

2010

# LINQ 标准查询操作符

Via C#

标准查询操作符是 LINQ 查询表达式的构成模块，他们提供了许多查询功能，如过滤和排序等。标准查询操作符是构成查询模式的一系列方法的集合，并通过各自的 LINQ 提供程序来实现



## 目录

一、投影操作符 .....	4
1. Select.....	4
2. SelectMany.....	4
二、限制操作符 .....	5
三、排序操作符 .....	5
1. OrderBy .....	5
2. OrderByDescending .....	6
3. ThenBy.....	6
4. ThenByDescending .....	6
5. Reverse.....	7
四、联接操作符 .....	7
1. Join.....	7
2. GroupJoin.....	8
五、分组操作符 .....	9
六、串联操作符 .....	10
七、聚合操作符 .....	10
1. Aggregate.....	10
2. Average.....	11
3. Count .....	11
4. LongCount .....	11
5. Max.....	11
6. Min.....	11
7. Sum .....	11
八、集合操作符 .....	12
九、生成操作符 .....	12
1. Empty.....	12
2. DefaultIfEmpty.....	12
3. Range .....	13
4. Repeat .....	13
十、转换操作符 .....	14
1. AsEnumerable .....	14
2. Cast .....	14
3. OfType .....	15
4. ToArray.....	15
5. ToDictionary .....	15
6. ToList .....	15
7. ToLookup.....	15
十一、元素操作符.....	16
1. First .....	16
2. FirstOrDefault .....	17
3. Last.....	18
4. LastOrDefault.....	18
5. ElementAt .....	18

6. ElementAtOrDefault .....	18
7. Single .....	18
8. SingleOrDefault .....	18
十二、相等操作符.....	19
十三、限定操作符.....	20
1. All.....	20
2. Any.....	20
3. Contains .....	20
十四、分区操作符.....	21
1. Take.....	21
2. TakeWhile.....	21
3. Skip .....	22
4. SkipWhile .....	22
本文总结.....	23
关于作者.....	23

# 一、投影操作符

## 1. Select

Select 操作符对单个序列或集合中的值进行投影。下面的示例中使用 select 从序列中返回 Employee 表的所有列：

```
using (NorthwindDataContext db=new NorthwindDataContext())
{
    //查询语法
    var query =
        from e in db.Employees
        where e.FirstName.StartsWith("M")
        select e;

    //方法语法
    var q =
        db.Employees
        .Where(e => e.FirstName.StartsWith("M"))
        .Select(e => e);

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

当然，你也可以返回单个列，例如：

```
var query =
    from e in db.Employees
    where e.FirstName.StartsWith("M")
    select e.FirstName;
```

你也可以返回序列中的某几列，例如：

```
var query =
    from e in db.Employees
    where e.FirstName.StartsWith("M")
    select new
    {
        e.FirstName,
        e.LastName,
        e.Title
    };
```

## 2. SelectMany

SelectMany 操作符提供了将多个 from 子句组合起来的功能，它将每个对象的结果合并成单个序列。下面是一个示例：

```
using (NorthwindDataContext db=new NorthwindDataContext())
```

```
{  
    //查询语法  
    var query =  
        from e in db.Employees  
        from o in e.Orders  
        select o;  
  
    //方法语法  
    var q =  
        db.Employees  
        .SelectMany(e => e.Orders);  
  
    foreach (var item in query)  
    {  
        Console.WriteLine(item.Freight);  
    }  
}
```

## 二、限制操作符

Where 是限制操作符，它将过滤标准应用在序列上，按照提供的逻辑对序列中的数据进行过滤。

Where 操作符不启动查询的执行。当开始对序列进行遍历时查询才开始执行，此时过滤条件将被应用到查询中。Where 操作符的使用方法已经在第一节中出现过，这里不再冗述。

## 三、排序操作符

排序操作符，包括 OrderBy、OrderByDescending、ThenBy、ThenByDescending 和 Reverse，提供了升序或者降序排序。

### 1. OrderBy

OrderBy 操作符将序列中的元素按照升序排列。下面的示例演示了这一点：

```
using (NorthwindDataContext db = new NorthwindDataContext())  
{  
    //查询语法  
    var query =  
        from e in db.Employees  
        orderby e.FirstName  
        select e;  
  
    //方法语法  
    var q =  
        db.Employees  
        .OrderBy(e => e.FirstName)  
        .Select(e => e);  
}
```

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

这里可以使用 `OrderBy` 的重载方法 `OrderBy(Func<T,TKey>,IComparer<Tkey>)`来指定序列的排序方式。

## 2. OrderByDescending

`OrderByDescending` 操作符将序列中的元素按照降序排列。用法与 `OrderBy` 相同，这里不再演示。

## 3. ThenBy

`ThenBy` 操作符实现按照次关键字对序列进行升序排列。此操作符的查询语法与方法语法略有不同，以下代码演示了这一点：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    //查询语法
    var query =
        from e in db.Employees
        orderby e.FirstName, e.LastName
        select e;

    //方法语法
    var q =
        db.Employees
        .OrderBy(e => e.FirstName)
        .ThenBy(e => e.LastName)
        .Select(e => e);

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

## 4. ThenByDescending

`ThenByDescending` 操作符实现按照次关键字对序列进行降序排列。此操作符的查询语法与方法语法略有不同，以下代码演示了这一点：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    //查询语法
    var query =
        from e in db.Employees
        orderby e.FirstName, e.LastName descending
```

```
select e;

//方法语法
var q =
    db.Employees
        .OrderBy(e => e.FirstName)
        .ThenByDescending(e => e.LastName)
        .Select(e => e);

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

## 5. Reverse

`Reverse` 将会把序列中的元素按照从后到前的循序反转。需要注意的是, `Reverse` 方法的返回值是 `void`, 以下代码演示了这一点:

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    //方法语法
    var q =
        db.Employees
            .Select(e => e.FirstName)
            .ToList();
    q.Reverse();

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

## 四、联接操作符

联接是指将一个数据源对象与另一个数据源对象进行关联或者联合的操作。这两个数据源对象通过一个共同的值或者属性进行关联。

LINQ 有两个联接操作符: `Join` 和 `GroupJoin`。

### 1. Join

`Join` 操作符类似于 T-SQL 中的 `inner join`, 它将两个数据源相联接, 根据两个数据源中相等的值进行匹配。例如, 可以将产品表与产品类别表相联接, 得到产品名称和与其相对应的类别名称。以下的代码演示了这一点:

```
using (NorthwindDataContext db = new NorthwindDataContext())
```

```
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //查询语法
    var query =
        from p in db.Products
        join c in db.Categories on p.CategoryID equals c.CategoryID
        where p.CategoryID == 1
        select p;

    //方法语法
    var q =
        db.Products
        .Join
        (
            db.Categories,
            p => p.CategoryID,
            c => c.CategoryID,
            (p, c) => p
        )
        .Where(p => p.CategoryID == 1);

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

以上代码为表述清晰加入了一个条件“where p.CategoryID == 1”，即仅返回产品类别 ID 为 1 的所有产品。

## 2. GroupJoin

GroupJoin 操作符常应用于返回“主键对象-外键对象集合”形式的查询，例如“产品类别-此类别下的所有产品”。以下的代码演示了这一点：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //查询语法
    var query =
        from c in db.Categories
        join p in db.Products on c.CategoryID equals p.CategoryID into r
        select new
        {
            c.CategoryName,
            Products = r
        };
}
```



```
//方法语法
var q =
    db.Categories
    .GroupJoin
    (
        db.Products,
        c => c.CategoryID,
        p => p.CategoryID,
        (c, p) => new
        {
            c.CategoryName,
            Products = p
        }
    );

foreach (var item in query)
{
    Console.WriteLine("{0} =>", item.CategoryName);
    foreach (var p in item.Products)
    {
        Console.WriteLine(p.ProductName);
    }
    Console.WriteLine("-----");
}
}
```

## 五、分组操作符

分组是根据一个特定的值将序列中的元素进行分组。LINQ 只包含一个分组操作符：**GroupBy**。

下面的示例中使用了产品表，以 **CategoryID** 作为分组关键值，按照产品类别对产品进行了分组。

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //查询语法
    var query =
        from p in db.Products
        group p by p.CategoryID;

    //方法语法
    var q =
        db.Products
        .GroupBy(p => p.CategoryID);

    foreach (var item in query)
    {
```

```
Console.WriteLine("{0} =>", item.Key);
foreach (var p in item)
{
    Console.WriteLine(p.ProductName);
}
Console.WriteLine("-----");
}
```

执行 **GroupBy** 得到的序列中包含的元素类型为 **IGrouping<TKey?, T>**，其 **Key** 属性代表了分组时使用的关键值，遍历 **IGrouping<TKey?, T>** 元素可以读取到每一个 **T** 类型。在此示例中，对应的元素类型为 **IGrouping<int?, Products>**，其 **Key** 属性即为类别 ID，遍历它可以读取到每一个产品对象。

## 六、串联操作符

串联是一个将两个集合联接在一起的过程。在 LINQ 中，这个过程通过 **Concat** 操作符来实现。

在下面的示例中，将会把类别名称串联在产品名称之后：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中

    //方法语法
    var q =
        db.Products
        .Select(p => p.ProductName)
        .Concat
        (
            db.Categories.Select(c => c.CategoryName)
        );

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

## 七、聚合操作符

聚合函数将在序列上执行特定的计算，并返回单个值，如计算给定序列平均值、最大值等。共有 7 种 LINQ 聚合查询操作符：**Aggregate**、**Average**、**Count**、**LongCount**、**Max**、**Min** 和 **Sum**。

### 1. Aggregate

**Aggregate** 操作符对集合值执行自定义聚合运算。例如，需要列出所有产品类别清单，每个类别名称之间用顿号连接。以下的代码演示了这一过程：

```
using (NorthwindDataContext db = new NorthwindDataContext())
```

```
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中

    //方法语法
    var q =
        db.Categories
        .Select(c => c.CategoryName)
        .ToArray()
        .Aggregate((current, next) => String.Format("{0}、{1}", current, next));

    Console.WriteLine(q);
}
```

如果你对这一过程有些迷惑，那么请参照以下代码：

```
var query =
    db.Categories
    .Select(c => c.CategoryName)
    .ToArray();
string r = String.Empty;
foreach (var item in query)
{
    r += "、";
    r += item;
}
r = r.Substring(1); //去除第一个顿号
Console.WriteLine(r);
```

## 2. Average

求集合中元素的平均值，返回值类型 `double`

## 3. Count

求集合中元素的个数，返回值类型 `Int32`

## 4. LongCount

求集合中元素的个数，返回值类型 `Int64`

## 5. Max

求集合中元素的最大值

## 6. Min

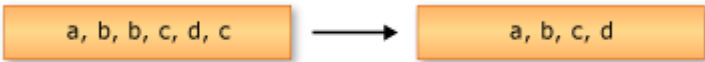

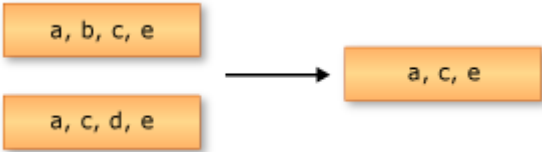
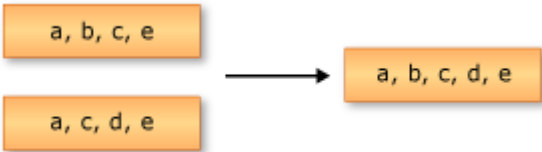
求集合中元素的最小值

## 7. Sum

求集合中元素的和

## 八、集合操作符

LINQ 中的集合操作符是指根据相同或不同集合（或集）中是否存在等效元素来生成结果集的查询操作，一共有 4 种：

方法名	说明
Distinct	从集合移除重复值。
	
Except	返回差集，差集是指位于一个集合但不位于另一个集合的元素。
	
Intersect	返回交集，交集是指同时出现在两个集合中的元素。
	
Union	返回并集，并集是指位于两个集合中任一集合的唯一的元素。
	

使用方式均为“集合 1.方法名(集合 2)”，返回值为运算结果的集合，这里就不演示了。

## 九、生成操作符

生成是指创建新的值序列。

### 1. Empty

Empty 操作符返回一个指定类型的空集合。这里的空不是 null，而是元素数量为 0 的集合。以下的示例演示了如何创建一个 IEnumerable<int>类型的空集合：

```
var q = Enumerable.Empty<int>();  
Console.WriteLine(q == null);  
Console.WriteLine(q.Count());
```

### 2. DefaultIfEmpty

DefaultIfEmpty 将空集合替换为具有默认值的单一实例集合。执行此方法获得的集合将至少含有一个元素，这是因为 DefaultIfEmpty 方法需要两个参数，第一个参数是一个泛型集合，第二个参数是相应类型的单个元素，如果第一个参数中不含有任何元素，它将返回第二个参数指定的单个元素。如果你使用了

`DefaultIfEmpty` 方法的重载方法 `DefaultIfEmpty<T>(IEnumerable<T> array)`，如果指定的 `array` 集合为空，那么将返回一个类型为 `T`，值为 `null` 的单个对象。以下的代码演示了这一过程：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中

    //方法语法
    var q =
        Enumerable.DefaultIfEmpty
        (
            db.Employees
                .Where(e => e.FirstName.StartsWith("Aaf")) //更改此处的条件可获得不同的集合
                , new Employees() { FirstName = "Sunny D.D" }
        );
    Console.WriteLine(q.Count());
    foreach (var item in q)
    {
        Console.WriteLine(item.FirstName);
    }
}
```

### 3. Range

`Range` 操作符用于生成指定范围内的整数的序列。它需要两个参数，第一个参数是序列开始的整数值，第二个参数是序列中整数的数量。下面的示例演示了使用 `Range` 操作符来生成从 0 到 9 的整数序列：

```
var q =
    Enumerable.Range(0, 10);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

### 4. Repeat

`Repeat` 操作符用于生成包含一个重复值的集合。它需要两个参数，第一个参数是任意类型的元素，第二个参数是生成的序列中所包含此元素的数量。下面的示例演示了使用 `Repeat` 来生成一个包含 10 个 0 的序列：

```
var q =
    Enumerable.Repeat(0, 10);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

## 十、转换操作符

转换操作符是用来实现将输入对象的类型转变为序列的功能。名称以“**As**”开头的转换方法可更改源集合的静态类型但不枚举（延迟加载）此源集合。名称以“**To**”开头的方法可枚举（即时加载）源集合并将项放入相应的集合类型。

### 1. AsEnumerable

所有实现了 `IEnumerable<T>` 接口的类型都可以调用此方法来获取一个 `IEnumerable<T>` 集合。此方法一般仅用于实现类中的方法与 `IEnumerable<T>` 接口方法重名时。例如，实现类 `Test` 中有一个 `Where` 方法，当使用 `Test` 对象调用 `Where` 时，将执行 `Test` 自身的 `Where` 方法过程。如果要执行 `IEnumerable<T>` 的 `Where` 方法，便可以使用 `AsEnumerable` 进行转换后，再调用 `Where` 方法即可。当然，将实现类 `Test` 隐式转换为 `IEnumerable<T>` 接口，再调用接口的 `Where` 方法也能达到同样的效果。以下的代码演示了这一过程：

```
class AsEnumerableTest<T> : List<T>
{
    public void Where(Func<T, bool> func)
    {
        Console.WriteLine("AsEnumerableTest的Where方法");
    }
}

public static void AsEnumerable()
{
    AsEnumerableTest<int> q = new AsEnumerableTest<int>() { 1, 2, 3, 4 };
    q.Where(r => r < 3);

    //q.AsEnumerable().Where(r => r < 3);

    //IEnumerable<int> i = q;
    //i.Where(r => r < 3);
}
```

### 2. Cast

`Cast<T>` 方法通过提供必要的类型信息，可在 `IEnumerable`(非泛型)的派生对象上调用 `Cast<T>` 方法来获得一个 `IEnumerable<T>` 对象。例如，`ArrayList` 并不实现 `IEnumerable<T>`，但通过调用 `ArrayList` 对象上的 `Cast<T>()`，就可以使用标准查询运算符查询该序列。

如果集合中的元素无法强制转换为 `T` 类型，则此方法将引发异常。以下代码演示了这一过程：

```
ArrayList array = new ArrayList();
array.Add("Bob");
array.Add("Jack");
array.Add(1);
foreach (var item in array.Cast<string>())
{
    Console.WriteLine(item);
}
```

运行此代码，可以输出“Bob”、“Jack”，然后会报出一个异常“无法将 int 强制转换为 string”，这说明

Cast 方法也是延迟执行实现的，只有在枚举过程中才将对象逐个强制转换为 T 类型。

### 3. OfType

OfType<T> 方法通过提供必要的类型信息，可在 IEnumerable(非泛型)的派生对象上调用 OfType<T> 方法来获得一个 IEnumerable<T>对象。执行 OfType<T>方法将返回集合中强制转换类型成功的所有元素。也就是说，OfType<T>方法与 Cast<T> 方法的区别在于，如果集合中的元素在强制转换失败的时候会跳过，而不是抛出异常。

### 4. ToArray

ToArray 操作符可以在 IEnumerable<T> 类型的任何派生对象上调用，返回值为 T 类型的数组。

### 5. ToDictionary

ToDictionary 操作符根据指定的键选择器函数，从 IEnumerable<T>创建一个 Dictionary<TKey, TValue>。下面的示例中，将查询到的产品类别集合转换为 Dictionary<类别 ID,类别名称>的键-值集合：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //方法语法
    var q =
        db.Categories
        .ToDictionary
        (
            c => c.CategoryID,
            c => c.CategoryName
        );

    foreach (var item in q)
    {
        Console.WriteLine("{0} - {1}", item.Key, item.Value);
    }
}
```

需要注意的是，如果省略 ToDictionary 方法的第二个参数（值选择函数），那么 Value 将会保存一个类别对象。还有，如果 Key 为 null，或者出现重复的 Key，都将导致抛出异常。

### 6. ToList

ToList 操作符可以在 IEnumerable<T> 类型的任何派生对象上调用，返回值为 List<T>类型的对象。

### 7. ToLookup

ToLookup 操作符将创建一个 Lookup<TKey, TElement>对象，这是一个 one-to-many 集合，一个 Key 可以对应多个 Value。以下的示例以产品表的所有数据作为数据源，以类别 ID 作为 Key 调用了 ToLookup 方法，然后遍历返回的 Lookup<TKey, TElement>对象，输出了类别 ID 以及此类别下的所有产品名称：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
```

```
db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
//方法语法
var q =
    db.Products
    .ToLookup
    (
        p => p.CategoryID,
        p => p.ProductName
    );

foreach (var item in q)
{
    Console.WriteLine(item.Key);
    foreach (var p in item)
    {
        Console.WriteLine(p);
    }
}
```

可以看出，ToLookup 操作与 GroupBy 操作很相似，只不过 GroupBy 是延迟加载的，而 ToLookup 是即使加载。

## 十一、元素操作符

元素操作符将从一个序列中返回单个指定的元素。

### 1. First

First 操作将返回序列中的第一个元素。如果序列中不包含任何元素，则 First<T>方法将引发异常。若要在源序列为空时返回默认值，需要使用 FirstOrDefault 方法。以下代码演示了 First<T>方法的使用方式：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //无参
    var query =
        db.Employees
        .First();

    //有参
    var q =
        db.Employees
        .First(e => e.FirstName.StartsWith("S"));

    Console.WriteLine(q.FirstName);
}
```

上述代码中使用了 First<T>方法的无参方式与有参方式。First<T>的有参方式中可以指定一个条件，操



作将返回序列中满足此条件的第一个元素。从查询结果上看，`source.First<T>(条件)`方法与 `source.Where(条件).First<T>()`是一样的，但是需要注意 “`First<T>(条件)`操作将返回序列中满足此条件的第一个元素”，这将忽略后面的遍历操作，效率更高。

## 2. FirstOrDefault

`FirstOrDefault` 方法将返回序列中的第一个元素；如果序列中不包含任何元素，则返回默认值。它也可以像 `First` 方法一样传递一个条件。需要说明的是如果序列中不包含任何元素，返回的默认值是个怎样的元素。在这之前，先来看一下 `FirstOrDefault` 方法是如何实现的：

```
public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source)
{
    if (source == null)
    {
        throw Error.ArgumentNull("source");
    }
    IList<TSource> list = source as IList<TSource>;
    if (list != null)
    {
        if (list.Count > 0)
        {
            return list[0];
        }
    }
    else
    {
        using (IEnumerator<TSource> enumerator = source.GetEnumerator())
        {
            if (enumerator.MoveNext())
            {
                return enumerator.Current;
            }
        }
    }
    return default(TSource);
}
```

1. 如果调用 `FirstOrDefault` 方法的序列为空，抛出异常
2. 如果序列成功转换为 `List<T>`，并且元素数量大于 0，则返回首个元素
3. 如果序列没有成功转换为 `List<T>`，则尝试获取序列的遍历器，然后再调用遍历器的 `MoveNext` 方法，如果返回值为 `true`，则返回当前的元素。
4. 如果上述操作都没有执行，则使用 `default(T)`关键字返回类型 `T` 的默认值

以下给出 MSDN 中，对于 `default(T)`关键字的描述：

在泛型类和泛型方法中产生的一个问题是，在预先未知以下情况时，如何将默认值分配给参数化类型 `T`：

- `T` 是引用类型还是值类型。
- 如果 `T` 为值类型，则它是数值还是结构。

给定参数化类型 `T` 的一个变量 `t`，只有当 `T` 为引用类型时，语句 `t = null` 才有效；只有当 `T` 为数值类型而不是结构时，语句 `t = 0` 才能正常使用。解决方案是使用 `default` 关键字，此关键字对于引用类型会返

回 `null`，对于数值类型会返回零。对于结构，此关键字将返回初始化为零或 `null` 的每个结构成员，具体取决于这些结构是值类型还是引用类型。

### 3. Last

`Last` 方法将返回序列中的最后一个元素。使用方法参照 `First`。

### 4. LastOrDefault

`LastOrDefault` 方法将返回序列中的最后一个元素；如果序列中不包含任何元素，则返回默认值。使用方法参照 `FirstOrDefault`。

### 5. ElementAt

`ElementAt` 方法返回序列中指定索引处的元素。使用方法参照 `First`。需要注意的是如果索引超出范围会导致异常。

### 6. ElementAtOrDefault

`ElementAtOrDefault` 方法将返回序列中指定索引处的元素；如果索引超出范围，则返回默认值。使用方法参照 `FirstOrDefault`。

### 7. Single

`Single` 方法的无参形式将从一个序列中返回单个元素，如果该序列包含多个元素，或者没有元素数为 0，则会引发异常。也就是说，在序列执行 `Single` 方法的无参形式时，必须保证该序列有且仅有一个元素。

`Single` 方法的有参形式将从一个序列中返回符合指定条件的唯一元素，如果有多个元素，或者没有元素符合这一条件，则会引发异常。以下代码演示了 `Single` 的使用方式：

```
using (NorthwindDataContext db = new NorthwindDataContext())
{
    db.Log = Console.Out; //将生成的T-SQL语句输出到控制台中
    //方法语法
    var q =
        db.Employees
        .Single();

    var query =
        db.Employees
        .Single(e => e.FirstName.StartsWith("S"));
    Console.WriteLine(query.FirstName);
}
```

### 8. SingleOrDefault

`SingleOrDefault` 方法的无参形式将从一个序列中返回单个元素。如果元素数为 0，则返回默认值。如果该序列包含多个元素，则会引发异常。

`SingleOrDefault` 方法的有参形式将从一个序列中返回符合指定条件的唯一元素，如果元素数为 0，则返回默认值；如果该序列包含多个元素，则会引发异常。`SingleOrDefault` 的使用方式与 `Single` 相同。

需要注意的是，`Single` 方法与 `SingleOrDefault` 方法都是即时加载的，在代码进行到方法所在位置时，

如果引发了异常，会立刻抛出。

## 十二、相等操作符

如果两个序列的对应元素相等且这两个序列具有相同数量的元素，则视这两个序列相等。

`SequenceEqual` 方法通过并行地枚举两个数据源并比较相应元素来判断两个序列是否相等。如果两个序列完全相等，返回 `true`，否则返回 `false`。以下代码是 `SequenceEqual` 方法的实现过程：

```
public static bool SequenceEqual<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second,
IEqualityComparer<TSource> comparer)
{
    if (comparer == null)
    {
        comparer = EqualityComparer<TSource>.Default;
    }
    if (first == null)
    {
        throw Error.ArgumentNull("first");
    }
    if (second == null)
    {
        throw Error.ArgumentNull("second");
    }
    using (IEnumerator<TSource> enumerator = first.GetEnumerator())
    {
        using (IEnumerator<TSource> enumerator2 = second.GetEnumerator())
        {
            while (enumerator.MoveNext())
            {
                if (!enumerator2.MoveNext() || !comparer.Equals(enumerator.Current, enumerator2.Current))
                {
                    return false;
                }
            }
            if (enumerator2.MoveNext())
            {
                return false;
            }
        }
    }
    return true;
}
```

以上代码的执行过程为：

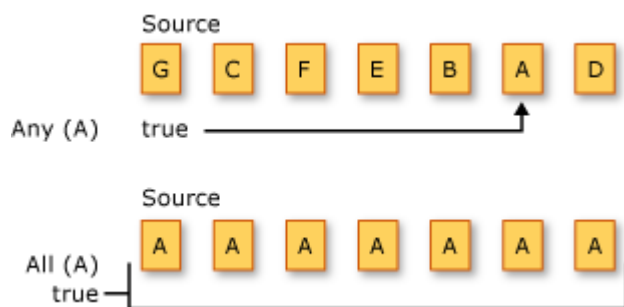
1. 如果比较器为 `null`，赋值为默认值 `EqualityComparer<TSource>.Default`。
2. 如果序列 1 为 `null`，抛出异常。
3. 如果序列 2 为 `null`，抛出异常。

4. 遍历序列 1。在此过程中，如果序列 2 到达底端则返回 **false**；如果序列 1 的当前值与序列 2 的当前值不同，则返回 **false**。
5. 序列 1 遍历完成后，如果序列 2 未到达底端，则返回 **false**。
6. 如果第 2-5 步都没有执行，则返回 **true**。

## 十三、限定操作符

限定符运算返回一个 **Boolean** 值，该值指示序列中是否有一些元素满足条件或是否所有元素都满足条件。

下图描述了两个不同源序列上的两个不同限定符运算。第一个运算询问是否有一个或多个元素为字符“A”，结果为 **true**。第二个运算询问是否所有元素都为字符“A”，结果为 **true**。



### 1. All

**All** 方法用来确定是否序列中的所有元素都满足条件。以下代码演示了 **All** 的用法：

```
string[] source1 = new string[] { "A", "B", "C", "D", "E", "F" };
string[] source2 = new string[] { "A", "A", "A", "A", "A", "A" };

Console.WriteLine(source1.All(w => w == "A")); //console will print "False"
Console.WriteLine(source2.All(w => w == "A")); //console will print "True"
```

### 2. Any

**Any** 方法的无参方式用来确定序列是否包含任何元素。如果源序列包含元素，则为 **true**；否则为 **false**。

**Any** 方法的有参方式用来确定序列中是否有元素满足条件。只要有一个元素符合指定条件即返回 **true**，如果一个符合指定条件的元素都没有则返回 **false**。以下代码演示了 **Any** 方法有参方式的用法：

```
string[] source1 = new string[] { "A", "B", "C", "D", "E", "F" };
string[] source2 = new string[] { "A", "A", "A", "A", "A", "A" };

Console.WriteLine(source1.Any(w => w == "A")); //console will print "True"
Console.WriteLine(source2.Any(w => w == "A")); //console will print "True"
```

### 3. Contains

**Contains** 方法用来确定序列是否包含满足指定条件的元素。如果有返回 **true**，否则返回 **false**。以下代码使用默认的 **String** 比较器来判断序列中是否含有指定的字符串：

```
string[] source1 = new string[] { "A", "B", "C", "D", "E", "F" };

Console.WriteLine(source1.Contains("A")); //console will print "True"
```

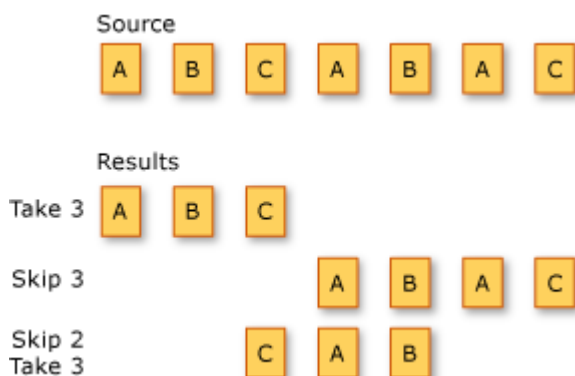
```
Console.WriteLine(source1.Contains("G")); //console will print "False"
```

如果要对序列中的元素进行自定义比较，需要一个 `IEqualityComparer<T>` 接口的实现类作为比较器，用于比较序列中的元素。

## 十四、分区操作符

LINQ 中的分区指的是在不重新排列元素的情况下，将输入序列划分为两部分，然后返回其中一个部分的操作。

下图显示对一个字符序列执行三个不同的分区操作的结果。第一个操作返回序列中的前三个元素。第二个操作跳过前三个元素，返回剩余的元素。第三个操作跳过序列中的前两个元素，返回接下来的三个元素。



### 1. Take

`Take(int n)` 方法将从序列的开头返回数量为 `n` 的连续元素。以下代码演示了从一个序列中返回其前五个元素：

```
int[] source = new int[] { 86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45 };
var q = source.Take(5);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

上述代码的运行结果为下图所示：

```
86
2
77
94
100
请按任意键继续. . .
```

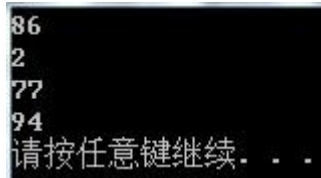
### 2. TakeWhile

`TakeWhile` 方法执行时将逐个比较序列中的每个元素是否满足指定条件，直到碰到不符合指定的条件的元素时，返回前面所有的元素组成的序列。以下代码演示了这一过程：

```
int[] source = new int[] { 86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45 };
var q = source.TakeWhile(i => i < 100);
```

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

上述代码的运行结果为下图所示：



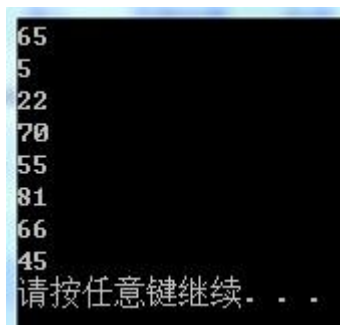
```
86
2
77
94
100
请按任意键继续...
```

### 3. Skip

`Skip(int n)`方法将跳过序列开头的 `n` 个元素，然后返回其余的连续元素。以下代码演示了从一个序列中跳过头五个元素，然后返回其余的元素组成的序列：

```
int[] source = new int[] { 86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45 };
var q = source.Skip(5);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

上述代码的运行结果为下图所示：



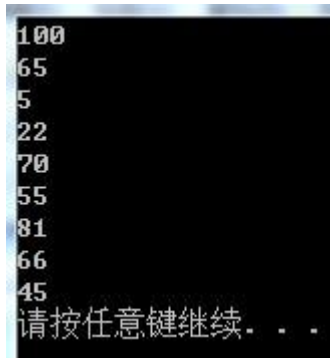
```
65
5
22
70
55
81
66
45
请按任意键继续...
```

### 4. SkipWhile

`SkipWhile` 方法执行时将逐个比较序列中的每个元素是否满足指定条件，直到碰到不符合指定的条件的元素时，返回其余所有的元素组成的序列。以下代码演示了这一过程：

```
int[] source = new int[] { 86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45 };
var q = source.SkipWhile(i => i < 100);
foreach (var item in q)
{
    Console.WriteLine(item);
}
```

上述代码的运行结果为下图所示：



```
100
65
5
22
70
55
81
66
45
请按任意键继续...
```

## 本文总结

本文介绍了 LINQ 标准查询操作符。没有这些操作符，LINQ 就不会存在。本文为理解这些操作符的功能提供了很好的基础。了解它们将会很有帮助，因为 LINQ 的各种 **Provider** 都是基于这些操作符来完成各自丰富的功能。

## 关于作者

Sunny D.D

<http://www.sunnycoder.cn>

<http://www.cnblogs.com/sunnycoder>

[sunny19788989@gmail.com](mailto:sunny19788989@gmail.com)