1() { int[] scores = { 90, 92, 42, 46, 37, 32, 74, 5, 16, 32 }; IEnumerable<int> scoreQuery = from score in scores //必须 where score > 40 //可选条件 orderby score descending //可选条件 select score; //必须 foreach (int score in scoreQuery) { Console.WriteLine(score); } int hightestScore = scores.Max(); //group 分组 var queryGroups = from score in scores group score by score; //into 存储查询的内容 //// percentileQuery is an IEnumerable<IGrouping<int, Country>> var percentileQuery = from score in scores group score by score into scorequery where scorequery.Key > 10 select scorequery; //在 from 开始子句以及 select 或 group 结束子句之间， //所有其他子句（where、join、orderby、from、let）都是可选的。 //任何可选子句都可以在查询正文中使用零次或多次。 //let 子句 //使用 let 子句可以将表达式（如方法调用）的结果存储到新的范围变量中。 string[] names = { "a n", "b c", "c n", "d m" }; IEnumerable<string> queryFirstNames = from name in names let firstName = name.Split(new char[] { ' ' })[0] select firstName; //对一个序列应用累加器函数。 //Aggregate<TSource>(IEnumerable<TSource>, Func<TSource，TSource，TSource>) string sentence = "the quick brown fox jumps over the lazy dog"; string[] words = sentence.Split(' '); string reversed = words.Aggregate((wording, next) => wording + " " + next); Console.WriteLine(reversed); //确定是否对序列中的所有元素都满足条件。 //All<TSource>(IEnumerable<TSource>，Func<TSource，Boolean>) //确定序列是否包含任何元素。 //Any<TSource>(IEnumerable<TSource>) List<int> numbers = new List<int> { 1, 2 }; bool hasElements = numbers.Any(); Console.WriteLine("The list {0} empty.", hasElements ? "is not" : "is"); // 返回输入类型化为 IEnumerable<T>。 // AsEnumerable<TSource>(IEnumerable<TSource>) //将强制转换的元素 IEnumerable 为指定的类型。符合规则热任意类型，与上面的区别。 //Cast<TResult>(IEnumerable) //计算序列的平均值 Double 值。 //Average(IEnumerable<Double>) //连接两个序列。 //Concat<TSource>(IEnumerable<TSource>，IEnumerable<TSource>) //确定序列是否包含指定的元素使用的默认相等比较器。 //Contains<TSource>(IEnumerable<TSource>，TSource) //返回序列中的元素数。 //Count<TSource>(IEnumerable<TSource>) //返回单一实例集合中指定的序列或类型参数的默认值的元素，如果序列为空。 //DefaultIfEmpty<TSource>(IEnumerable<TSource>) //通过使用的默认相等比较器对值进行比较从序列返回非重复元素。 //Distinct<TSource>(IEnumerable<TSource>) //返回序列中的指定索引处的元素。 //ElementAt<TSource>(IEnumerable<TSource>，Int32) //通过使用默认的相等比较器对值进行比较，生成两个序列的差集。 //Except<TSource>(IEnumerable<TSource>，IEnumerable<TSource>) double[] numbers1 = { 2.0, 2.1, 2.2, 2.3, 2.4, 2.5 }; double[] numbers2 = { 2.2 }; IEnumerable<double> onlyInFirstSet = numbers1.Except(numbers2); foreach (double number in onlyInFirstSet) Console.WriteLine(number); //返回一个序列的第一个元素。 //First<TSource>(IEnumerable<TSource>) //返回序列中满足指定条件的第一个元素。 //FirstOrDefault<TSource>(IEnumerable<TSource>) //根据指定的键选择器函数对序列的元素进行分组。 //GroupBy<TSource，TKey>(IEnumerable<TSource>，Func<TSource，TKey>) // 通过使用默认的相等比较器对值进行比较，生成两个序列的交集。 //Intersect<TSource>(IEnumerable<TSource>, IEnumerable<TSource>) int[] id1 = { 44, 26, 92, 30, 71, 38 }; int[] id2 = { 39, 59, 83, 47, 26, 4, 30 };
```

```
IEnumerable<int> both = id1.Intersect(id2); foreach (int id in both) Console.WriteLine(id); //通过使用默认的相等比较器生成的两个序列的并集。 //Union<TSource>(IEnumerable<TSource>, IEnumerable<TSource>) //返回一个序列的最后一个元素。 //Last<TSource>(IEnumerable<TSource>) // 从序列的开头返回指定的数量的连续元素。 // Take<TSource>(IEnumerable<TSource>，Int32) // 返回序列中的最大值 Decimal 值。 //Max(IEnumerable<Decimal>) //返回序列的唯一元素；如果该序列并非恰好包含一个元素，则会引发异常。 //Single<TSource>(IEnumerable<TSource>) //ToDictionary<TSource，TKey>(IEnumerable<TSource>，Func<TSource，TKey>) //Where<TSource>(IEnumerable<TSource>，Func<TSource，Boolean>) // 一个序列的每个元素投影 IEnumerable<T> 并将合并为一个序列将结果序列。 //SelectMany<TSource，TResult>(IEnumerable<TSource>，Func<TSource，IEnumerable<TResult>>) } }
```

其他的基础功能

1. C#高级功能（四）扩展方法和索引
2. C#高级功能（三）Action、Func，Tuple
3. C#高级功能（二）LINQ 和Enumerable类
4. C#高级功能（一）Lambda 表达式
5. C#中泛型的解释（object,list,var,dynamic的区别）
6. C#中委托
7. C#和.NET版本对比

转载于:https://www.cnblogs.com/zhao123/p/5621841.html