第 20 章　ASP.NET 3.5 与 LINQ

对于长期发展的面向对象编程模型而言，其发展基本处于一个比较稳定的阶段，可是面向对象的编程模型并没有解决数据的访问和整合的复杂问题。对于数据库的访问和 XML 的访问，面向对象方法论无法从根本意义上解决其复杂度和难度，而 LINQ 提供了一种更好的解决方案。

20.1　什么是 LINQ

任何技术都不可能凭空搭建起来，为了解决工业生产生中某个实际问题，当现有的技术已经无法很好的完成工业的要求，就会促发新技术的诞生。LINQ 就是为了解决复杂的数据访问和整合而出现的一种新技术。

20.1.1　LINQ 起源

从传统的意义上来说，面向过程的编程模型在数据访问和整合的能力上有一定的限度。因为面向过程的编程方法不能很好的描述一个事务，必须通过不同函数之间的调用来描述一个现有的对象，而且面向过程的编程方法在代码复用性上比较低，所以当面向过程的编程语言需要对数据库进行访问时，就需要编写大量的额外代码。虽然面向过程的编程模型可以通过良好的函数引用和编码提高复用性，但是并没有解决面向过程编程模型中对数据的访问和整合的复杂度。

随着计算机和编程模型的发展，人们发现了另一个更好的编程模型，这就是现在最常用的面向对象编程模型。相比面向过程的编程模型而言，面向对象的编程模型能够更好的描述一个事务，事务能够通过面向对象中的属性、字段和方法很好的模拟实际的事务，而面向对象编程模型中的派生、继承等特性同样能够极大的提高代码的复用性，提升开发效率。

但是面向对象的编程模型同样没有解决复杂的数据库访问和数据整合，开发人员还是需要通过繁琐的手段进行数据库的访问和数据整合。在.NET 3.0 框架或更早，LINQ 就已经被提及，LINQ 是一种能够快速对大部分数据源进行访问和数据整合的一种技术，LINQ 解决了复杂的数据应用中开发人员需要面对和解决的问题。

虽然面向对象的数据库已经在几年前就被提及并且各大IT 公司投入了对面向对象的数据库的研发，但是传统的关系型数据库在当今还是应用最为广泛的。关系型数据库中将数据整合和呈现成为一张张的表的形式，开发人员和数据库管理人员能够通过 SQL 管理工具提供的 SQL 语句进行数据的查询和整理。但是在开发过程中，开发人员不能够像使用 SQL 语句一样对数据集进行查询和处理。任何数据库中的数据都会以一种数据集的形式反馈给用户，这种数据集的形式可以反映成为数学中的集合的概念，其实在数据库早期的发展中，数据是以集合的概念呈现的，而随着数据库的发展，集合的概念依旧是数据库最基本的概念。

正式因为如此，开发人员不能够方便的是从一个集合中查询数据，这里不仅仅是一个数据库，还包括其他能够以数据库形式存在的文件，例如 ACCESS、TXT 等，当在开发中需要使用到多个数据库或者数据描述形式的文件时，更多的情况是将这些数据填充到数据集中并通过遍历来访问数据，这样却造成

了更多的数据访问问题和麻烦。

LINQ 能够很方便的进行数据的查询，使用 LINQ 对数据集进行查询的形式很像使用 SQL 语句对数据库中的表进行查询，而与之不同的是，LINQ 能够面向更多的对象，这些对象包括数组、集合以及数据库，LINQ 对数组的查询示例代码如下所示。

```
static void Main(string[] args)
{
    string[] str = { "你好","今天的","天气真不错","生活很阳光"};          //创建数组
    var s = from n in str select n;                                    //编写查询字串
    foreach (var n in s)                                               //遍历查询对象
    {
        Console.WriteLine(n.ToString());                               //输出对象值
    }
    Console.ReadKey();                                                 //等待用户按键
}
```

上述代码对数组 str 进行了查询，这种方式很像 SQL 语句。的确 LINQ 的查询方式和 SQL 语句很像，其语法和基本内容都没有什么太大的差别，但是 LINQ 提供了更好的查询的解决方案。LINQ 能够查询更多对象（例如上述代码中的数组）而无法使用 SQL 语句进行查询。另外，LINQ 查询语句还能够使用 WHERE 等关键字进行查询，示例代码如下所示。

```
static void Main(string[] args)
{
    string[] str = { "你好","今天的","天气真不错","生活很阳光"};          //创建数组
    var s = from n in str where n.Length > 3 select n;                 //使用条件查询
    foreach (var n in s)                                               //遍历查询对象
    {
        Console.WriteLine(n.ToString());                               //输出对象值
    }
    Console.ReadKey();                                                 //等待用户按键
}
```

上述代码修改了 LINQ 查询语句，为 LINQ 查询语句增加了条件查询，该条件的意义为查询字符串长度大于 3 的字符串，运行后如图 20-1 所示。

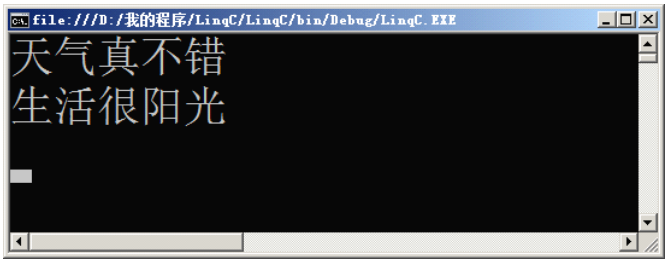

图 20-1　LINQ 查询语句

从上图可以看出，能够使用类似于 SQL 语句的形式进行数据集的查询，很大程度上方便了开发人员对于数据库中数据的访问和整理。LINQ 可以使用条件语句进行筛选，并且能够使用.NET 提供的语法进行判断，这样就简化了开发人员对于数据集中的数据的筛选。有关 LINQ 的语句，会在后面的章节中详细的讲解。

20.1.2 LINQ 构架

在.NET 3.5 中，LINQ（Language Integrated Query）已经成为了编程语言的一部分，开发人员已经能够使用 Visual Studio 2008 创建使用 LINQ 的应用程序。LINQ 对基于.NET 平台的编程语言提供了标准的查询操作。在.NET 3.5 中，LINQ 的基本构架如图 20-2 所示。

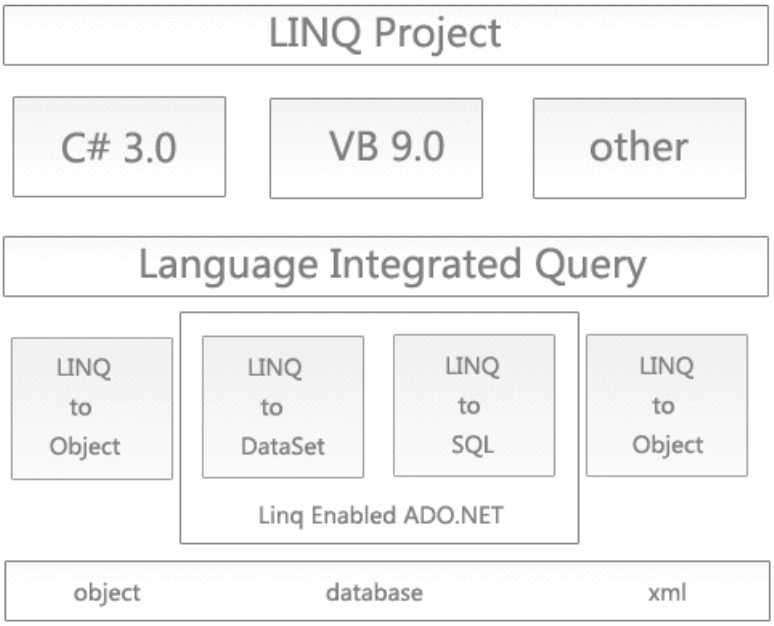

图 20-2　LINQ 基本构架

如图 20-2 所示，LINQ 能够对不同的对象进行查询。在.NET 3.5 中，微软提供了不同的命名空间以支持不同的数据库配合 LINQ 进行数据查询。在 LINQ 框架中，处于最上方的就是 LINQ 应用程序，LINQ 应用程序基于.NET 框架而存在的，LINQ 能够支持 C#、VB 等.NET 平台下的宿主语言进行 LINQ 查询。在 LINQ 框架中，还包括 Linq Enabled ADO.NET 层，该层提供了 LINQ 查询操作并能够提供数据访问和整合功能。

LINQ 包括五个部分，这五个部分分别是 LINQ to Objects、LINQ to DataSet、LINQ to SQL、LINQ to Entities、LINQ to XML，在 .NET 开发中最常用的是 LINQ to SQL 和 LINQ to XML，本书也详细介绍 LINQ 的这两个部分。

LINQ to SQL 提供了对 SQL Server 中数据库的访问和整合功能，同时能够以对象的形式进行数据库管理，前面已经提到，现在的数据库依旧以关系型数据库为主，在面向对象开发过程中，很难通过对象的方法描述数据库，而 LINQ 提供了通过对象的形式对数据库进行描述。LINQ to XML 提供了对 XML 中数据集的访问和整合功能，LINQ to XML 使用 System.Xml.Linq 命名控件，为 XML 操作提供了高效易用的方法。

20.1.3 LINQ 与 Visual Studio 2008 新特性

讲到 LINQ 就不得不讲解 Visual Studio 2008 的新特性，LINQ 作为 Visual Studio 2008 中的一部分，Visual Studio 2008 为 LINQ 提供很好的编程环境，LINQ 也使用到了 C#编程语言中的很多特性，以提高

开发人员的开发效率。

- Visual Studio 2008 重定向：使用 Visual Studio 2008 与 Visual Studio 2005 不同的是，Visual Studio 2008 支持多个版本.NET 框架的共存，在 Visual Studio 2008 中可以选择基于.NET 2.0 或.NET 3.X 版本的框架来开发不同的应用程序，当选择不同的应用程序基础框架时，Visual Studio 2008 能够智能的提供不同的命名空间。
- Visual Studio 2008 AJAX：在 ASP.NET 2.0 开发中，需要使用 ASP.NET AJAX 1.0 作为 AJAX 开发必备的工具，在 Visual Studio 2008 中已经集成了对 AJAX 的支持，创建 ASP.NET 3.5 应用程序已经能够非常方便的使用 AJAX 功能。
- Visual Studio 2008 可视化操作：在 Visual Studio 2008 中，微软提供了可视化操作，开发人员能够选择不同的视图进行页面分离形式的开发，在 Visual Studio 2008 中开发人员可以选择视图，拆分，代码三种视图进行不同的开发体验。
- Visual Studio 2008 集成 LINQ：这是 Visual Studio 2008 中比较值得期待的功能，Visual Studio 2008 将 LINQ 作为编程语言中的一部分，为开发人员提供了 LINQ 开发的原生环境。

在 LINQ 与 Visual Studio 2008 中，开发人员最为期待的新特性还是 Visual Studio 2008 对 LINQ 的原生支持，使用 LINQ 能够快速的进行数据库的访问和整合，这样在一定的意义上降低了开发难度，LINQ 在.NET Framework 3.5 中的位置如图 20-3 所示。

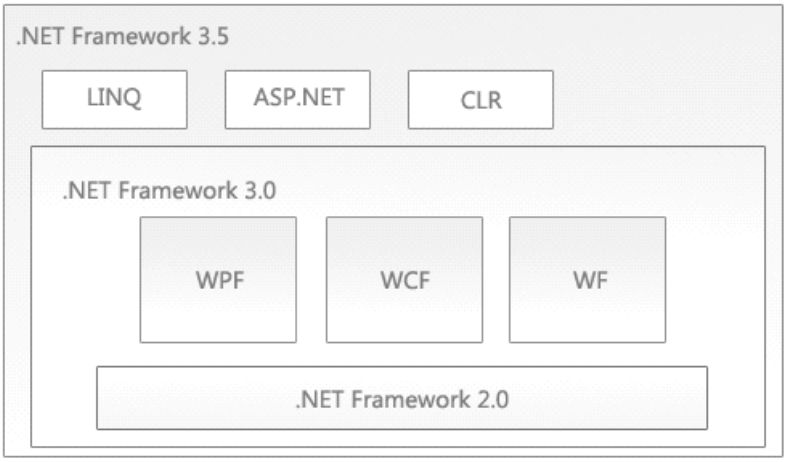

图 20-3　.NET 框架中的 LINQ

正如图 20-3 所示，.NET 2.0 后面几个版本的框架都是基于.NET Framework 2.0 而存在的，在.NET Framework 3.0 中，微软已经增加了 WPF，WCF，WF 等新特性以提供快速的面向服务的开发和完善的用户体验解决方案。而 LINQ 是作为.NET Framework 3.5 存在于.NET Framework 中的，这也就是说只有在.NET Framework 3.5 框架中才能够使用 LINQ 技术。由于.NET Framework 3.5 版本的框架基于.NET Framework 3.0 版本，开发人员可以使用 LINQ 特性进行分布式开发和面向服务的开发，这样就能够更近一步的提高代码的复用性和安全性。

20.2　LINQ 与 Web 应用程序

在 ASP.NET 应用程序开发中，常常需要涉及到数据的显式和整合，使用 ASP.NET 2.0 中提供的控件能够编写用户控件，开发人员还能够选择开发自定义控件进行数据显示和整合，但是在数据显示和整

合过程中，开发人员往往需要大量的连接、关闭连接等操作，而且传统的方法也破坏了面向对象的特性，使用 LINQ 能够方便的使用面向对象的方法进行数据库操作。

20.2.1 创建使用 LINQ 的 Web 应用程序

创建 LINQ 的 Web 应用程序非常的容易，只要创建 Web 应用程序时选择的平台是基于.NET Framework 3.5 的就能够创建使用 LINQ 的 Web 应用程序，如图 20-4 所示。

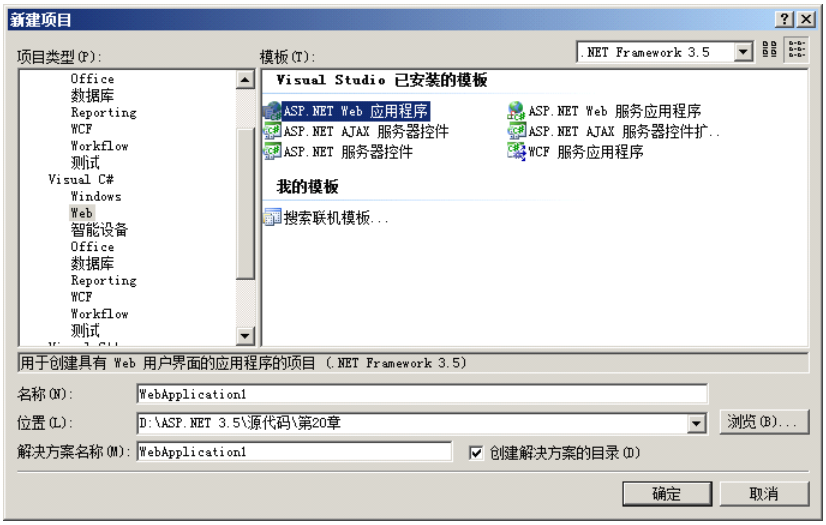

图 20-4　选择.NET Framework 3.5

当创建一个基于系统.NET Framework 3.5 的应用程序，系统就能够自动为应用程序创建 LINQ 所需要的命名空间，示例代码如下所示。

```
using System.Xml.Linq;                                          //使用 LINQ 命名空间
using System.Linq;                                              //使用 LINQ 命名空间
```

上述命名空间提供了应用程序中使用 LINQ 所需要的基础类和枚举，在 ASP.NET 应用程序中就能够使用 LINQ 查询语句进行查询，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    string[] str = { "我爱 C#", "我喜欢 C#", "我做 C#开发", "基于.NET 平台", "LINQ 应用" };  //数据集
    var s = from n in str where n.Contains("C#") select n;                  //执行 LINQ 查询
    foreach (var t in s)                                                    //遍历对象
    {
        Response.Write(t.ToString() + "<br/>");                             //输出查询结果
    }
}
```

上述代码在 ASP.NET 页面中执行了一段 LINQ 查询，查询字符串中包含“C#”的字符串，运行后如图 20-5 所示。

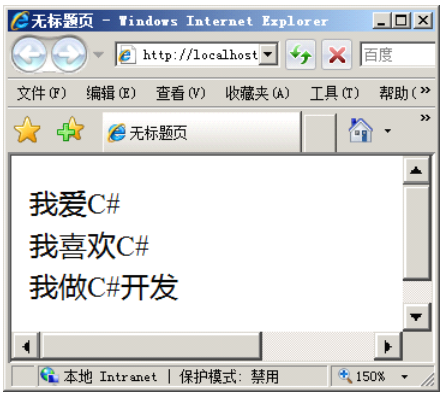

图 20-5　ASP.NET 执行 LINQ 查询

在 ASP.NET 中能够使用 LINQ 进行数据集的查询，Visual Studio 2008 已经将 LINQ 整合成为编程语言中的一部分，基于.NET Framework 3.5 的应用程序都可以使用 LINQ 特性进行数据访问和整合。

20.2.2　基本的 LINQ 数据查询

使用 LINQ 能够对数据集进行查询，在 ASP.NET 中，可以创建一个新的 LINQ 数据库进行数据集查询，右击现有项目，单击【添加新项】选项，选择【LINQ to SQL 类】选项，如图 20-6 所示。

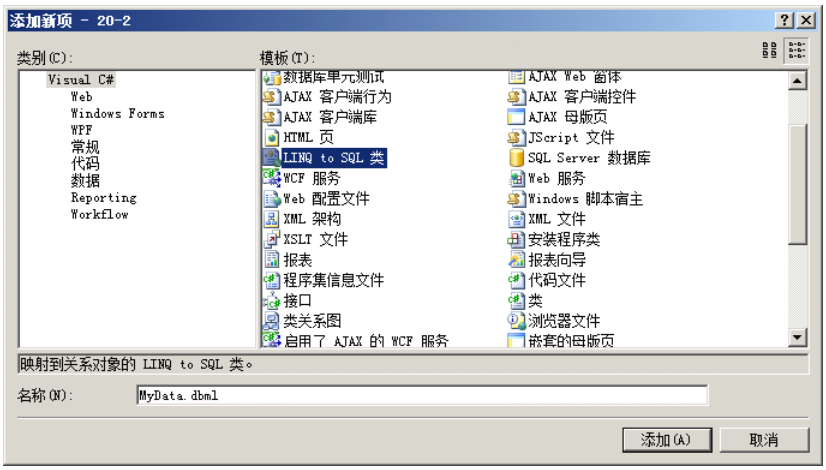

图 20-6　创建 LINQ to SQL 类

创建一个 LINQ to SQL 类，能够映射一个数据库，实现数据对象的创建，如图 20-7 所示。创建一个 LINQ to SQL 类后，可以直接在服务资源管理器中拖动相应的表到 LINQ to SQL 类文件中，如图 20-8 所示。

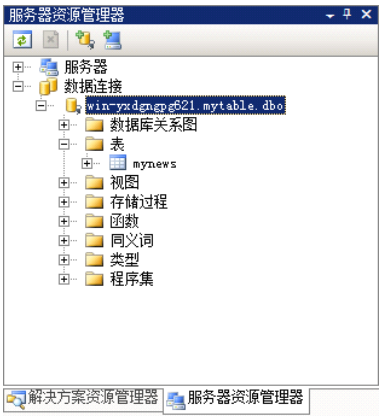

图 20-7　服务资源管理器

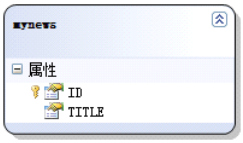

图 20-8　拖动一个表

开发人员能够直接将服务资源管理器中的表拖动到 LINQ to SQL 类中，在 LINQ to SQL 类文件中就会呈现一个表的视图。在视图中，开发人员能够在视图中添加属性和关联，并且能够在 LINQ to SQL 类文件中可以设置多个表，进行可视化关联操作。

创建一个 LINQ to SQL 类文件后，LINQ to SQL 类就将数据进行对象化，这里的对象化就是以面向对象的思想针对一个数据集建立一个相应的类，开发人员能够使用 LINQ to SQL 创建的类进行数据库查询和整合操作，示例代码如下所示。

```
protected void Page_Load(object sender, EventArgs e)
{
    MyDataDataContext data = new MyDataDataContext();                    //使用 LINQ 类
    var s = from n in data.mynews where n.ID==1 select n;                //执行查询
    foreach (var t in s)                                                 //遍历对象
    {
        Response.Write(t.TITLE.ToString() + "<br/>");                    //输出对象
    }
}
```

上述创建了一个 MyData.dbml 的 LINQ to SQL 文件，开发人员能够直接使用该类的对象提供数据操作。上述代码使用了 LINQ to SQL 文件提供的类进行数据查询，LINQ 查询语句示例代码如下所示。

```
var s = from n in data.mynews where n.ID==1 select n;                    //编写查询语句
```

上述代码使用了 LINQ 查询语句查询了一个 mynews 表中 ID 为 1 的行，使用 LINQ to SQL 文件提供的对象能够快速的进行数据集中对象的操作。创建一个 MyData.dbml 的 LINQ to SQL 文件，其中 MyDataDataContext 为类的名称，该类提供 LINQ to SQL 操作方法，示例代码如下所示。

```
MyDataDataContext data = new MyDataDataContext();                        //使用 LINQ 类
```

上述代码使用了 LINQ to SQL 文件提供的类创建了一个对象 data，data 对象包含数据中表的集合，通过“.”操作符可以选择相应的表，示例代码如下所示。

```
data.mynews                                                              //选择相应表
```

使用 LINQ 查询后运行结果如图 20-9 所示。

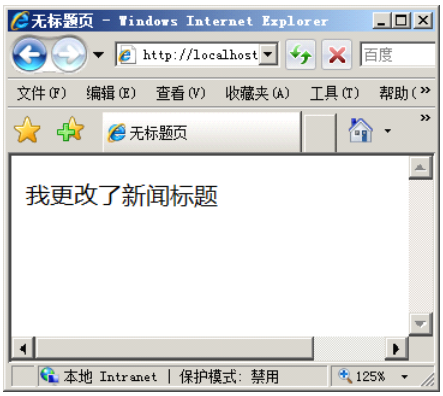

图 20-9　LINQ 执行数据库查询

使用 LINQ 技术能够方便的进行数据库查询和整合操作，LINQ 不仅能够实现类似 SQL 语句的查询操作，还能够支持.NET 编程方法进行数据查询条件语句的编写。使用 LINQ 技术进行数据查询的顺序如下所示：

- 创建 LINQ to SQL 文件：创建一个 LINQ to SQL 类文件进行数据集封装。
- 拖动数据表：将数据表拖动到 LINQ to SQL 类文件中，可以进行数据表的可视化操作。
- 使用 LINQ to SQL 类文件：使用 LINQ to SQL 类文件提供的数据集的封装进行数据操作。

使用 LINQ to SQL 类文件能够极快的创建一个 LINQ 到 SQL 数据库的映射并进行数据集对象的封装，开发人员能够使用面向对象的方法进行数据集操作并提供快速开发的解决方案。

20.2.3　IEnumerable 和 IEnumerable<T>接口

IEnumerable 和 IEnumerable<T>接口在.NET 中是非常重要的接口,它允许开发人员定义 foreach 语句功能的实现并支持非泛型方法的简单的迭代，IEnumerable 和 IEnumerable<T>接口是.NET Framework 中最基本的集合访问器，这两个接口对于 LINQ 的理解是非常重要的。

在面向对象的开发过程中，常常需要创建若干对象，并进行对象的操作和查询，在创建对象前，首先需要声明一个类为对象提供描述，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;                                          //使用 LINQ 命名控件
using System.Text;
namespace IEnumeratorSample
{
    class Person                                            //定义一个 Person 类
    {
        public string Name;                                 //定义 Person 的名字
        public string Age;                                 //定义 Person 的年龄
        public Person(string name, string age)              //为 Person 初始化（构造函数）
        {
            Name = name;                                    //配置 Name 值
            Age = age;                                      //配置 Age 值
        }
    }
```

上述代码定义了一个 Person 类并抽象一个 Person 类的属性，这些属性包括 Name 和 Age。Name 和

Age 属性分别用于描述 Person 的名字和年龄，用于数据初始化。初始化之后的数据就需要创建一系列 Person 对象，通过这些对象的相应属性能够进行对象的访问和遍历，示例代码如下所示。

```
class Program
{
    static void Main(string[] args)
    {
        Person[] per = new Person[2]                                    //创建并初始化 2 个 Person 对象
        {
            new Person("guojing","21"),                                 //通过构造函数构造对象
            new Person("muqing","21"),                                  //通过构造函数构造对象
        };
        foreach (Person p in per)                                       //遍历对象
            Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
        Console.ReadKey();
    }
}
```

上述代码创建并初始化了 2 个 Person 对象，并通过 foreach 语法进行对象的遍历。但是上述代码是在数组中进行查询的，就是说如果要创建多个对象，则必须创建一个对象的数组，如上述代码中的 Per 变量，而如果需要直接对对象的集合进行查询，却不能够实现查询功能。例如增加一个构造函数，该构造函数用户构造一组 Person 对象，示例代码如下所示。

```
private Person[] per;
public Person(Person[] array)                                           //重载构造函数,迭代对象
{
    per = new Person[array.Length];                                     //创建对象
    for (int i = 0; i < array.Length; i++)                              //遍历初始化对象
    {
        per[i] = array[i];                                              //数组赋值
    }
}
```

上述构造函数动态的构造了一组 People 类的对象，那么应该也能够使用 foreach 语句进行遍历，示例代码如下所示。

```
Person personlist = new Person(per);                                    //创建对象
foreach (Person p in personlist)                                        //遍历对象
{
    Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
}
```

在上述代码的 foreach 语句中，直接在 Person 类的集合中进行查询，系统则会报错“ConsoleApplication1.Person”不包含“GetEnumerator”的公共定义，因此 foreach 语句不能作用于“ConsoleApplication1.Person”类型的变量，因为 Person 类并不支持 foreach 语句进行遍历。为了让相应的类能够支持 foreach 语句执行遍历操作，则需要实现派生自类 IEnumerable 并实现 IEnumerable 接口，示例代码如下所示。

```
public IEnumerator GetEnumerator()                                      //实现接口
{
    return new GetEnum(_people);
}
```

为了让自定义类型能够支持 foreach 语句，则必须对 Person 类的构造函数进行编写并实现接口，示

例代码如下所示。

```
    class Person:IEnumerable                                    //派生自 IEnumerable,同样定义一个 PersonI 类
    {
        public string Name;                                     //创建字段
        public string Age;                                    //创建字段
        public Person(string name, string age)                  //字段初始化
        {
            Name = name;                                        //配置 Name 值
            Age = age;                                          //配置 Age 值
        }
        public IEnumerator GetEnumerator()                      //实现接口
        {
            return new PersonEnum(per);                         //返回方法
        }
}
```

上述代码重构了 Person 类并实现了接口，接口实现的具体方法如下所示。

```
    class PersonEnum : IEnumerator                              //实现 foreach 语句内部,并派生
    {
    public Person[] _per;                                       //实现数组
    int position = -1;                                          //设置“指针”
    public PersonEnum(Person[] list)
    {
        _per = list;                                            //实现 list
    }
    public bool MoveNext()                                      //实现向前移动
    {
        position++;                                             //位置增加
        return (position < _per.Length);                        //返回布尔值
    }
    public void Reset()                                         //位置重置
    {
        position = -1;                                          //重置指针为-1
    public object Current                                       //实现接口方法
    {
        get
        {
            try
            {
                return _per[position];                          //返回对象
            }
            catch (IndexOutOfRangeException)                    //捕获异常
            {
                throw new InvalidOperationException();          //抛出异常信息
            }
        }
    }
    }
```

上述代码实现了 foreach 语句的功能，当开发 Person 类初始化后就可以直接使用 Personal 类对象的集合进行 LINQ 查询，示例代码如下所示。

```
static void Main(string[] args)
{
    Person[] per = new Person[2]                    //同样初始化并定义 2 个 Person 对象
    {
        new Person("guojing","21"),                 //构造创建新的对象
        new Person("muqing","21"),                  //构造创建新的对象
    };
    Person personlist = new Person(per);            //初始化对象集合
    foreach (Person p in personlist)                //使用 foreach 语句
        Console.WriteLine("Name is " + p.Name + " and Age is " + p.Age);
    Console.ReadKey();
}
```

从上述代码中可以看出，初始化 Person 对象时初始化的是一个对象的集合，在该对象的集合中可以通过 LINQ 直接进行对象的操作，这样做即封装了 Person 对象也能够让编码更加易读。在.NET Framework 3.5 中，LINQ 支持数组的查询，开发人员不必自己手动创建 IEnumerable 和 IEnumerable<T>接口以支持某个类型的 foreach 编程方法，但是 IEnumerable 和 IEnumerable<T>是 LINQ 中非常重要的接口，在 LINQ 中也大量的使用 IEnumerable 和 IEnumerable<T>进行封装，示例代码如下所示。

```
public static IEnumerable<TSource> Where<TSource>
(this IEnumerable<TSource> source,Func<TSource, Boolean> predicate)      //内部实现
{
    foreach (TSource element in source)                                  //内部遍历传递的集合
    {
        if (predicate(element))
            yield return element;                                        //返回集合信息
    }
}
```

上述代码为 LINQ 内部的封装，从代码中可以看到，在 LINQ 内部也大量的使用了 IEnumerable 和 IEnumerable<T>接口实现 LINQ 查询。IEnumerable 原本就是.NET Framework 中最基本的集合访问器，而 LINQ 是面向关系（有序 N 元组集合）的，自然也就是面向 IEnumerable<T>的，所以了解 IEnumerable 和 IEnumerable<T>对 LINQ 的理解是有一定帮助的。

20.2.4 IQueryProvider 和 IQueryable<T>接口

IQueryable 和 IQueryable<T>同样是 LINQ 中非常重要的接口，在 LINQ 查询语句中，IQueryable 和 IQueryable<T>接口为 LINQ 查询语句进行解释和翻译工作，开发人员能够通过重写 IQueryable 和 IQueryable<T>接口以实现用不同的方法进行不同的 LINQ 查询语句的解释。

IQueryable<T>继承于 IEnumerable<T>和 IQueryable 接口，在 IQueryable 中包括两个重要的属性，这两个属性分别为 Expression 和 Provider。Expression 和 Provider 分别表示获取与 IQueryable 的实例关联的表达式目录树和获取与数据源关联的查询提供程序，Provider 作为其查询的翻译程序实现 LINQ 查询语句的解释。通过 IQueryable 和 IQueryable<T>接口，开发人员能够自定义 LINQ Provider。

注意：Provider 可以被看做是一个提供者，用于提供 LINQ 中某个语句的解释工具，在 LINQ 中通过编程的方法能够实现自定义 Provider。

在 IQueryable 和 IQueryable<T>接口中，还需要另外一个接口，这个接口就是 IQueryProvider，该接口用于分解表达式，实现 LINQ 查询语句的解释工作，这个接口也是整个算法的核心。IQueryable<T>接口在 MSDN 中的定义如下所示。

```
public interface IQueryable<T> : IEnumerable<T>, IQueryable, IEnumerable
{
}
public interface IQueryable : IEnumerable
{
   Type ElementType { get; }                                          //获取元素类型
   Expression Expression { get; }                                     //获取表达式
   IQueryProvider Provider { get; }                                   //获取提供者
}
```

上述代码定义了 IQueryable<T>接口的规范，用于保持数据源和查询状态，IQueryProvider 在 MSDN 中定义如下所示。

```
public interface IQueryProvider
{
   IQueryable CreateQuery(Expression expression);                     //创建可执行对象
   IQueryable<TElement> CreateQuery<TElement>(Expression expression); //创建可执行对象
   object Execute(Expression expression);                             //计算表达式
   TResult Execute<TResult>(Expression expression);                   //计算表达式
}
```

IQueryProvider 用于 LINQ 查询语句的核心算法的实现，包括分解表达式和表达式计算等。为了能够创建自定义 LINQ Provider，可以编写接口的实现。示例代码如下所示。

```
public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
{
    query.expression = expression;                                    //声明表达式
    return (IQueryable<TElement>)query;                               //返回 query 对象
}
```

上述代码用于构造一个可用来执行表达式计算的 IQueryable 对象，在接口中可以看到需要实现两个相同的执行表达式的 IQueryable 对象，另一个则是执行表达式对象的集合，其实现代码如下所示。

```
public IQueryable CreateQuery(Expression expression)
{
    return CreateQuery<T>(expression);                                //返回表达式的集合
}
```

而作为表达式解释和翻译的核心接口，则需要通过算法实现相应 Execute 方法，示例代码如下所示。

```
public TResult Execute<TResult>(Expression expression)
{
    var exp = expression as MethodCallExpression;                     //创建表达式对象
    var data = ((exp.Arguments[0] as ConstantExpression).Value as MyQuery<T>).Data;
    var func = (exp.Arguments[1] as UnaryExpression).Operand as Expression
    <System.Func<T, bool>>;
    var lambda = Expression.Lambda<Func<T, bool>>(func.Body, func.Parameters[0]);
    var r = data.Where(lambda.Compile());                             //编译表达式
    return (TResult)r.GetEnumerator();
}
```

上述代码通过使用 lambda 表达式进行表达式的计算，实现了 LINQ 中查询的解释功能。在 LINQ 中，对于表达式的翻译和执行过程都是通过 IQueryProvider 和 IQueryable<T>接口来实现的。IQueryProvider 和 IQueryable<T>实现用户表达式的翻译和解释，在 LINQ 应用程序中，通常无需通过 IQueryProvider 和 IQueryable<T>实现自定义 LINQ Provider，因为 LINQ 已经提供强大表达式查询和计算功能。了解 IQueryProvider 和 IQueryable<T>接口有助于了解 LINQ 内部是如何执行的。

20.2.5 LINQ 相关的命名空间

LINQ 开发为开发人员提供了便利，可以让开发人员以统一的方式对 IEnumerable<T>接口的对象、数据库、数据集以及 XML 文档进行访问。从整体上来说，LINQ 是这一系列访问技术的统称，对于不同的数据库和对象都有自己的 LINQ 名称，例如 LINQ to SQL、LINQ to Object 等等。当使用 LINQ 操作不同的对象时，可能使用不同的命名空间。常用的命名空间如下所示。

- System.Data.Linq： 该命名空间包含支持与 LINQ to SQL 应用程序中的关系数据库进行交互的类。
- System.Data.Linq.Mapping：该命名空间包含用于生成表示关系数据库的结构和内容的 LINQ to SQL 对象模型的类。
- System.Data.Linq.SqlClient：该命名空间包含与 SQL Server 进行通信的提供程序类，以及包含查询帮助器方法的类。
- System.Linq：该命名空间提供支持使用语言集成查询 (LINQ)进行查询的类和接口。
- System.Linq.Expression：该命名空间包含一些类、接口和枚举，它们使语言级别的代码表达式能够表示为表达式树形式的对象。
- System.Xml.Linq：包含 LINQ to XML 的类，LINQ to XML 是内存中的 XML 编程接口。

LINQ 中常用的命名空间为开发人员提供 LINQ 到数据库和对象的简单的解决方案，开发人员能够通过这些命名空间提供的类进行数据查询和整理，这些命名空间统一了相应的对象的查询方法，如数据集和数据库都可以使用类似的 LINQ 语句进行查询操作。

20.3 Lambda 表达式

Lambda 表达式是一种高效的类似于函数式编程的表达式，Lambda 简化了开发中需要编写的代码量 。Lambda 表达式是由.NET 2.0 演化过来的，也是 LINQ 的基础，熟练的掌握 Lambda 表达式能够快速的上手 LINQ 应用开发。

20.3.1 匿名方法

在了解 Lambda 表达式之前，需要了解什么是匿名方法，匿名方法简单的说就是没有名字的方法，而通常情况下的方法定义是需要名字的，示例代码如下所示。

```
public int sum(int a, int b)                                    //创建方法
{
    return a + b;                                               //返回值
}
```

上面这个方法就是一个常规方法，这个方法需要方法修饰符（public）、返回类型（int）方法名称（sum）和参数列表。而匿名方法可以看作是一个委托的扩展，是一个没有命名的方法，示例代码如下所示。

```
delegate int Sum(int a,int b);                                  //声明匿名方法
protected void Page_Load(object sender, EventArgs e)
{
    Sum s = delegate(int a,int b)                               //使用匿名方法
    {
```

```
            return a + b;                                    //返回值
        };
    }
```

上述代码声明了一个匿名方法 Sum 但是没有实现匿名方法的操作的实现，在声明匿名方法对象时，可以通过参数格式创建一个匿名方法。匿名方法能够通过传递的参数进行一系列操作，示例代码如下所示。

```
        Response.Write(s(5,6).ToString());
```

上述代码使用了 s(5,6)方法进行两个数的加减，匿名方法虽然没有名称，但是同样可以使用“（”“）”号进行方法的使用。

注意：虽然匿名方法没有名称，但是编译器在编译过程中，还是会为该方法定义一个名称，只是在开发过程中这个名称是不被开发人员所看见的。

除此之外，匿名方法还能够使用一个现有的方法作为其方法的委托，示例代码如下所示。

```
    delegate int Sum(int a,int b);                           //方法委托
    public int retSum(int a, int b)                          //普通方法
    {
        return a + b;
    }
```

上述代码声明了一个匿名方法，并声明了一个普通的方法，在代码中使用匿名方法代码如下所示。

```
    protected void Page_Load(object sender, EventArgs e)
    {
        Sum s = retSum;                                      //使用匿名方法
        int result = s(10, 10);
    }
```

从上述代码中可以看出，匿名方法是一个没有名称的方法，但是匿名方法可以将方法名作为参数进行传递，如上述代码中变量 s 就是一个匿名方法，这个匿名方法的方法体被声明为 retSum，当编译器进行编译时，匿名方法会使用 retSum 执行其方法体并进行运算。匿名方法最明显的好处就是可以降低常规方法编写时的工作量，另外一个好处就是可以访问调用者的变量，降低传参数的复杂度。

20.3.2 Lambda 表达式基础

在了解了匿名方法后，就能够开始了解 Lambda 表达式，Lambda 表达式在一定程度上就是匿名方法的另一种表现形式。为了方便对 Lambda 表达式的解释，首先需要创建一个 People 类，示例代码如下所示。

```
public class People
{
    public int age { get; set; }                             //设置属性
    public string name { get; set; }                         //设置属性
    public People(int age,string name)                       //设置属性（构造函数构造）
    {
        this.age = age;                                      //初始化属性值 age
        this.name = name;                                    //初始化属性值 name
    }
}
```

上述代码定义了一个 People 类，并包含一个默认的构造函数能够为 People 对象进行年龄和名字的定义。在应用程序设计中，很多情况下需要创建对象的集合，创建对象的集合有利于对对象进行操作和

排序等操作，以便在集合中筛选相应的对象。使用 List 进行泛型编程，可以创建一个对象的集合，示例代码如下所示。

```
List<People> people = new List<People>();                    //创建泛型对象
People p1 = new People(21,"guojing");                        //创建一个对象
People p2 = new People(21, "wujunmin");                      //创建一个对象
People p3 = new People(20, "muqing");                        //创建一个对象
People p4 = new People(23, "lupan");                         //创建一个对象
people.Add(p1);                                              //添加一个对象
people.Add(p2);                                              //添加一个对象
people.Add(p3);                                              //添加一个对象
people.Add(p4);                                              //添加一个对象
```

上述代码创建了 4 个对象，这 4 个对象分别初始化了年龄和名字，并添加到 List 列表中。当应用程序需要对列表中的对象进行筛选时，例如需要筛选年龄大于 20 岁的人时，就需要从列表中筛选，示例代码如下所示。

```
IEnumerable<People> results = people.Where(delegate(People p) { return p.age > 20; });//匿名方法
```

上述代码通过使用 IEnumerable 接口创建了一个 result 集合，并且该集合中填充的是年龄大于 20 的 People 对象。细心的读者就能够发现在这里使用了一个匿名方法进行筛选，因为该方法没有名称，该匿名方法通过使用 People 类对象的 age 字段进行筛选。

虽然上述代码中执行了筛选操作，但是使用匿名方法往往不太容易理解和阅读，而 Lambda 表达式相比于匿名方法而言更加容易理解和阅读，示例代码如下所示。

```
IEnumerable<People> results = people.Where(People => People.age > 20);          //Lambda
```

上述代码同样返回了一个 People 对象的集合给变量 results，但是其编写的方法更加容易阅读，这里可以看出 Lambda 表达式在编写的格式上和匿名方法非常相似。其实当编译器开始编译并运行，Lambda 表达式最终也表现为匿名方法。

使用匿名方法实际上并不是创建了没有名称的方法，实际上编译器会创建一个方法，这个方法对于开发人员来说是看不见的，该方法会将 People 类的对象中符合 p.age>20 条件的对象返回并填充到集合中。相同的是，使用 Lambda 表达式，当编译器编译时，Lambda 表达式同样会被编译成一个匿名方法进行相应的操作，但是相比于匿名方法而言，Lambda 表达式更容易阅读，Lambda 表达式的格式如下所示。

```
(参数列表)=>表达式或者语句块
```

如上述代码中，参数列表就是 People 类，表达式和语句块就是 People.age>20，使用 Lambda 表达式能够让人很容易的理解该语句究竟是如何执行的，虽然匿名方法提供了同样的功能，却并不容易理解。相比之下 People => People.age > 20 却能够很好的理解为“返回一个年纪大于 20 的人”。其实 Lambda 表达式并没有什么高深的技术，Lambda 表达式可以看作是匿名方法的另一种表现形式。其实 Lambda 表达式经过反编译后，与匿名方法并没有什么区别。

20.3.3 Lambda 表达式格式

Lambda 表达式是匿名方法的另一种表现形式。比较 Lambda 表达式和匿名方法，在匿名方法中“（”，“）”内是方法的参数的集合，这就对应了 Lambda 表达式中“(参数列表)”，而匿名方法中“{”，“}”内是方法的语句块，这也对应了 Lambda 表达式“=>”符号右边的表达式和语句块项。由于 Lambda 表达式是一种匿名方法，所以 Lambda 表达式也包含一些基本格式，这些基本格式如下所示。

Lambda 表达式可以有多个参数，一个参数，或者无参数。其参数类型可以隐式或者显式。示例代码如下所示：

```
(x, y) => x * y                                              //多参数，隐式类型=> 表达式
```

```
x => x * 5                                  //单参数，  隐式类型=>表达式
x => { return x * 5; }                      //单参数，隐式类型=>语句块
(int x) => x * 5                            //单参数，显式类型=>表达式
(int x) => { return x * 5; }                //单参数，显式类型=>语句块
() => Console.WriteLine()                   //无参数
```

上述格式都是 Lambda 表达式的合法格式，在编写 Lambda 表达式时，可以忽略参数的类型，因为编译器能够根据上下文直接推断参数的类型，示例代码如下所示。

```
(x, y) => x + y                             //多参数，隐式类型=>  表达式
```

Lambda 表达式的主体可以是表达式也可以是语句块，这样就节约了代码的编写。

注意：Lambda 表达式与匿名方法的另一个不同是，Lambda 表达式的主体可以是表达式也可以是语句块，而匿名方法中不能包含表达式。

Lambda 表达式中的表达式和表达式体都能够被转换成表达式树，这在表达式树的构造上会起到很好的作用，表达式树也是 LINQ 中最基本最重要的概念。

20.3.4 Lambda 表达式树

Lambda 表达式树也是 LINQ 中最重要的一个概念，Lambda 表达式树允许开发人员像处理数据一样对 Lambda 表达式进行修改。理解 Lambda 表达式树的概念并不困难，Lambda 表达式树就是将 Lambda 表达式转换成树状结构，在使用 Lambda 表达式树之前还需要使用 System.Linq.Expressions 命名空间，示例代码如下所示。

```
using System.Linq.Expressions;                                  //使用命名空间
```

Lambda 表达式树的基本形式有两种，这两种形式代码如下所示。

```
        Func<int, int> func = pra => pra * pra;                 //创建表达式树
        Expression<Func<int, int>> expression = pra => pra * pra;   //创建表达式树
```

Lambda 表达式树就是将 Lambda 表达式转换成树状结构，示例代码如下所示。

```
        Func<int, int> func = (pra => pra * pra);               //创建表达式
        Response.Write(func(8).ToString());                     //执行表达式
        Response.Write("<hr/>");
```

上述代码直接用 Lambda 表达式初始化 Func 委托，运行后返回的结果为 64，同样使用 Expression 类也可以实现相同的效果，示例代码如下所示。

```
        Expression<Func<int, int>> expression = pra => pra * pra;   //创建表达式
        Func<int, int> func1 = expression.Compile();            //编译表达式
        Response.Write(func1(8).ToString());
```

上述代码运行后同样返回 64，运行后如图 20-10 所示。使用 Func 类和 Expression 类创建 Lambda 表达式运行结果基本相同，但是 Func 方法和 Expression 方法是有区别的，如 Lambda 表达式 pra => pra * pra，Expression 首先会分析该表达式并将表达式转换成树状结构，如图 20-11 所示。

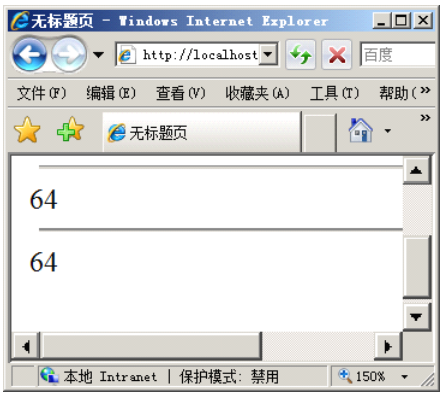

图 20-10　Lambda 表达式树

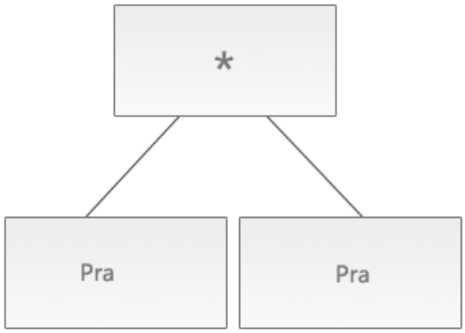

图 20-11　Lambda 表达式树格式

当编译器编译 Lambda 表达式时，如果 Lambda 表达式使用的是 Func 方法，则编译器会将 Lambda 表达式直接编译成匿名方法，而如果 Lambda 表达式使用的是 Expression 方法，则编译器会将 Lambda 表达式进行分析、处理然后得到一种数据结构。

20.3.5　访问 Lambda 表达式树

既然在 LINQ 应用开发中常常需要解析 Lambda 表达式，则就不能避免的对 Lambda 表达式树进行访问，访问 Lambda 表达式的方法非常简单，直接将表达式输出即可，示例代码如下所示。

```
Expression<Func<int, int>> expression = pra => pra * pra;          //创建表达式树
Func<int, int> func1 = expression.Compile();                       //表达式树编译
Response.Write(func1(8).ToString());                               //执行表达式
Response.Write(expression.ToString());                             //执行表达式
```

上述代码直接使用 Expression 类的对象进行表达式输出，这时候读者可能会想到，是否能够像 Expression 类的对象一样直接将 Func 对象进行输出，答案是否定的，而如果直接使用 Func 对象是不能够输出表达式的，如图 20-12 所示。

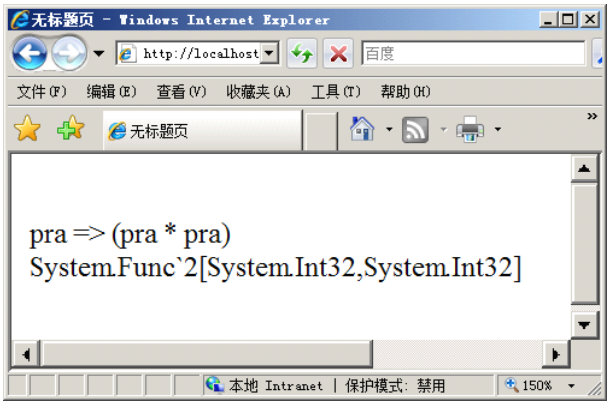

图 20-12　Lambda 表达式输出

正如图 20-12 所示，直接输出 Expression 类的对象的值能够进行表达式的输出，而如果输出 Func 类的对象只会输出系统信息，如 System.Func`2[System.Int32,System.Int32]，这并不是期待的结果。这也从另一个角度说明了 Func 方法和 Expression 方法是有区别的。

20.4 小结

本章介绍了 LINQ 的起源，包括什么是 LINQ，以及 LINQ 在.NET 3.5 Framework 中的位置，本章还介绍了 LINQ 基础，包括在 LINQ 中常用的接口和类，以及使用 LINQ 需要的命名空间。本章还包括：

- 创建使用 LINQ 的 Web 应用程序：简单的介绍了使用 LINQ 实现 Web 应用程序中的查询功能。
- 基本的 LINQ 数据查询：介绍了 LINQ 基本查询功能。
- Lambda 表达式基础：介绍了 Lambda 表达式基础，以及何为 Lambda 表达式。
- Lambda 表达式格式：介绍了 Lambda 表达式书写中需要使用的格式。
- Lambda 表达式树：介绍了 Lambda 表达式树。

在介绍 LINQ 之前还介绍了 Lambda 表达式，Lambda 表达式是 LINQ 的基础，加深对 Lambda 表达式的理解能够更加容易的理解 LINQ 的内部机制和原理，实际上，LINQ 的使用并不是非常难，关于 LINQ 查询将会在下一章中讲到。