**[30分钟LINQ教程](http://www.cnblogs.com/liulun/archive/2013/02/26/2909985.html)**

在说LINQ之前必须先说说几个重要的C#语言特性

**一：与LINQ有关的语言特性**

**1.隐式类型**

　　　　（1）源起

　　　　　　在隐式类型出现之前,我们在声明一个变量的时候,总是要为一个变量指定他的类型,甚至在foreach一个集合的时候，也要为遍历的集合的元素，指定变量的类型,隐式类型的出现，程序员就不用再做这个工作了。

　　　　（2）使用方法

　　　　 　　来看下面的代码：

　　　　　　var a = 1; //int a = 1;

　　　　　　var b = "123";//string b = "123";

　　　　　　var myObj = new MyObj();//MyObj myObj = new MyObj()

　　　　　　上面的每行代码，与每行代码后面的注释，起到的作用是完全一样的

　　　　　　也就是说，在声明一个变量（并且同时给它赋值）的时候，完全不用指定变量的类型，只要一个var就解决问题了

　　　　（3）你担心这样写会降低性能吗？

　　　　　　我可以负责任的告诉你，这样写不会影响性能！

　　　　　　上面的代码和注释里的代码，编译后产生的IL代码（中间语言代码）是完全一样的

　　　　　　（编译器根据变量的值，推导出变量的类型，才产生的IL代码）

　　　　（4）这个关键字的好处：

　　　　　　你不用在声明一个变量并给这个变量赋值的时候，写两次变量类型

　　　　　　（这一点真的为开发者节省了很多时间）

　　　　　　在foreach一个集合的时候，可以使用var关键字来代替书写循环变量的类型

　　　　 （5）注意事项

　　　　　　你不能用var关键字声明一个变量而不给它赋值

　　　　　　因为编译器无法推导出你这个变量是什么类型的。

**2.匿名类型**

　　　　(1)源起

　　　　　　创建一个对象，一定要先定义这个对象的类型吗？

　　　　　　不一定的！

　　　　　　来看看这段代码

　　　　（2）使用

var obj = new {Guid.Empty, myTitle = "匿名类型", myOtherParam = new int[] { 1, 2, 3, 4 } };

Console.WriteLine(obj.Empty);//另一个对象的属性名字，被原封不动的拷贝到匿名对象中来了。

Console.WriteLine(obj.myTitle);

Console.ReadKey();

　　　　　　new关键字之后就直接为对象定义了属性，并且为这些属性赋值

　　　　　　而且，对象创建出来之后，在创建对象的方法中，还可以畅通无阻的访问对象的属性

　　　　　　当把一个对象的属性拷贝到匿名对象中时，可以不用显示的指定属性的名字，这时原始属性的名字会被“拷贝”到匿名对象中

　　　　（3）注意

　　　　　　如果你监视变量obj，你会发现，obj的类型是Anonymous Type类型的

　　　　　　不要试图在创建匿名对象的方法外面去访问对象的属性！

　　　　（4）优点

　　　　　　这个特性在网站开发中，序列化和反序列化JSON对象时很有用

**3.自动属性**

　　　　（1）源起

　　　　　　为一个类型定义属性，我们一般都写如下的代码：

[复制代码](javascript:void(0);)

　　　　public class MyObj2

　　　　{

　　　　private Guid \_id;

　　　　private string \_Title;

　　　　public Guid id

　　　　{

　　　　get { return \_id; }

　　　　set { \_id = value; }

　　　　}

　　　　public string Title

　　　　{

　　　　get { return \_Title; }

　　　　set { \_Title = value; }

　　　　}

　　　　}

[复制代码](javascript:void(0);)

　　　　　　但很多时候，这些私有变量对我们一点用处也没有，比如对象关系映射中的实体类。

　　　　　　自C#3.0引入了自动实现的属性，

　　　　　　以上代码可以写成如下形式：

　　　　（2）使用

　　　　public class MyObj

　　　　{

　　　　public Guid id { get; set; }

　　　　public string Title { get; set; }

　　　　}

　　　　　　这个特性也和var关键字一样，是编译器帮我们做了工作，不会影响性能的

**4.初始化器**

　　　　（1）源起

　　　　　　我们创建一个对象并给对象的属性赋值，代码一般写成下面的样子

var myObj = new MyObj();

myObj.id = Guid.NewGuid();

myObj.Title = "allen";

　　　　　　自C#3.0引入了对象初始化器，

　　　　　　代码可以写成如下的样子

　　　　（2）使用

　　　　　　var myObj1 = new MyObj() { id = Guid.NewGuid(), Title = "allen" };

　　　　　　如果一个对象是有参数的构造函数

　　　　　　那么代码看起来就像这样

　　　　　　var myObj1 = new MyObj ("allen") { id = Guid.NewGuid(), Title = "allen" };

　　　　　　集合初始化器的样例代码如下：

　　　　　　var arr = new List<int>() { 1, 2, 3, 4, 5, 6 };

　　　　（3）优点

　　　　　　我个人认为：这个特性不是那么amazing，

　　　　　　这跟我的编码习惯有关，集合初始化器也就罢了，

　　　　　　真的不习惯用对象初始化器初始化一个对象！

**5.委托**

　　　　（1）使用

　　　　　　我们先来看一个简单的委托代码

[复制代码](javascript:void(0);)

　　　　delegate Boolean moreOrlessDelgate(int item);

　　　　class Program

　　　　{

　　　　static void Main(string[] args)

　　　　{

　　　　var arr = new List<int>() { 1, 2, 3, 4, 5, 6,7,8 };

　　　　var d1 = new moreOrlessDelgate(More);

　　　　Print(arr, d1);

　　　　Console.WriteLine("OK");

　　　　var d2 = new moreOrlessDelgate(Less);

　　　　Print(arr, d2);

　　　　Console.WriteLine("OK");

　　　　Console.ReadKey();

　　　　}

　　　　static void Print(List<int> arr,moreOrlessDelgate dl)

　　　　{

　　　　foreach (var item in arr)

　　　　{

　　　　if (dl(item))

　　　　{

　　　　Console.WriteLine(item);

　　　　}

　　　　}

　　　　}

　　　　static bool More(int item)

　　　　{

　　　　if (item > 3)

　　　　{

　　　　return true;

　　　　}

　　　　return false;

　　　　}

　　　　static bool Less(int item)

　　　　{

　　　　if (item < 3)

　　　　{

　　　　return true;

　　　　}

　　　　return false;

　　　　}

　　　　}

[复制代码](javascript:void(0);)

　　　　　　这段代码中

　　　　　　<1>首先定义了一个委托类型

　　　　　　　　delegate Boolean moreOrlessDelgate(int item);

　　　　　　　　你看到了，委托和类是一个级别的，确实是这样：委托是一种类型

　　　　　　　　和class标志的类型不一样，这种类型代表某一类方法。

　　　　　　　　这一句代码的意思是：moreOrlessDelgate这个类型代表返回值为布尔类型，输入参数为整形的方法

　　　　　　<2>有类型就会有类型的实例

　　　　　　　　var d1 = new moreOrlessDelgate(More);

　　　　　　　　var d2 = new moreOrlessDelgate(Less);

　　　　　　　　这两句就是创建moreOrlessDelgate类型实例的代码，

　　　　　　　　它们的输入参数是两个方法

　　　　　　<3>有了类型的实例，就会有操作实例的代码

　　　　　　　　Print(arr, d1);

　　　　　　　　Print(arr, d2);

　　　　　　　　我们把前面两个实例传递给了Print方法

　　　　　　　　这个方法的第二个参数就是moreOrlessDelgate类型的

　　　　　　　　在Print方法内用如下代码，调用委托类型实例所指向的方法

　　　　　　　　dl(item)

**6.泛型**

　　　　(1)为什么要有泛型

　　　　　　假设你是一个方法的设计者，

　　　　　　这个方法有一个传入参数，有一个返回值。

　　　　　　但你并不知道这个参数和返回值是什么类型的，

　　　　　　如果没有泛型，你可能把参数和返回值的类型都设定为Object了

　　　　　　那时，你心里肯定在想：反正一切都是对象，一切的基类都是Object

　　　　　　没错！你是对的！

　　　　　　这个方法的消费者，会把他的对象传进来（有可能会做一次装箱操作）

　　　　　　并且得到一个Object的返回值，他再把这个返回值强制类型转化为他需要的类型

　　　　　　除了装箱和类型转化时的性能损耗外，代码工作的很好！

　　　　　　那么这些新能损耗能避免掉吗？

　　　　　　有泛型之后就可以了！

　　　　（2）使用

　　　　　　<1>使用简单的泛型

　　　　　　　　先来看下面的代码：

[复制代码](javascript:void(0);)

　　var intList = new List<int>() { 1,2,3};

　　intList.Add(4);

　　intList.Insert(0, 5);

　　foreach (var item in intList)

　　{

　　Console.WriteLine(item);

　　}

　　Console.ReadKey();

[复制代码](javascript:void(0);)

　　　　　　　　在上面这段代码中我们声明了一个存储int类型的List容器

　　　　　　　　并循环打印出了容器里的值

　　　　　　　　注意：如果这里使用Hashtable、Queue或者Stack等非泛型的容器

　　　　　　　　就会导致装箱操作，损耗性能。因为这些容器只能存储Object类型的数据

　　　　　　<2>泛型类型

　　　　　　　　List<T>、Dictionary<TKey, TValue>等泛型类型都是.net类库定义好并提供给我们使用的

　　　　　　　　但在实际开发中，我们也经常需要定义自己的泛型类型

　　　　　　　　来看下面的代码：

[复制代码](javascript:void(0);)

　　　　　　public static class SomethingFactory<T>

　　　　　　{

　　　　　　public static T InitInstance(T inObj)

　　　　　　{

　　　　　　if (false)//你的判断条件

　　　　　　{

　　　　　　//do what you want...

　　　　　　return inObj;

　　　　　　}

　　　　　　return default(T);

　　　　　　}

　　　　　　}

[复制代码](javascript:void(0);)

　　　　　　　　这段代码的消费者如下：

　　var a1 = SomethingFactory<int>.InitInstance(12);

　　Console.WriteLine(a1);

　　Console.ReadKey();

　　　　　　　　输出的结果为0

　　　　　　　　这就是一个自定义的静态泛型类型，

　　　　　　　　此类型中的静态方法InitInstance对传入的参数做了一个判断

　　　　　　　　如果条件成立，则对传入参数进行操作之后并把它返回

　　　　　　　　如果条件不成立，则返回一个空值

　　　　　　　　注意：

　　　　　　　　　　[1]

　　　　　　　　　　　　传入参数必须为指定的类型，

　　　　　　　　　　　　因为我们在使用这个泛型类型的时候，已经规定好它能接收什么类型的参数

　　　　　　　　　　　　但在设计这个泛型的时候，我们并不知道使用者将传递什么类型的参数进来

　　　　　　　　　　[2]

　　　　　　　　　　　　如果你想返回T类型的空值，那么请用default(T)这种形式

　　　　　　　　　　　　因为你不知道T是值类型还是引用类型，所以别擅自用null

　　　　　　<3>泛型约束

　　　　　　　　很多时候我们不希望使用者太过自由

　　　　　　　　我们希望他们在使用我们设计的泛型类型时

　　　　　　　　不要很随意的传入任何类型

　　　　　　　　对于泛型类型的设计者来说，要求使用者传入指定的类型是很有必要的

　　　　　　　　因为我们只有知道他传入了什么东西，才方便对这个东西做操作

　　　　　　　　让我们来给上面设计的泛型类型加一个泛型约束

　　　　　　　　代码如下：

　　　　　　　　　　public static class SomethingFactory<T> where T:MyObj

　　　　　　　　这样在使用SomethingFactory的时候就只能传入MyObj类型或MyObj的派生类型啦

　　　　　　　　注意：

　　　　　　　　　　还可以写成这样

　　　　　　　　　　where T:MyObj,new()

　　　　　　　　　　来约束传入的类型必须有一个构造函数。

　　　　（3）泛型的好处

　　　　　　<1>算法的重用

　　　　　　　　想想看：list类型的排序算法，对所有类型的list集合都是有用的

　　　　　　<2>类型安全

　　　　　　<3>提升性能

　　　　　　　　没有类型转化了，一方面保证类型安全，另一方面保证性能提升

　　　　　　<4>可读性更好

　　　　　　　　这一点就不解释了

**7.泛型委托**

　　　　（1）源起

　　　　　　委托需要定义delgate类型

　　　　　　使用起来颇多不便

　　　　　　而且委托本就代表某一类方法

　　　　　　开发人员经常使用的委托基本可以归为三类，

　　　　　　哪三类呢？

　　　　　　请看下面：

　　　　（2）使用

　　　　　　<1>Predicate泛型委托

　　　　　　　　把上面例子中d1和d2赋值的两行代码改为如下：

　　//var d1 = new moreOrlessDelgate(More);

　　var d1 = new Predicate<int>(More);

　　//var d2 = new moreOrlessDelgate(Less);

　　var d2 = new Predicate<int>(Less);

　　　　　　　　把Print方法的方法签名改为如下：

　　　　//static void Print(List<int> arr, moreOrlessDelgate<int> dl)

　　　　static void Print(List<int> arr, Predicate<int> dl)

　　　　　　　　然后再运行方法，控制台输出的结果和原来的结果是一模一样的。

　　　　　　　　那么Predicate到底是什么呢？

　　　　　　　　来看看他的定义：

[复制代码](javascript:void(0);)

　　　　　　// 摘要:

　　　　　　// 表示定义一组条件并确定指定对象是否符合这些条件的方法。

　　　　　　//

　　　　　　// 参数:

　　　　　　// obj:

　　　　　　// 要按照由此委托表示的方法中定义的条件进行比较的对象。

　　　　　　//

　　　　　　// 类型参数:

　　　　　　// T:

　　　　　　// 要比较的对象的类型。

　　　　　　//

　　　　　　// 返回结果:

　　　　　　// 如果 obj 符合由此委托表示的方法中定义的条件，则为 true；否则为 false。

　　　　　　public delegate bool Predicate<in T>(T obj);

[复制代码](javascript:void(0);)

　　　　　　　　看到这个定义，我们大致明白了。

　　　　　　　　.net为我们定义了一个委托，

　　　　　　　　这个委托表示的方法需要传入一个T类型的参数，并且需要返回一个bool类型的返回值

　　　　　　　　有了它，我们就不用再定义moreOrlessDelgate委托了，

　　　　　　　　而且，我们定义的moreOrlessDelgate只能搞int类型的参数，

　　　　　　　　Predicate却不一样，它可以搞任意类型的参数

　　　　　　　　但它规定的还是太死了，它必须有一个返回值，而且必须是布尔类型的，同时，它必须有一个输入参数

　　　　　　　　除了Predicate泛型委托，.net还为我们定义了Action和Func两个泛型委托

　　　　　　<2>Action泛型委托

　　　　　　　　Action泛型委托限制的就不那么死了，

　　　　　　　　他代表了一类方法：

　　　　　　　　可以有0个到16个输入参数，

　　　　　　　　输入参数的类型是不确定的，

　　　　　　　　但不能有返回值，

　　　　　　　　来看个例子：

　　var d3 = new Action(noParamNoReturnAction);

　　var d4 = new Action<int, string>(twoParamNoReturnAction);

 　　　　　　　　注意：尖括号中int和string为方法的输入参数

[复制代码](javascript:void(0);)

　　　　static void noParamNoReturnAction()

　　　　{

　　　　//do what you want

　　　　}

　　　　static void twoParamNoReturnAction(int a, string b)

　　　　{

　　　　//do what you want

　　　　}

[复制代码](javascript:void(0);)

 　　　　　　<3>Func泛型委托

　　　　　　　　为了弥补Action泛型委托，不能返回值的不足

　　　　　　　　.net提供了Func泛型委托，

　　　　　　　　相同的是它也是最多0到16个输入参数，参数类型由使用者确定

　　　　　　　　不同的是它规定要有一个返回值，返回值的类型也由使用者确定

　　　　　　　　如下示例：

　　　　　　　　　　var d5 = new Func<int, string>(oneParamOneReturnFunc);

　　　　　　　　注意：string类型（最后一个泛型类型）是方法的返回值类型

　　　　static string oneParamOneReturnFunc(int a)

　　　　{

　　　　//do what you want

　　　　return string.Empty;

　　　　}

**8.匿名方法**

　　　　（1）源起

　　　　　　在上面的例子中

　　　　　　为了得到序列中较大的值

　　　　　　我们定义了一个More方法

　　　　　　var d1 = new Predicate<int>(More);

　　　　　　然而这个方法，没有太多逻辑（实际编程过程中，如果逻辑较多，确实应该独立一个方法出来）

　　　　　　那么能不能把More方法中的逻辑，直接写出来呢？

　　　　　　C#2.0之后就可以了，

　　　　　　请看下面的代码：

　　　　（2）使用

[复制代码](javascript:void(0);)

var arr = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8 };

//var d1 = new moreOrlessDelgate(More);

//var d1 = new Predicate<int>(More);

var d1 = new Predicate<int>(delegate(int item)

{

　　　　　　　　　　//可以访问当前上下文中的变量  
　　　　　　　　　　Console.WriteLine(arr.Count);

                if (item > 3)

{

return true;

}

return false;

});

Print(arr, d1);

Console.WriteLine("OK");

[复制代码](javascript:void(0);)

　　　　　　我们传递了一个代码块给Predicate的构造函数

　　　　　　其实这个代码块就是More函数的逻辑

　　　　（3）好处

　　　　　　<1>代码可读性更好

　　　　　　<2>可以访问当前上下文中的变量

　　　　　　　　这个用处非常大，

　　　　　　　　如果我们仍旧用原来的More函数

　　　　　　　　想要访问arr变量，势必要把arr写成类级别的私有变量了

　　　　　　　　用匿名函数的话，就不用这么做了。

**9.Lambda表达式**

　　　　（1）源起

　　　　　　.net的设计者发现在使用匿名方法时，

　　　　　　仍旧有一些多余的字母或单词的编码工作

　　　　　　比如delegate关键字

　　　　　　于是进一步简化了匿名方法的写法

　　　　（2）使用

List<int> arr = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };

arr.ForEach(new Action<int>(delegate(int a) { Console.WriteLine(a); }));

arr.ForEach(new Action<int>(a => Console.WriteLine(a)));

 　　　　　　匿名方法的代码如下：

　　　　　　delegate(int a) { Console.WriteLine(a); }

　　　　　　使用lambda表达式的代码如下：

　　　　　　a => Console.WriteLine(a)

　　　　　　这里解释一下这个lambda表达式

　　　　　　<1>

　　　　　　　　a是输入参数，编译器可以自动推断出它是什么类型的，

　　　　　　　　如果没有输入参数，可以写成这样：

　　　　　　　　() => Console.WriteLine("ddd")

　　　　　　<2>

　　　　　　　　=>是lambda操作符

　　　　　　<3>

　　　　　　　　Console.WriteLine(a)是要执行的语句。

　　　　　　　　如果是多条语句的话，可以用{}包起来。

　　　　　　　　如果需要返回值的话，可以直接写return语句

**10.扩展方法**

　　　　（1）源起

　　　　　　如果想给一个类型增加行为，一定要通过继承的方式实现吗？

　　　　　　不一定的！

　　　　（2）使用

　　　　　　来看看这段代码：

　　public static void PrintString(this String val)

　　{

　　Console.WriteLine(val);

　　}

　　　　　　消费这段代码的代码如下：

var a = "aaa";

a.PrintString();

Console.ReadKey();

　　　　　　我想你看到扩展方法的威力了。

　　　　　　本来string类型没有PrintString方法

　　　　　　但通过我们上面的代码，就给string类型"扩展"了一个PrintString方法

　　　　　　（1）先决条件

　　　　　　　　<1>扩展方法必须在一个非嵌套、非泛型的静态类中定义

　　　　　　　　<2>扩展方法必须是一个静态方法

　　　　　　　　<3>扩展方法至少要有一个参数

　　　　　　　　<4>第一个参数必须附加this关键字作为前缀

　　　　　　　　<5>第一个参数不能有其他修饰符（比如ref或者out）

　　　　　　　　<6>第一个参数不能是指针类型

　　　　　　（2）注意事项

　　　　　　　　<1>跟前面提到的几个特性一样，扩展方法只会增加编译器的工作，不会影响性能（用继承的方式为一个类型增加特性反而会影响性能）

　　　　　　　　<2>如果原来的类中有一个方法，跟你的扩展方法一样（至少用起来是一样），那么你的扩展方法奖不会被调用，编译器也不会提示你

　　　　　　　　<3>扩展方法太强大了，会影响架构、模式、可读性等等等等....

**11.迭代器**

　　·　　（1）使用

　　　　　　我们每次针对集合类型编写foreach代码块，都是在使用迭代器

　　　　　　这些集合类型都实现了IEnumerable接口

　　　　　　都有一个GetEnumerator方法

　　　　　　但对于数组类型就不是这样

　　　　　　编译器把针对数组类型的foreach代码块

　　　　　　替换成了for代码块。

　　　　　　来看看List的类型签名：

　　　　　　public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable

　　　　　　IEnumerable接口，只定义了一个方法就是：

　　　　　　IEnumerator<T> GetEnumerator();

　　　　（2）迭代器的优点：

　　　　　　假设我们需要遍历一个庞大的集合

　　　　　　只要集合中的某一个元素满足条件

　　　　　　就完成了任务

　　　　　　你认为需要把这个庞大的集合全部加载到内存中来吗？

　　　　　　当然不用（C#3.0之后就不用了）！

　　　　　　来看看这段代码：

[复制代码](javascript:void(0);)

static IEnumerable<int> GetIterator()

{

Console.WriteLine("迭代器返回了1");

yield return 1;

Console.WriteLine("迭代器返回了2");

yield return 2;

Console.WriteLine("迭代器返回了3");

yield return 3;

}

[复制代码](javascript:void(0);)

　　　　　　消费这个函数的代码如下：

[复制代码](javascript:void(0);)

foreach (var i in GetIterator())

{

if (i == 2)

{

break;

}

Console.WriteLine(i);

}

Console.ReadKey();

[复制代码](javascript:void(0);)

　　　　　　输出结果为：

　　　　　　迭代器返回了1

　　　　　　1

　　　　　　迭代器返回了2

　　　　　　大家可以看到：

　　　　　　当迭代器返回2之后，foreach就退出了

　　　　　　并没有输出“迭代器返回了3”

　　　　　　也就是说下面的工作没有做。

　　　　（3）yield 关键字

　　　　　　MSDN中的解释如下：

　　　　　　在迭代器块中用于向枚举数对象提供值或发出迭代结束信号。

　　　　　　也就是说，我们可以在生成迭代器的时候，来确定什么时候终结迭代逻辑

　　　　　　上面的代码可以改成如下形式：

[复制代码](javascript:void(0);)

　　static IEnumerable<int> GetIterator()

　　{

　　Console.WriteLine("迭代器返回了1");

　　yield return 1;

　　Console.WriteLine("迭代器返回了2");

　　yield break;

　　Console.WriteLine("迭代器返回了3");

　　yield return 3;

　　}

[复制代码](javascript:void(0);)

 　　　　(4)注意事项

　　　　　　<1>做foreach循环时多考虑线程安全性

　　　　　　　　在foreach时不要试图对被遍历的集合进行remove和add等操作

　　　　　　　　任何集合，即使被标记为线程安全的，在foreach的时候，增加项和移除项的操作都会导致异常

　　　　　　　　（我在这里犯过错）

　　　　　　<2>IEnumerable接口是LINQ特性的核心接口

　　　　　　　　只有实现了IEnumerable接口的集合

　　　　　　　　才能执行相关的LINQ操作，比如select,where等

　　　　　　　　这些操作，我们接下来会讲到。

**二：LINQ**

**1.查询操作符**

　　　　（1）源起

　　　　　　.net的设计者在类库中定义了一系列的扩展方法

　　　　　　来方便用户操作集合对象

　　　　　　这些扩展方法构成了LINQ的查询操作符

　　　　（2）使用

　　　　　　这一系列的扩展方法，比如：

　　　　　　Where，Max，Select，Sum，Any，Average，All，Concat等

　　　　　　都是针对IEnumerable的对象进行扩展的

　　　　　　也就是说，只要实现了IEnumerable接口，就可以使用这些扩展方法

　　　　　　来看看这段代码：

List<int> arr = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };

var result = arr.Where(a => { return a > 3; }).Sum();

Console.WriteLine(result);

Console.ReadKey();

　　　　　　这段代码中，用到了两个扩展方法。

　　　　　　<1>

　　　　　　　　Where扩展方法，需要传入一个Func<int,bool>类型的泛型委托

　　　　　　　　这个泛型委托，需要一个int类型的输入参数和一个布尔类型的返回值

　　　　　　　　我们直接把a => { return a > 3; }这个lambda表达式传递给了Where方法

　　　　　　　　a就是int类型的输入参数，返回a是否大于3的结果。

　　　　　　<2>

　　　　　　　　Sum扩展方法计算了Where扩展方法返回的集合的和。

　　　　（3）好处

　　　　　　上面的代码中

　　　　　　arr.Where(a => { return a > 3; }).Sum();

　　　　　　这一句完全可以写成如下代码：

　　　　　　(from v in arr where v > 3 select v).Sum();

　　　　　　而且两句代码的执行细节是完全一样的

　　　　　　大家可以看到,第二句代码更符合语义，更容易读懂

　　　　　　第二句代码中的where，就是我们要说的查询操作符。

　　　　（4）标准查询操作符说明

　　　　　　<1>过滤

　　　　　　　　Where

　　　　　　　　用法：arr.Where(a => { return a > 3; })

　　　　　　　　说明：找到集合中满足指定条件的元素

　　　　　　　　OfType

　　　　　　　　用法：arr.OfType<int>()

　　　　　　　　说明：根据指定类型，筛选集合中的元素

　　　　　　<2>投影

　　　　　　　　Select

　　　　　　　　用法：arr.Select<int, string>(a => a.ToString());

　　　　　　　　说明：将集合中的每个元素投影的新集合中。上例中：新集合是一个IEnumerable<String>的集合

　　　　　　　　SelectMany

　　　　　　　　用法：arr.SelectMany<int, string>(a => { return new List<string>() { "a", a.ToString() }; });

　　　　　　　　说明：将序列的每个元素投影到一个序列中，最终把所有的序列合并

　　　　　　<3>还有很多查询操作符，请翻MSDN，以后有时间我将另起一篇文章把这些操作符写全。

**2.查询表达式**

　　　　（1）源起

　　　　　　上面我们已经提到，使用查询操作符表示的扩张方法来操作集合

　　　　　　虽然已经很方便了，但在可读性和代码的语义来考虑，仍有不足

　　　　　　于是就产生了查询表达式的写法。

　　　　　　虽然这很像SQL语句，但他们却有着本质的不同。

　　　　（2）用法

　　　　　　from v in arr where v > 3 select v

　　　　　　这就是一个非常简单的查询表达式

　　　　（3）说明：

　　　　　　先看一段伪代码：

　　　　　　from [type] id in source

　　　　　　[join [type] id in source on expr equals expr [into subGroup]]

　　　　　　[from [type] id in source | let id = expr | where condition]

　　　　　　[orderby ordering,ordering,ordering...]

　　　　　　select expr | group expr by key

　　　　　　[into id query]

　　　　　　<1>第一行的解释：

　　　　　　　　type是可选的，

　　　　　　　　id是集合中的一项，

　　　　　　　　source是一个集合，

　　　　　　　　如果集合中的类型与type指定的类型不同则导致强制转化

　　　　　　<2>第二行的解释：

　　　　　　　　一个查询表达式中可以有0个或多个join子句，

　　　　　　　　这里的source可以不等于第一句中的source

　　　　　　　　expr可以是一个表达式

　　　　　　　　[into subGroup] subGroup是一个中间变量，

　　　　　　　　它继承自IGrouping，代表一个分组，也就是说“一对多”里的“多”

　　　　　　　　可以通过这个变量得到这一组包含的对象个数，以及这一组对象的键

　　　　　　　　比如：

[复制代码](javascript:void(0);)

　　　　　　　　from c in db.Customers

　　　　join o in db.Orders on c.CustomerID

　　　　equals o.CustomerID into orders

　　　　select new

　　　　{

　　　　c.ContactName,

　　　　OrderCount = orders.Count()

　　　　};

[复制代码](javascript:void(0);)

　　　　　　<3>第三行的解释：

　　　　　　　　一个查询表达式中可以有1个或多个from子句

　　　　　　　　一个查询表达式中可以有0个或多个let子句，let子句可以创建一个临时变量

　　　　　　　　比如：

　　　　from u in users

　　　　let number = Int32.Parse(u.Username.Substring(u.Username.Length - 1))

　　　　where u.ID < 9 && number % 2 == 0

　　　　select u

　　　　　　　　一个查询表达式中可以有0个或多个where子句，where子句可以指定查询条件

　　　　　　<4>第四行的解释：

　　　　　　　　一个查询表达式可以有0个或多个排序方式

　　　　　　　　每个排序方式以逗号分割

　　　　　　<5>第五行的解释：

　　　　　　　　一个查询表达式必须以select或者group by结束

　　　　　　　　select后跟要检索的内容

　　　　　　　　group by 是对检索的内容进行分组

　　　　　　　　比如：

　　　　from p in db.Products

　　　　group p by p.CategoryID into g

　　　　select new { g.Key, NumProducts = g.Count()};

　　　　　　<6>第六行的解释：

　　　　　　　　最后一个into子句起到的作用是

　　　　　　　　将前面语句的结果作为后面语句操作的数据源

　　　　　　　　比如：

[复制代码](javascript:void(0);)

　　　　from p in db.Employees

　　　　select new

　　　　{

　　　　LastName = p.LastName,

　　　　TitleOfCourtesy = p.TitleOfCourtesy

　　　　} into EmployeesList

　　　　orderby EmployeesList.TitleOfCourtesy ascending

　　　　select EmployeesList;