1、Where　　操作符用于限定输入集合中的元素，将符合条件的元素组织声称一个序列结果。  
2、Select　　 操作符用于根据输入序列中的元素创建相应的输出序列中的元素，输出序列中的元素类型可以与输入序列中的元素类型相同，也可以不同。下面来看看Select方法的原型。  
3、SelectMany　　操作符用于根据输入序列中的每一个元素，在输出序列中创建相应的零个或者多个元素，与Select操作符不同，Select操作符会根据输入序列中的每一个元素创建一个对应的输出序列元素，而SelectMany操作符可以创建多个。  
4、Take　　　　  操作符用于从输入序列中返回指定数量的元素，常用于分页。  
5、TakeWhile　　操作符用于从输入序列中返回指定数量且满足一定条件的元素。  
6、Skip　　　　　操作符用于从输入序列中跳过指定数量的元素，返回由序列中剩余的元素所组成的新序列。  
7、SkipWhile　　操作符用于从输入序列中跳过满足一定条件指定数量的元素，与TakeWhile操作符类似。  
8、Concat　　　 操作符用于连接两个序列，生成一个新序列。  
9、OrderBy　　　操作符用于对输入序列中的元素进行排序，排序基于一个委托方法的返回值顺序，排序过程完成后，会返回一个类型为IOrderEnumerable<T>的集合对象。  
10、OrderByDescending　　操作符的功能与OrderBy操作符基本相同，二者只是排序的方式不同OrderBy是顺序排序，而OrderByDescending则是逆序排序。  
11、ThenBy　　　　　　　　操作符可以对一个类型为IOrderedEnumerable<T>,(OrderBy和OrderByDesceding操作符的返回值类型)的序列再次按照特定的条件顺序排序。  
12、ThenByDescending　　操作符与ThenBy操作符非常类似，只是排序顺序倒过来而已，不在过多阐述。  
13、Reverse　　　　　　操作符用于生成一个与输入序列中元素相同，但元素排列顺序相反的新序列。  
14、Join　　　　　　　　操作符类似于SQL语句中的Join语句用于连接多个表，Linq to OBJECT中Join操作符可以用来连接两个输入序列。  
15、GroupJoin　　操作符也用于连接两个输入序列，但与Join操作符不同稍有不同，Join操作符在列举outer序列元素时，会将一个outer序列元素和其对应的inner序列元素作为一组参数传递给委托resultSelector委托，这就意味着如果某一个outer序列元素有多个对应的inner序列元素，Join操作符将会分多次将outer序列元素和每一个对应的inner序列元素传递给委托resultSelector。使用GroupJoin操作符时，如果某一个outer序列元素有多个对应的inner序列元素，那么这多个对应的inner序列元素会作用一个序列一次性传递给委托resultSelecotr，可以针对此序列添加一些处理逻辑。  
16、GroupBy　　操作符类似于SQL语言仲的Gruop By语句，这里的GroupBy操作符用于将输入序列中的元素进行分组。  
17、Distinct　　  操作符类似于SQL语句中的Distinct语句，这里的Distinct操作符也用于去除一个序列中的重复元素。  
18、Union　　　  操作符用于将两个序列中的元素合并成一个新的序列，新序列将自动去除重复的元素。  
19、Intersect　　操作符会将两个输入序列中的重复元素，即同时存在于两个序列中的元素挑选出来，生成一个新的集合，也就是求交集。  
20、Except　　   操作符可以实现一种集合之间的减法运算，它返回两个序列中存在于第一个序列但不存在于第二个序列的元素所组成的新序列。  
21、Cast　　　　 操作符用于将一个类型为IEnumerable的集合对象转换为IEnumerable<T>类型的集合对象。也就是非泛型集合转成泛型集合，因为在Linq to OBJECT中，绝大部分操作符都是针对IEnumerable<T>类型进行的扩展方法。因此对非泛型集合并不适用。  
22、OfType　　  操作符与Cast操作符类似，用于将类型为IEnumerable的集合对象转换为IEnumerable<T>类型的集合对象。不同的是，Cast操作符会视图将输入序列中的所有元素转换成类型为T的对象，，如果有转换失败的元素存在Cast操作符将抛出一个异常；而OfType操作符仅会将能够成功转换的元素进行转换，并将这些结果添加到结果序列中去。与Cast操作符相比，OfType操作符更加安全。  
23、AsEnumerable　　操作符可以将一个类型为IEnumerable<T>的输入序列转换成一个IEnumerable<T>的输出序列，其主要用于将一个实现了IEnumerable<T>接口的对象转换成一个标准的IEnumerable<T>接口对象。在Linq中、不同领域的Linq实现都有自己专属的操作符。  
24、DefaultEmpty　　操作符可以用来为一个空的输入序列生成一个对应的含有默认元素的新序列。引用类型为null，值类型为相应的默认值。有些标准操作符在一个空的序列上调用时会抛出一个异常，而DefaultEmpty恰恰可以解决这个问题。  
25、Range　　　　操作符用于辅助生成一个整数序列。  
26、Repeat　　　  操作符用于生成一个包含指定数量重复元素的序列。  
27、Empty　　　　操作符用于生成一个包含指定类型元素的空序列。

Linq to OBJECT是用于操作内存对象的LINQ编程接口，包含了大量的查询操作符，针对内存中的集合对象进行操作。

Linq to OBJECT的实现基于IEnumerable<T>、序列(sequences)以及标准查询操作符(Standard Query Operators)等基本概念。标准查询操作符本质上是一些扩展方法(Extension Methods)，这些扩展方法定义在静态类System.Linq.Enumerable中，其原型的第一个参数(带this修饰符的参数)是IEnumerable<T>类型。由于这些方法都是扩展方法，他们可以在IEnumerable<T>的实例对象上直接调用。

下面我们来看下Linq to OBJECT上经常使用的一个Where扩展方法(这个在Linq中成为操作符)的方法签名

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

从方法签名中，我们能得到哪些信息呢？

1、这是一个扩展方法，扩展的是IEnumerable<T>接口。这就是说，实现了这个接口，或者继承了这个接口的实例都能够使用此扩展方法。而IEnumerable<T>是比较底层的一个集合接口，很多集合 、集合接口和数组都实现或继承了该接口。因此这些类和接口的实例都能用。

2、这是泛型方法，Where<T>表明，你可以根据自己的类型传入不同的类型参数。各种类型都能够使用。

3、方法参数是什么东西呢？我们转到定义之后看到如下代码：

[TypeForwardedFrom("System.Core, Version=3.5.0.0, Culture=Neutral, PublicKeyToken=b77a5c561934e089")]

public delegate TResult Func<in T, out TResult>(T arg);

很明显这是一个泛型委托。也就是说，我们可以用匿名方法或lambda表达式为该泛型委托赋值。

下面来演示下该方法的使用：

[IMG_256](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

class Program

{

static void Main(string[] args)

{

List<int> listInt = new List<int>(); //List<T>类实现了IEnumerable<T>接口，因此它也可以使用IEnumerable<T>的扩展方法

listInt.Add(1);

listInt.Add(4);

listInt.Add(5);

listInt.Add(12);

listInt.Add(15);

//外部定义方法为委托赋值

IEnumerable<int> IEnum1 = listInt.Where<int>(GetBiggerThanTen);

foreach (int i in IEnum1)

{

Console.WriteLine(i); //输出12 15 }

Console.WriteLine("==============================");

//匿名方法为委托赋值

IEnumerable<int> IEnum2 = listInt.Where<int>(

delegate(int input)

{

if (input > 10)

{

return true;

}

return false;

});

foreach (int i in IEnum2)

{

Console.WriteLine(i); //输出12 15 }

Console.WriteLine("==============================");

//Lambda表达式为委托赋值

IEnumerable<int> IEnum3 = listInt.Where<int>(m => m>10);

foreach (int i in IEnum3)

{

Console.WriteLine(i); //输出12 15 }

Console.WriteLine("==============================");

//以上3行代码相当于复习了委托的相关知识，用不同的方式为委托赋值，就是想说明参数是一个委托

//但如果真的像上面书写那样复杂，Linq就失去了它的意义，以下来看看最简单的书写形式

var Arr = listInt.Where(m => m>10);

foreach (var i in Arr)

{

Console.WriteLine(i); //输出12 15 同样的效果 }

Console.ReadKey();

}

public static bool GetBiggerThanTen(int input)

{

if (input > 10)

{

return true;

}

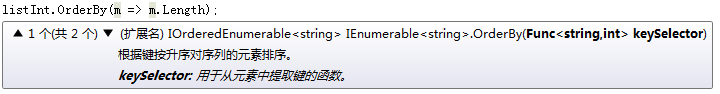
return false;

}

}

[IMG_257](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 通常我们将鼠标放到扩展方面签名上，看到智能代码提示为一个委托的时候，应该仔细解读该委托。例如：



例如以上这个OrderBy这个操作符要求传入一个委托，通常左边是输入，右边是输出（如果不确定，可以转到定义确定下）。例如此方法左边输入是string，右边输出是整数。注意只能提示会根据OBJECT的类型动态改变，因此给出的一定是最准确的提示。通常左边都是集合中的元素的类型。右边根据不同的方法变化(本身是泛型)。

案例如下：

[IMG_259](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<string> listInt = new List<string>(); //List<T>类实现了IEnumerable<T>接口，因此它也可以使用IEnumerable<T>的扩展方法

listInt.Add("你好");

listInt.Add("吗");

listInt.Add("我爱你");

listInt.Add("真的好想家了");

listInt.Add("哎");

IEnumerable<string> stringArr = listInt.OrderBy(m => m.Length);

foreach (var item in stringArr)

{

Console.WriteLine(item); //输出 吗 哎 你好 我爱你 真的好想家了 }

Console.ReadKey();

}

[IMG_260](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 下面来说下Linq中的延时查询(Deferred Query)特性。Linq的延时特性表示，在用的时候Linq才去查。如下面这个例子。

[IMG_261](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>(); //List<T>类实现了IEnumerable<T>接口，因此它也可以使用IEnumerable<T>的扩展方法

listInt.Add(1);

listInt.Add(2);

listInt.Add(3);

IEnumerable<int> IEnumInt = listInt.Select(m => m); //Linq查询到的对象给IEnumInt集合赋值

foreach (int i in IEnumInt)

{

Console.Write(i + " "); //输出 1 2 3 }

Console.WriteLine();

listInt[0] = 4;

listInt[1] = 5;

listInt[2] = 6;

foreach (int i in IEnumInt) //按道理说，我们的程序没有改变过IEnumerInt，理论上是不会更新IEnumerInt这个集合中的值 {

Console.Write(i + " "); //但当改变原集合中的值再遍历没有更改过得IEnumInt时，输出 4 5 6。注意IEnumInt这个集合没有代码改变过。

} //这就是Linq的延时查询，在用的时候才去查

Console.ReadKey();

}

[IMG_262](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

当然，如果我们不想要这类延时查询，可以在查询表达式上调用一些转换方法，如ToArray，ToList以及ToDictionary等方法来缓存结果数据。

[IMG_263](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(2);

listInt.Add(3);

IEnumerable<int> IEnumInt = listInt.Select(m => m).ToArray(); //此处加个ToArray()，查询立即执行，执行结果保存在IEnumerInt集合中

foreach (int i in IEnumInt)

{

Console.Write(i + " "); //输出 1 2 3 }

Console.WriteLine();

listInt[0] = 4;

listInt[1] = 5;

listInt[2] = 6;

foreach (int i in IEnumInt)

{

Console.Write(i + " "); //输出1 2 3 留意到1 2 3保存了起来，即使改变原集合再遍历也不会改为最新的数据 }

Console.ReadKey();

}

[IMG_264](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 下面来介绍Linq提供的操作符。

一、延时标准查询操作符。

1、Where操作符。

　　Where操作符用于限定输入集合中的元素，将符合条件的元素组织声称一个序列结果。

 　  Where方法原型有两个，如下所示：

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

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);

　　扩展方法的概念就不说了，来看委托参数，此委托接受的方法为，接受任何一个类型的参数，当然，这是根据你的泛型集合的类型确定的，第二个参数表示输出，即当每一个元素调用赋给委托的方法返回值为true时，就能添加该元素到结果序列对象中。

　　而第二个方法的委托参数多了个int参数，该参数表示元素的下标，下标从0开始，看示例：

[IMG_265](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(5);

listInt.Add(11);

listInt.Add(20);

listInt.Add(25);

IEnumerable<int> IEInt = listInt.Where((m, p) => m > 10 && p > 2); //m表示输入元素，p表示元素下标

foreach (int i in IEInt) //整个表达式的意思是，下标大于2，且元素值大于10 {

Console.WriteLine(i); //输出20 25 }

Console.ReadKey();

}

[IMG_266](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

2、Select操作符

　　Select操作符用于根据输入序列中的元素创建相应的输出序列中的元素，输出序列中的元素类型可以与输入序列中的元素类型相同，也可以不同。下面来看看Select方法的原型。

public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, int, TResult> selector);

public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector);

Linq的东西大致相同，Select，Select的是元素的任意值或属性。委托里的参数两个的话表示输入和输出，注意到都是泛型。而第一个int还是表示下标。

Select方法的作用，很明显，输入是T，输出是S。两个都是泛型，输入就肯定是集合的元素了，输出搞成泛型的作用就是让你能够输出任意类型的数据，例如元素的某个属性啊等等，或自定义一个匿名类型。而Where的输出参数设置成bool，目的在于它仅仅用于判断，判断哪个元素复合条件，还是返回元素。

[IMG_267](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

class Program

{

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(1, "关羽", 22);

People p3 = new People(1, "刘备", 23);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

IEnumerable<string> IEP = pList.Select(p => p.Name); //查询的是元素的姓名属性

foreach (string str in IEP)

{

Console.WriteLine(str); //输出张飞 关羽 刘备 }

//此处必须使用var，因为是匿名类型，无法写成IEnumerable<T>的形式

var newList = pList.Select((m, i) => new { index = i, m.Name }); //输入元素与下标，返回一个新的匿名类型的集合

foreach (var item in newList)

{

Console.Write(item.Name); //输出 张飞关羽刘备 }

Console.ReadKey();

}

}

public class People

{

public People(int id, string name, int age)

{

this.Id = id;

this.Name = name;

this.Age = age;

}

public int Id

{

get;

set;

}

public string Name

{

get;

set;

}

public int Age

{

get;

set;

}

}

[IMG_268](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 3、SelectMany操作符

　　SelectMany操作符用于根据输入序列中的每一个元素，在输出序列中创建相应的零个或者多个元素，与Select操作符不同，Select操作符会根据输入序列中的每一个元素创建一个对应的输出序列元素，而SelectMany操作符可以创建多个。

　　下面来看下SelectMany操作符的方法原型：

public static IEnumerable<TResult> SelectMany<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, IEnumerable<TResult>> selector);

public static IEnumerable<TResult> SelectMany<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, int, IEnumerable<TResult>> selector);

public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(this IEnumerable<TSource> source, Func<TSource, IEnumerable<TCollection>> collectionSelector, Func<TSource, TCollection, TResult> resultSelector);

public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(this IEnumerable<TSource> source, Func<TSource, int, IEnumerable<TCollection>> collectionSelector, Func<TSource, TCollection, TResult> resultSelector);

来看示例

[IMG_269](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(1, "关羽", 22);

People p3 = new People(1, "刘备", 23);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

var newList = pList.SelectMany(p => p.Name); //newList是一个包含所有p.Name的字符的集合IEnumerable<char>

foreach (var item in newList) //Select是一个元素返回一个字符串，而SelectMany是一个元素返回多个字符 {

Console.Write(item); //输出 张飞关羽刘备 }

Console.WriteLine();

var items = pList.SelectMany((p, i) => i < 2 ? p.Name.ToArray() : new char[] { }); //前两个元素才转成字符输出

foreach (var i in items)

{

Console.Write(i); //输出 张飞关羽 }

Console.ReadKey();

}  
　　　　　//Person类与上面例子一样，省略

[IMG_270](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

**Select与SelectMany的区别说明**  
　　Select()每一次遍历，输出的是T，然后将所有遍历后得到的T组合成一个IEnumerable<T>。  
　　SelectMany()每遍历一次，输出的是IEnumerable<T>，然后合并成一个大的IEnumerable<T>。  
　　假如举个例子：

　　Select(p => p.Name); //遍历IEnumerable<Person>，返回string类型的Name

　　Selectmany(p => p.Name); //每次返回一个IEnumerable<char>，然后合并成一个大IEnumerable<char>。

　　通常来说，由于selectmany强制每次遍历返回IEnumerable<T>，因此一般用于嵌套的集合中，如：

[IMG_271](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

List<List<int>> numbers = new List<List<int>>()

{

new List<int>{1,2,3},

new List<int>{4,5,6},

new List<int>{7,8,9}

};

IEnumerable<int> result = numbers.SelectMany(collection => collection); //collection的类型是List<int>，要求返回IEnumerable<int>

foreach(int i in result)

{

Console.WriteLine(i);

}

[IMG_272](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

　　输出1,2,3,4,5,6,7,8,9

4、Take操作符

　　Take操作符用于从输入序列中返回指定数量的元素，常用于分页。

　　来看下Take操作符的方法原型。

public static IEnumerable<TSource> Take<TSource>(this IEnumerable<TSource> source, int count);

　　该方法只接受一个整数，表示要返回的结果的数量。

[IMG_273](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(1, "关羽", 22);

People p3 = new People(1, "刘备", 23);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

IEnumerable<People> newList = pList.Take(2); //只取前两个结果

foreach (var item in newList)

{

Console.WriteLine(item.Name); //返回 张飞 关羽 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_274](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

5、TakeWhile操作符

　　TakeWhile操作符用于从输入序列中返回指定数量且满足一定条件的元素。

　　来看下TakeWhile的方法原型：

public static IEnumerable<TSource> TakeWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static IEnumerable<TSource> TakeWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);

　　当TakeWhile操作符被调用时，会将source序列中的每一个元素顺序传递给委托predicate，只有那些使得predicate返回值为true的元素才会被添加到结果序列中。要特别注意的是，当TakeWhile操作符在查找过程中，遇到第一个返回false的元素就会立即停止执行，跳出，不管后面还有没有符合条件的元素，即使后面有符合条件的元素也不会要了。对于第二个扩展方法，委托里面含有一个int类型的参数，该参数代表集合的下标，可以依据此进行一些据下标操作的逻辑。

代码示例：

[IMG_275](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(1, "关羽", 22);

People p3 = new People(1, "刘备", 23);

People p4 = new People(1, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.TakeWhile(p => p.Age<23); //返回前两个,注意TakeWhile操作符中，当遇到第一个返回false的元素就跳出执行

foreach (var item in newList) //不管后面还有没有符合条件的元素 {

Console.WriteLine(item.Name); //返回 张飞 关羽 //遇到刘备，返回false，则跳出，不会再执行到孔明。 }

Console.WriteLine();

IEnumerable<People> newList1 = pList.TakeWhile((p,i) => p.Age<23 && i<1); //年龄小于23且下标小于1

foreach (People p in newList1)

{

Console.WriteLine(p.Name); //输出张飞 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_276](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 6、Skip操作符

　　Skip操作符用于从输入序列中跳过指定数量的元素，返回由序列中剩余的元素所组成的新序列。

　　来看下Skip的方法原型：

public static IEnumerable<TSource> Skip<TSource>(this IEnumerable<TSource> source, int count);

　　可以看到，该扩展方法只接受一个整形的参数，表示跳过的元素数量。

　　代码示例：

[IMG_277](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(1, "关羽", 22);

People p3 = new People(1, "刘备", 23);

People p4 = new People(1, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.Skip(2); //跳过前两个

foreach (People p in newList)

{

Console.WriteLine(p.Name); //输出 刘备 孔明 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_278](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 7、SkipWhile操作符

　　SkipWhile操作符用于从输入序列中跳过满足一定条件指定数量的元素，与TakeWhile操作符类似。

　　来看下SkipWhile操作符的方法原型：

public static IEnumerable<TSource> SkipWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static IEnumerable<TSource> SkipWhile<TSource>(this IEnumerable<TSource> source, Func<TSource, int, bool> predicate);

　　当SkipWhile操作符被调用时，会将输入序列中的元素走位参数传递给委托predicate，只要predicate的返回值为true，该元素就会被跳过，继续下一个元素，直到遇到一个使predicate返回值为false的元素，此元素以及输入序列中剩余的元素将组合一个新的序列返回。注意后面的不再判断，直接添加到返回序列。

　　第二个扩展方法的委托里的参数多了个int，还是代表下标，可以根据下标做一些逻辑处理。

[IMG_279](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.SkipWhile(p => p.Age < 23); //跳过年龄小于23的，直到遇到刘备不在小于23，后面的不再判断，也添加到返回序列

foreach (People p in newList)

{

Console.WriteLine(p.Name); //输出 刘备 孔明 }

Console.WriteLine();

IEnumerable<People> newList1 = pList.SkipWhile((p, i) => p.Age<22 && i < 2); //跳过年龄小于22且下标小于2的，到关羽时不符合，因此关羽与后面的组合返回

foreach (People p in newList1)

{

Console.WriteLine(p.Name); //输出 关羽 刘备 孔明 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_280](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

8、Concat操作符

　　Concat操作符用于连接两个序列，生成一个新序列。

　　来看下此方法的方法原型：

public static IEnumerable<TSource> Concat<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second);

　　第二个参数为输入一个新的集合，与调用集合连接，生成并返回一个新序列。代码示例：

[IMG_281](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

List<People> pList1 = new List<People>();

People p5 = new People(5,"曹操",25);

pList1.Add(p5);

IEnumerable<People> newList = pList.Concat(pList1);

foreach (var item in newList)

{

Console.WriteLine(item.Name); //输出 张飞 关羽 刘备 孔明 曹操 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_282](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 9、OrderBy操作符

　　OrderBy操作符用于对输入序列中的元素进行排序，排序基于一个委托方法的返回值顺序，排序过程完成后，会返回一个类型为IOrderEnumerable<T>的集合对象。其中IOrdernumerable<T>接口继承自IEnumerable<T>接口。下面来看看OrderBy的方法原型：

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector, IComparer<TKey> comparer);

[IMG_283](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

class Program

{

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.OrderBy(p => p.Age);

foreach (var item in newList)

{

Console.WriteLine(item.Name); //输出 张飞 孔明 关羽 刘备 (注意顺序) }

Console.WriteLine();

AgeComparer ac = new AgeComparer();

IEnumerable<People> newList1 = pList.OrderBy(p => p.Age, ac);

foreach (var item in newList1)

{

Console.WriteLine(item.Name); //输出 刘备 关羽 张飞 孔明 (注意顺序) }

Console.ReadKey();

}

//People类与上个一样，省略 }

public class AgeComparer : IComparer<int>

{

public int Compare(int a1, int a2)

{

if (a1 > a2)

{

return (-1); //表示a1 > a2 }

else if (a1 < a2)

{

return (1); //表示a1 < a2 }

else

{

return (0); //表示a1=a2 }

}

}

[IMG_284](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

10、OrderByDescending操作符

　　OrderByDescending操作符的功能与OrderBy操作符基本相同，二者只是排序的方式不同OrderBy是顺序排序，而OrderByDescending则是逆序排序。

　　来看方法原型

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector, IComparer<TKey> comparer);

与OrderBy一样，只是排序顺序倒过来了，来看示例：

[IMG_285](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.OrderBy(p => p.Age); //顺序

foreach (var item in newList)

{

Console.WriteLine(item.Name); //输出 张飞 孔明 关羽 刘备 (注意顺序) }

IEnumerable<People> newList1 = pList.OrderByDescending(p => p.Age); //倒序

foreach (var item in newList1)

{

Console.WriteLine(item.Name); //输出 刘备 关羽 张飞 孔明 (注意顺序) }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_286](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 11、ThenBy操作符

　　 ThenBy操作符可以对一个类型为IOrderedEnumerable<T>,(OrderBy和OrderByDesceding操作符的返回值类型)的序列再次按照特定的条件顺序排序。

　　 下面来看ThenBy的方法原型：

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector, IComparer<TKey> comparer);

可以看到此方法扩展的是IOrderedEnumerable<T>，因此ThenBy操作符长常常跟在OrderBy和OrderByDesceding之后。

[IMG_287](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.OrderBy(p => p.Age).ThenByDescending(p => p.Id); //顺序

foreach (var item in newList)

{

Console.WriteLine(item.Name); //没用ThenBy原本是输出 张飞 孔明 关羽 刘备 (注意顺序)，但

//但是本处在ThenByDescending之后，实际输出的是孔明 张飞 关羽 刘备 ，孔明张飞年龄相同，再按Id倒序排序 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_288](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 12、ThenByDescending操作符

　　ThenByDescending操作符与ThenBy操作符非常类似，只是排序顺序倒过来而已，不在过多阐述。

13、Reverse操作符

　　Reverse操作符用于生成一个与输入序列中元素相同，但元素排列顺序相反的新序列。

　　方法原型：

public static IEnumerable<TSource> Reverse<TSource>(this IEnumerable<TSource> source);

　　从方法原型可以看到，这个扩展方法，不需要输入参数，返回一个新集合。

　　来看实例:

[IMG_289](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

IEnumerable<People> newList = pList.Reverse<People>(); //此处用的是List<T>，必须要写类型参数，因为List<T>本身也有个同名方法

foreach (People p in newList)

{

Console.WriteLine(p.Name); //输出 孔明 刘备 关羽 张飞 注意到顺序是与原集合顺序倒过来的。 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_290](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 14、Join操作符

　　  Join操作符类似于SQL语句中的Join语句用于连接多个表，Linq to OBJECT中Join操作符可以用来连接两个输入序列。

　　  来看方法原型：

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, TInner, TResult> resultSelector);

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, TInner, TResult> resultSelector, IEqualityComparer<TKey> comparer);

　　可以看到Join操作符的方法原型非常复杂，从方法原型可以看到，参数outer和inner是需要连接的两个输入集合。其中outer代表的是调用的集合。当Join操作符被调用时，首先列举inner序列中的所有元素，为序列中每一个类型为U的元素调用委托InnerKeySelector，生成一个类型为K的的对象innerKey作为连接关键字。(相当于数据库中的外键)，将inner序列中的每一个元素和其对应的连接关键字innerKey存储在一个临时哈希表中；其次列举outer序列中的所有元素，为每一个类型为T的元素调用委托outerKeySelector，生成一个类型为K的对象outKey用作连接关键字，在第一步生成的临时哈希表中查找与outKey相等的对应的innerKey对象，如果找到对应的记录，会将当前outer序列中的类型为T的元素和对应的inner序列中类型为U的元素作为一组参数传递给委托resultSelector，resultSelector会根据这两个参数返回一个类型为V的对象，此类型为V的对象会被添加到Join操作符的输出结果序列中去。Join操作符返回一个类型为IEnumerable<T>的序列。

　　来看代码示例：

[IMG_291](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

class Program

{

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

List<Record> rList = new List<Record>();

Record r1 = new Record(1, 3);

Record r2 = new Record(2, 5);

Record r3 = new Record(3, 7);

Record r4 = new Record(4, 20);

rList.Add(r1);

rList.Add(r2);

rList.Add(r3);

rList.Add(r4);

//下面一行代码的意思是，将PList集合拼接rList集合，根据People的Id与Record的PId属性(相当于外键)拼接，取People的姓名与Record的WarRecord返回

var Items = pList.Join(rList, p => p.Id, r => r.PId, (p, r) => new { Name = p.Name, WarRecord = r.WarRecord });

foreach (var item in Items)

{

Console.WriteLine(item.Name + ":" + item.WarRecord); //输出 张飞：3 关羽：5 刘备：7，孔明：20 }

Console.ReadKey();

}

//People类与上个一样，省略 }

public class Record

{

public Record(int id, int warRecord)

{

this.PId = id;

this.WarRecord = warRecord;

}

public int PId

{

get;

set;

}

public int WarRecord

{

get;

set;

}

}

public class People

{

public People(int id, string name, int age)

{

this.Id = id;

this.Name = name;

this.Age = age;

}

public int Id

{

get;

set;

}

public string Name

{

get;

set;

}

public int Age

{

get;

set;

}

}

[IMG_292](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 15、GroupJoin操作符

　　GroupJoin操作符也用于连接两个输入序列，但与Join操作符不同稍有不同，Join操作符在列举outer序列元素时，会将一个outer序列元素和其对应的inner序列元素作为一组参数传递给委托resultSelector委托，这就意味着如果某一个outer序列元素有多个对应的inner序列元素，Join操作符将会分多次将outer序列元素和每一个对应的inner序列元素传递给委托resultSelector。使用GroupJoin操作符时，如果某一个outer序列元素有多个对应的inner序列元素，那么这多个对应的inner序列元素会作用一个序列一次性传递给委托resultSelecotr，可以针对此序列添加一些处理逻辑。

　　来看看GroupJoin操作符的方法原型：

public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);

public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>, TResult> resultSelector, IEqualityComparer<TKey> comparer);

留意变红的那个委托，注意，与Join操作符的不同点就是一个outer序列中如果有多个对应的inner序列元素，会作为一个集合IEnumerable<TInner>传递到此委托。

来看代码示例：

[IMG_293](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

class Program

{

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

List<Record> rList = new List<Record>();

Record r1 = new Record(1, 3);

Record r2 = new Record(2, 5);

Record r3 = new Record(3, 7);

Record r4 = new Record(4, 20);

Record r5 = new Record(1, 33);

rList.Add(r1);

rList.Add(r2);

rList.Add(r3);

rList.Add(r4);

rList.Add(r5);

//下面一行代码的意思是，将PList集合拼接rList集合，根据People的Id与Record的PId属性(相当于外键)拼接，取People的姓名与Record的WarRecord

var Items = pList.Join(rList, p => p.Id, r => r.PId, (p, r) => new { Name = p.Name, WarRecord = r.WarRecord });

foreach (var item in Items)

{

Console.WriteLine(item.Name + ":" + item.WarRecord); //输出 张飞：3 张飞：33 关羽：5 刘备：7，孔明：20 }

Console.WriteLine();

var Items1 = pList.GroupJoin(rList, p => p.Id, r => r.PId, (p, List1) => new { Name = p.Name, WarRecords = List1.Sum(r=>r.WarRecord) });

foreach (var item in Items1)

{

Console.WriteLine(item.Name + ":" + item.WarRecords); //输出 张飞：36 关羽：5 刘备：7，孔明：20 }

//这个例子足以说明，Join与GrouyJoin的不同，Join每次都会传递一个元素到输出序列，而GruopJoin会将相同序列序号作为一个集合的形式传递给输出委托 Console.ReadKey();

}

//People类与上个一样，省略 }

public class Record

{

public Record(int id, int warRecord)

{

this.PId = id;

this.WarRecord = warRecord;

}

public int PId

{

get;

set;

}

public int WarRecord

{

get;

set;

}

}

public class People

{

public People(int id, string name, int age)

{

this.Id = id;

this.Name = name;

this.Age = age;

}

public int Id

{

get;

set;

}

public string Name

{

get;

set;

}

public int Age

{

get;

set;

}

}

[IMG_294](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 16、GroupBy操作符

　　  GroupBy操作符类似于SQL语言仲的Gruop By语句，这里的GroupBy操作符用于将输入序列中的元素进行分组。

　　  来看看GruopBy的方法原型：

[IMG_295](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<People> pList = new List<People>();

People p1 = new People(1, "张飞", 21);

People p2 = new People(2, "关羽", 22);

People p3 = new People(3, "刘备", 23);

People p4 = new People(4, "孔明", 21);

pList.Add(p1);

pList.Add(p2);

pList.Add(p3);

pList.Add(p4);

var pList1 = pList.GroupBy(p=>p.Age);

foreach (var item in pList1)

{

Console.Write(item.Key); //输出 21 22 23 可以看出按照年龄分成了三组

foreach (var item1 in item)

{

Console.Write(item1.Name); //输出21 张飞 孔明 22关羽 24刘备 }

Console.WriteLine();

}

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_296](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

17、Distinct操作符

　　Distinct操作符类似于SQL语句中的Distinct语句，这里的Distinct操作符也用于去除一个序列中的重复元素。

　　来看方法原型：

public static IEnumerable<TSource> Distinct<TSource>(this IEnumerable<TSource> source);

public static IEnumerable<TSource> Distinct<TSource>(this IEnumerable<TSource> source, IEqualityComparer<TSource> comparer);

　　代码示例：

[IMG_297](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(1);

listInt.Add(2);

listInt.Add(2);

listInt.Add(2);

IEnumerable<int> IEInt = listInt.Distinct();

foreach (var i in IEInt)

{

Console.WriteLine(i); //输出 1 2 留意到重复的只出现一次 }

Console.ReadKey();

}

[IMG_298](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 18、Union操作符

　　Union操作符用于将两个序列中的元素合并成一个新的序列，新序列将自动去除重复的元素。

　　Union操作符与Concat操作符的区别在于,Union会自动去除重复元素，而Concat不管有没有重复，所有都输出。

　　来看方法原型：

public static IEnumerable<TSource> Union<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second);

public static IEnumerable<TSource> Union<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second, IEqualityComparer<TSource> comparer);

 　  来看代码示例：

[IMG_299](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(2);

listInt.Add(3);

List<int> listInt1 = new List<int>();

listInt1.Add(2);

listInt1.Add(3);

listInt1.Add(4);

IEnumerable<int> IEInt = listInt.Concat(listInt1);

foreach (var i in IEInt)

{

Console.WriteLine(i); //输出 1 2 3 2 3 4 }

Console.WriteLine();

IEnumerable<int> IEInt1 = listInt.Union(listInt1);

foreach (var i in IEInt1)

{

Console.WriteLine(i); //输出 1 2 3 4 }

//这个例子足以说明两种连接的不同 Console.ReadKey();

}

[IMG_300](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 19、Intersect操作符

　　Intersect操作符会将两个输入序列中的重复元素，即同时存在于两个序列中的元素挑选出来，生成一个新的集合，也就是求交集。

 　  来看方法原型：

public static IEnumerable<TSource> Intersect<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second);

　　代码示例：

[IMG_301](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(2);

listInt.Add(3);

List<int> listInt1 = new List<int>();

listInt1.Add(2);

listInt1.Add(3);

listInt1.Add(4);

IEnumerable<int> IEInt = listInt.Intersect(listInt1);

foreach (var i in IEInt)

{

Console.WriteLine(i); //输出 2 3 //2 3是两个集合的交集 }

Console.ReadKey();

}

//People类与上个一样，省略

[IMG_302](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

20、Except操作符

　　 Except操作符可以实现一种集合之间的减法运算，它返回两个序列中存在于第一个序列但不存在于第二个序列的元素所组成的新序列。

　　 来看方法原型：

public static IEnumerable<TSource> Except<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second);

public static IEnumerable<TSource> Except<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second, IEqualityComparer<TSource> comparer);

　　代码示例：

[IMG_303](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<int> listInt = new List<int>();

listInt.Add(1);

listInt.Add(2);

listInt.Add(3);

List<int> listInt1 = new List<int>();

listInt1.Add(2);

listInt1.Add(3);

listInt1.Add(4);

IEnumerable<int> IEInt = listInt.Except(listInt1);

foreach (var i in IEInt)

{

Console.WriteLine(i); //输出 1` 只有1存在于第1个集合而不存在于第二个集合 }

Console.ReadKey();

}

[IMG_304](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

21、Cast操作符

　　Cast操作符用于将一个类型为IEnumerable的集合对象转换为IEnumerable<T>类型的集合对象。也就是非泛型集合转成泛型集合，因为在Linq to OBJECT中，绝大部分操作符都是针对IEnumerable<T>类型进行的扩展方法。因此对非泛型集合并不适用。

　　来看方法原型，注意，此方法必须传入类型参数。

public static IEnumerable<TResult> Cast<TResult>(this IEnumerable source);

　　来看代码示例：

[IMG_305](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

ArrayList al = new ArrayList();

al.Add(1);

al.Add(2);

al.Add(3);

IEnumerable<int> IEInt = al.Cast<int>(); //非泛型转泛型

foreach (var i in IEInt)

{

Console.WriteLine(i); //输出 1 2 3 }

Console.ReadKey();

}

[IMG_306](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

 22、OfType操作符

　　OfType操作符与Cast操作符类似，用于将类型为IEnumerable的集合对象转换为IEnumerable<T>类型的集合对象。不同的是，Cast操作符会视图将输入序列中的所有元素转换成类型为T的对象，，如果有转换失败的元素存在Cast操作符将抛出一个异常；而OfType操作符仅会将能够成功转换的元素进行转换，并将这些结果添加到结果序列中去。与Cast操作符相比，OfType操作符更加安全。

　　来看下OfType操作符的方法原型：

　　　　public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source);

　　代码示例：

[IMG_307](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

ArrayList al = new ArrayList();

al.Add(1);

al.Add(2);

al.Add("a");

//IEnumerable<int> IECast = al.Cast<int>(); //抛出异常

//foreach (var i in IECast)

//{

// Console.WriteLine(i);

//}

IEnumerable<int> IEOfType = al.OfType<int>();

foreach (int i in IEOfType)

{

Console.WriteLine(i); //输出 1 2 其中转换不了的a不转换 }

Console.ReadKey();

}

[IMG_308](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

23、AsEnumeralbe操作符

　　AsEnumerable操作符可以将一个类型为IEnumerable<T>的输入序列转换成一个IEnumerable<T>的输出序列，其主要用于将一个实现了IEnumerable<T>接口的对象转换成一个标准的IEnumerable<T>接口对象。在Linq中、不同领域的Linq实现都有自己专属的操作符。

　　例如IQueryable<T>通常是Linq to SQL的返回类型，当我们直接在上面调用Where<>方法时，调用的是Linq to sql的扩展方法，因此有时候我们需要转换为标准的IEnumerable<T>在能调用Linq to OBJECT的扩展方法。

　　来看方法原型：

　　public static IEnumerable<TSource> AsEnumerable<TSource>(this IEnumerable<TSource> source);

24、DefaultEmpty操作符

　　DefaultEmpty操作符可以用来为一个空的输入序列生成一个对应的含有默认元素的新序列。引用类型为null，值类型为相应的默认值。有些标准操作符在一个空的序列上调用时会抛出一个异常，而DefaultEmpty恰恰可以解决这个问题。

　　方法原型：

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(this IEnumerable<TSource> source);

public static IEnumerable<TSource> DefaultIfEmpty<TSource>(this IEnumerable<TSource> source, TSource defaultValue);

　　代码示例：

[IMG_309](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

List<string> ListInt = new List<string>();

ListInt.Add("one");

ListInt.Add("two");

ListInt.Add("three");

string str = ListInt.Where(s => s.StartsWith("a")).DefaultIfEmpty().First();

Console.WriteLine(str); //什么也不输出，或者说输出空白

//string str1 = ListInt.Where(s => s.StartsWith("a")).First(); //如果去掉DefaultEmpty就会报异常("序列中不包含任何元素")

Console.ReadKey();

}

[IMG_310](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

25、Range操作符

　　Range操作符用于辅助生成一个整数序列。

　　其方法原型如下：

　　　　public static IEnumerable<int> Range(int start, int count);

　　从方法原型可以看出，这并不是一个扩展方法，只是一个普通的静态方法，其中第一个参数表示开始的数字，第二个是要生成的数量，返回一个IEnumerable<ine>的集合。

　　代码示例如下：

[IMG_311](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

IEnumerable<int> ints = Enumerable.Range(1,10);

foreach (int i in ints)

{

Console.WriteLine(i); //输出 1 2 3 4 5 6 7 8 9 10 }

Console.ReadKey();

}

[IMG_312](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

26、Repeat操作符

　　Repeat操作符用于生成一个包含指定数量重复元素的序列。

　　方法原型如下：

public static IEnumerable<TResult> Repeat<TResult>(TResult element, int count);

　　留意到这是一个泛型的静态方法，你可以生成任何类型重复的元素，第二个参数代表个数，第一个表示要重复生成的元素。

　　代码示例：

[IMG_313](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

static void Main(string[] args)

{

IEnumerable<int> ints = Enumerable.Repeat(1,10);

foreach (int i in ints)

{

Console.WriteLine(i); //输出 1 1 1 1 1 1 1 1 1 1 }

Console.ReadKey();

}

[IMG_314](https://www.cnblogs.com/kissdodog/archive/2013/02/02/javascript:void(0);)

27、Empty操作符

　　Empty操作符用于生成一个包含指定类型元素的空序列。

　　方法原型如下：

　　　　public static IEnumerable<TResult> Empty<TResult>();

　　从方法原型可以看出，这只是一个普通的静态方法，但是挺有用，因为IEnumerable<T>是个接口，不能new，但是用这个方法可以生成一个空的IEnumerable<T>对象了。

 　　代码示例：

static void Main(string[] args)

{

IEnumerable<int> ints = Enumerable.Empty<int>();

Console.WriteLine(ints.Count()); //输出0 Console.ReadKey();

}

https://www.cnblogs.com/kissdodog/archive/2013/02/02/2890273.html