的通知。所有通知都发送完毕之后，代码清单13-9通过引发一个AggregateException来报告所有已发生的异常。AggregateException包装了一个异常集合。集合中的异常可以通过InnerExceptions属性访问。用这种方法，所有异常都得到报告，同时所有订阅者都不会错过通知。

13.1.9 方法返回值和传引用

还有一种值得注意的情形，在这种情形下，也有必要遍历委托调用列表，而非直接激活一个通知。这种情形涉及的委托要么不返回void，要么有一个ref或out参数。在Thermostat例子中，OnTemperatureChange委托是Action<float>类型，它返回void，而且没有ref或out参数。其结果是没有数据返回给发布者。这一点相当重要，因为调用委托可能造成将一个通知发送给多个订阅者。假如订阅者会返回一个值，就无法确定应该使用哪个订阅者的返回值。

假如修改OnTemperatureChange，让它不是返回void，而是返回枚举值，指出设备是否因温度的改变而启动，那么新委托就是Func<float, Status>类型，其中Status是包含元素On和Off的枚举。所有订阅者方法都要使用和委托相同的方法签名，所以都必须返回状态值。由于OnTemperatureChange潜在地对应于一个委托链，所以需要遵循和执行错误处理时相同的模式。也就是说，必须使用GetInvocationList()方法遍历每个委托调用列表来获取每个单独的返回值。类似地，使用ref和out参数的委托类型也需要特别对待。然而，虽然在极少数情况下需要，但一般原则是通过返回void来彻底避免该情形。

13.2 事件

到目前为止，使用的委托都存在两个关键的问题。C#使用关键字event（事件）来解决这些问题。本节描述了如何使用事件，以及它们是如何工作的。

13.2.1 事件的作用

本章前面已全面描述了委托是如何工作的。然而，委托结构中存在的缺陷可能造成程序员在不经意中引入一个bug。这个问题和封装有关，无论事件的订阅还是发布，都不能得到充分的控制。

1.封装订阅

如前所述，可以使用赋值操作符将一个委托赋给另一个。遗憾的是，这同时可能造成bug。来看看代码清单13-10的例子。

代码清单13-10 错误地使用赋值操作符=而不是+=

```
class Program
{
  public static void Main()
  {
      Thermostat thermostat = new Thermostat();
      Heater heater = new Heater(60);
      Cooler cooler = new Cooler(80);
      string temperature;

      // Note: Use new Action(
```

```
        //      cooler.OnTemperatureChanged) if C# 1.0
        thermostat.OnTemperatureChange =
            heater.OnTemperatureChanged;

        // Bug:  assignment operator overrides
        // previous assignment.
        thermostat.OnTemperatureChange =
            cooler.OnTemperatureChanged;

        Console.Write("Enter temperature: ");
        temperature = Console.ReadLine();
        thermostat.CurrentTemperature = int.Parse(temperature);
    }
}
```

代码清单13-10和代码清单13-6如出一辙，只是它不是使用+=操作符，而是使用一个简单赋值操作符。其结果就是，当代码将cooler.OnTemperatureChanged赋给OnTemperatureChange时，heater.OnTemperatureChanged会被清除，因为一个全新的委托链替代了之前的链。在本该使用+=操作符的地方使用了赋值操作符“=”，由于这是一个十分容易犯的错误，所以最好的解决方案就是仅为包容类内部的对象提供对赋值操作符的支持。event关键字的作用就是提供额外的封装，避免不小心地取消其他订阅者。

2.封装发布

委托和事件的第二个重要区别在于，事件确保只有包容类才能触发事件通知。来看看代码清单13-11的例子。

代码清单13-11 从事件包容者的外部触发事件

```
class Program
{
  public static void Main()
  {
      Thermostat thermostat = new Thermostat();
      Heater heater = new Heater(60);
      Cooler cooler = new Cooler(80);
      string temperature;

      // Note: Use new Action(
      //       cooler.OnTemperatureChanged) if C# 1.0.
      thermostat.OnTemperatureChange +=
          heater.OnTemperatureChanged;
      thermostat.OnTemperatureChange +=
          cooler.OnTemperatureChanged;

      thermostat.OnTemperatureChange(42);
    }
}
```

在代码清单13-11中，即使thermostat的CurrentTemperature没有发生改变，Program也能调用OnTemperatureChange委托。因此，Program触发了对所有thermostat订阅者的一个通知，告诉它们温度已发生改变，而实际上thermostat的温度并没有变化。和之前一样，委托的问题在于封装不充分。Thermostat应禁止其他任何类调用OnTemperatureChange委托。

13.2.2 事件的声明

C#用event关键字解决了上述两个问题。虽然看起来像是一个字段修饰符，但event定义的是一个新的成员类型，如代码清单13-12所示。

代码清单13-12 为Event-Coding（事件-编码）模式使用event关键字

```
public class Thermostat
{
    public class TemperatureArgs: System.EventArgs
    {
        public TemperatureArgs( float newTemperature )
        {
            NewTemperature = newTemperature;
        }

        public float NewTemperature
        {
            get{return _newTemperature;}
            set{_newTemperature = value;}
        }
        private float _newTemperature;
    }

    // Define the event publisher
    public event EventHandler<TemperatureArgs> OnTemperatureChange =
        delegate { };

    public float CurrentTemperature
    {
        ...
    }
    private float _CurrentTemperature;
}
```

这个新的Thermostat类进行了4处修改。首先，OnTemperatureChange属性被移除了。OnTemperatureChange被声明为一个public字段。从表面看，这似乎并不是在解决早先描述的封装问题。现在需要的是增强封装，而不是让一个字段变成public字段来削弱封装。然而，我们进行的第二处修改是在字段声明之前添加event关键字。这一处简单的修改提供了所需的全部封装。添加event关键字后，会禁止为一个public委托字段使用赋值操作符（比如thermostat.OnTemperatureChange = cooler.OnTemperatureChanged）。除此之外，只有包容类才能调用向所有订阅者发出通知的委托（例如，不允许在类的外部执行thermostat.OnTemperatureChange (42)）。换言之，event关键字提供了必要的封装来防止任何外部类发布一个事件或者取消之前不是由其添加的订阅者。这样，就完美地解决了普通委托存在的两个问题，这是在C#中提供event关键字的关键原因之一。

普通委托的另一个不利之处在于，很容易忘记在调用委托之前检查null值。这会引发一个非预期的NullReferenceException异常。幸好，通过event关键字提供的封装，可以在声明时（或者在构造器中）采用一个替代方案，如代码清单13-12所示。注意在声明事件时，我们赋的值是delegate { }，它是一个空委托，代表由零个侦听者构成的一个集合。通过赋值一个空委托，就可以引发事件而不必检查是否有任何侦听者。（这个行为类似于向变量赋一个包含零个元素的数组。然后，在调用一个数组成员时，就不必先检查变量是

否为null。）当然，如果委托存在被重新赋值为null的任何可能，那么仍需进行null值检查。不过，由于event关键字限制赋值只能在类的内部发生，所以要重新对委托进行赋值，只能在类中进行。如果从未在类中赋过null值，就不必在代码每次调用委托时检查null。

13.2.3 编码规范

为了获得希望的功能，唯一要做的就是将原始委托变量声明更改为字段，然后添加event关键字。进行了这两处修改之后，就可以提供全部必要的封装。与此同时，其他所有功能都和以前一样。然而，在代码清单13-12中，委托声明还进行了另一处修改。为了遵循标准的C#编码规范，要将Action<float>替换成一个新的委托类型EventHandler<TemperatureArgs>，这是一个CLR类型，其声明如代码清单13-13所示（自.NET Framework 2.0起添加）。

代码清单13-13 声明一个泛型委托类型

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
   where TEventArgs : EventArgs;
```

结果是Action<TEventArgs>委托类型中的单个温度参数被替换成两个新参数，一个代表发送者，另一个代表事件数据。这一处修改并不是C#编译器强制的。但是，声明一个打算作为事件来使用的委托时，规范是要求传递这些类型的两个参数。

第一个参数sender应包含调用委托的那个类的实例。假如一个订阅者方法注册了多个事件，这个参数就尤其有用。例如，假定两个Thermostat实例都订阅了heater.OnTemperatureChanged事件。在这种情况下，任何一个Thermostat实例都可能触发对heater.OnTemperatureChanged的调用。为了判断具体是哪个Thermostat实例触发了事件，要在Heater.OnTemperatureChanged()内部利用sender参数进行判断。如果事件是静态的，就无法做这种判断，所以要为sender传递null值作为实参。

第二个参数TEventArgse是Thermostat.TemperatureArgs类型。关于TemperatureArgs，一个重点在于它是从System.EventArgs派生的。（事实上，一直到.NET Framework 4.5，都通过一个泛型约束来强制从System.EventArgs派生。）System.EventArgs唯一重要的属性是Empty，它用于指出不存在事件数据。然而，从System.EventArgs派生出TemperatureArgs时添加了一个额外的属性，名为NewTemperature，用于将温度从自动调温器传递给订阅者。

这里简单总结一下事件的编码规范：第一个参数sender是object类型的，它包含对调用委托的那个对象的一个引用（静态事件则为null）。第二个参数是System.EventArgs类型的（或者从System.EventArgs派生，但包含了事件的附加数据）。调用委托的方式和以前几乎完全一样，只是要提供附加的参数。代码清单13-14展示了一个例子。

代码清单13-14 触发事件通知

```
public class Thermostat
{
  ...
  public float CurrentTemperature
  {
      get{return _CurrentTemperature;}
      set
```

```
        {
            if (value != CurrentTemperature)
            {
                _CurrentTemperature = value;
                // If there are any subscribers
                // then notify them of changes in
                // temperature
                if(OnTemperatureChange != null)
                {
                    // Call subscribers
                    OnTemperatureChange(
                        this, new TemperatureArgs(value) );
                }
            }
        }
    }
    private float _CurrentTemperature;
}
```

通常将sender指定为容器类（this），因为它是唯一一个能为事件调用委托的类。

在这个例子中，订阅者可以将sender参数转型为Thermostat，并以那种方式访问当前温度，或者通过TemperatureArgs实例来访问。然而，Thermostat实例上的当前温度可能由一个不同的线程改变。在由于状态改变而发生事件的时候，连同新值传递前一个值是常见的编程模式，它可以控制哪些状态变化是允许的。

规范

要在调用委托前检查它的值不为null。

不要为非静态事件的sender传递null值。

要为静态事件的sender传递null值。

不要为eventArgs传递null值。

要为事件使用EventHandler<TEventArgs>委托类型。

要为TEventArgs使用System.EventArgs类型或者它的派生类型。

考虑使用System.EventArgs的子类作为事件的实参类型（TEventArgs），除非完全确定事件永远不需要携带任何数据。

13.2.4 泛型和委托

上一节指出，为事件定义类型的规范是使用EventHandler<TEventArgs>委托类型。理论上任何委托类型都可以使用，但根据约定，第一个参数sender是object类型，第二个参数e是从System.EventArgs派生的一个类型。对于C# 1.0中的委托来说，一个较大的麻烦在于，一旦事件处理程序的参数发生改变，就不得不声明一个新的委托类型。每次从System.EventArgs派生（这是相当常见的一个情形），都要声明一个新的委托数据类型来使用新的EventArgs派生类型。例如，为了使用代码清单13-14的事件通知代码中的TemperatureArgs，必须声明委托类型TemperatureChangeHandler，它以TemperatureArgs作为参数，如代码清单13-15所示。

代码清单13-15 使用一个自定义委托类型

```
public class Thermostat
{
  public class TemperatureArgs: System.EventArgs
  {
      public TemperatureArgs( float newTemperature )
      {
          NewTemperature = newTemperature;
      }
      public float NewTemperature
      {
          get{return _newTemperature;}
          set{_newTemperature = value;}
      }
      private float _newTemperature;
  }

  public delegate void TemperatureChangeHandler(
      object sender, TemperatureArgs newTemperature);

  public event TemperatureChangeHandler
      OnTemperatureChange;

  public float CurrentTemperature
  {
      ...
  }
  private float _CurrentTemperature;
}
```

虽然通常应优先使用EventHandler<TEventArgs>，而非创建TemperatureChangeHandler这样的自定义委托类型，但后者也是有一些优点的。具体地说，如果使用自定义类型，就可以使用事件特有的参数名。例如代码清单13-15调用委托来引发事件时，第二个参数名是newTemperature，而不是一个让人摸不着头脑的e。

使用自定义委托类型的另一个原因涉及C# 2.0之前定义的CLR API。因为在框架中相当大一部分都是比较常见的类型，那么在处理来自于CLR API的事件时，就不难遇到具体的委托类型而不是事件的泛型形式。但无论如何，在C# 2.0和之后使用事件的大多数情形中，都没必要声明自定义委托数据类型。

规范

要为事件处理程序使用System.EventHandler<T>而非手动创建新的委托类型，除非自定义类型的参数名能提供有意义的说明。

高级主题：事件的内部机制

事件限制外部类只能通过“+=”操作符向发布者添加订阅方法，并用“-=”操作符取消订阅，除此之外的任何事情都不允许做。此外，它们还禁止除包容类之外的其他任何类调用事件。为了达到上述目的，C#编译器会获取带有event修饰符的public

委托变量，并将委托声明为private。此外，它还添加了两个方法和两个特殊的事件块。从本质上说，event关键字是编译器用于生成恰当封装逻辑的一个C#快捷方式。来看看代码清单13-16的事件声明示例。

代码清单13-16 声明OnTemperatureChange事件

```
public class Thermostat
{
  public event EventHandler<TemperatureArgs> OnTemperatureChange

  ...
}
```

C#编译器一旦遇到event关键字，就会生成与代码清单13-17的C#代码等价的CIL代码。

代码清单13-17 与编译器生成的事件CIL代码对应的C#代码

```
public class Thermostat
{
  // ...
  // Declaring the delegate field to save the
  // list of subscribers.
  private EventHandler<TemperatureArgs> _OnTemperatureChange;

  public void add_OnTemperatureChange(
      EventHandler<TemperatureArgs> handler)
  {
      System.Delegate.Combine(_OnTemperatureChange, handler);
  }

  public void remove_OnTemperatureChange(
      EventHandler<TemperatureArgs> handler)
  {
      System.Delegate.Remove(_OnTemperatureChange, handler);
  }

  public event EventHandler<TemperatureArgs> OnTemperatureChange
  {
      add
      {
          add_OnTemperatureChange(value)
      }
      remove
      {
          remove_OnTemperatureChange(value)
      }
  }
}
```

换言之，代码清单13-16的代码会造成编译器自动对代码进行扩展，并生成大致如代码清单13-17所示的代码。（“大致”一词不可或缺，因为为了简化问题，和线程同步有关的细节从代码清单中拿掉了。）

C#编译器首先获取原始事件定义，并在原位置定义一个私有委托变量。结果，从任何外部类中都无法使用该委托——即使是从它的派生类中。

接着，C#编译器定义了两个方法：add_OnTemperatureChange()和remove_

OnTemperatureChange()。其中，OnTemperatureChange后缀是从原始事件名称中截取的。这两个方法分别负责实现“+=”和“-=”赋值操作符。如代码清单13-17所示，这两个方法是使用本章前面讨论的静态方法System.Delegate.Combine()和System.Delegate.Remove()来实现的。传给方法的第一个参数是私有的EventHandler<TemperatureArgs>委托实例OnTemperatureChange。

在从event关键字生成的代码中，或许最奇怪的就是最后一部分。其语法与属性的取值和赋值方法非常相似，只是方法名变成了add和remove。其中，add块负责处理“+=”操作符，将调用传给add_OnTemperatureChange()。类似地，remove块处理“-=”操作符，将调用传给remove_OnTemperatureChange()。

必须重视这段代码与属性代码的相似性。本书前面讲过，C#在实现一个属性时，会创建get_<*propertyname*>和set_<*propertyname*>，然后将对get和set块的调用传给这些方法。显然，事件的语法与此非常相似。

另外要注意，在最终的CIL代码中，仍然保留了event关键字。换言之，事件是CIL代码能够显式识别的一样东西，并非只是一个C#构造。在CIL代码中保留一个等价的event关键字之后，所有语言和编辑器都能将事件识别为一个特殊的类成员，并正确地处理它。

13.2.5 自定义事件的实现

编译器为+=和-=生成的代码是可以自定义的。例如，假定改变OnTemperatureChange委托的作用域，使它成为protected而不是private。这样一来，从Thermostat派生的类也能直接访问委托，而无需受到和外部类一样的限制。为此，C#允许使用和代码清单13-15一样的属性语法。换言之，C#允许添加自定义的add和remove块，为事件封装的各个组成部分提供你自己的实现。代码清单13-18展示了一个例子。

代码清单13-18 自定义add和remove处理程序

```
public class Thermostat
{
  public class TemperatureArgs: System.EventArgs
  {
      ...
  }

    // Define the event publisher
  public event EventHandler<TemperatureArgs> OnTemperatureChange
  {
      add
      {
          System.Delegate.Combine(value, _OnTemperatureChange);
      }
      remove
      {
          System.Delegate.Remove(_OnTemperatureChange, value);
      }
  }
  protected EventHandler<TemperatureArgs> _OnTemperatureChange;

  public float CurrentTemperature
  {
```

```
            ...
        }
        private float _CurrentTemperature;
    }
```

在这个例子中，存储每个订阅者的委托_OnTemperatureChange变成了protected。除此之外，在add块的实现中，我们交换了两个委托存储的位置，使添加到链中的最后一个委托是接收通知的第一个委托。

13.3 小结

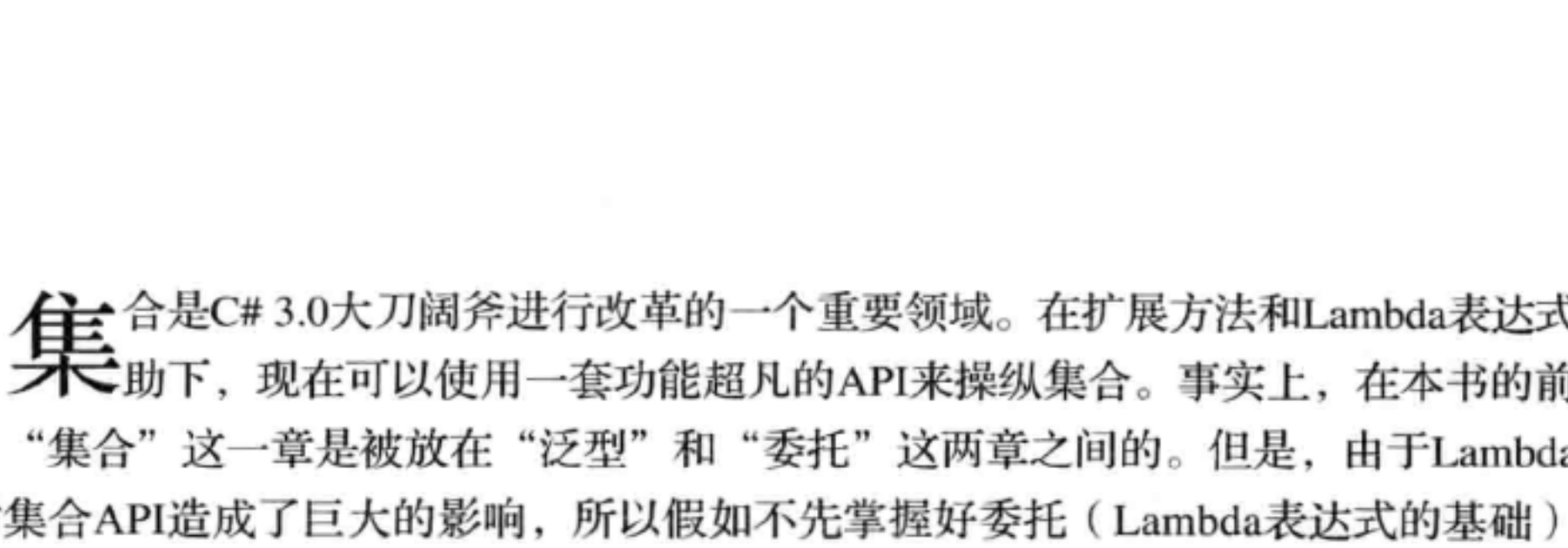
本章描述了事件，值得一提的是，通常，方法引用是唯一可以在事件上下文的外部用到委托变量的情况。换句话说，由于事件提供了额外的封装性，而且允许在必要时对实现进行自定义，所以最佳做法就是始终为Observer模式使用事件。

可能需要一段时间的练习，才能脱离示例代码，熟练地进行事件的编程。然而，事件编程正是以后要讲述的异步、多线程编程的重要基础。

第14章

支持标准查询操作符的集合接口

集合是C# 3.0大刀阔斧进行改革的一个重要领域。在扩展方法和Lambda表达式的帮助下，现在可以使用一套功能超凡的API来操纵集合。事实上，在本书的前几版中，“集合”这一章是被放在“泛型”和“委托”这两章之间的。但是，由于Lambda表达式对集合API造成了巨大的影响，所以假如不先掌握好委托（Lambda表达式的基础），集合是不好理解的。通过第12章和第13章的学习，你已经对Lambda表达式有了很好的理解，所以从本章起，将连续使用3章的篇幅来详细讨论集合。



本章首先讨论了匿名类型和集合初始化器（collection initializer），第5章曾以“高级主题”的形式简单介绍过这两个概念。接着，本章探讨了各种集合接口及其相互关系。这是理解集合的基础，请务必掌握。在讲解集合接口的同时，还介绍了C# 3.0新增的由IEnumerable<T>定义的扩展方法，这些方法实现了标准查询操作符。

有两套与集合相关的类和接口：支持泛型的和不支持泛型的。本章主要讨论泛型集合接口。通常，只有在你写的组件需要与老版本的“运行时”交互时，才应该使用不支持泛

型的集合类。这是由于任何非泛型的东西，现在都有了一个强类型的泛型替代物。虽然我们讨论的概念仍然适用于两种形式，但本书不会再显式地讨论非泛型版本[①]。

14.1 匿名类型和隐式类型的局部变量声明

C# 3.0显著增强了对集合的处理。令人印象深刻的是，为了支持这个高级API，语言本身只进行了不到9处增强。然而，正是由于这些增强，才使得C# 3.0成为语言发展史上的一个重要里程碑。其中两处增强是匿名类型和隐式局部变量。

14.1.1 匿名类型

匿名类型是由编译器声明的数据类型，而不是像第5章那样通过显式的类定义来声明。和匿名函数相似，当编译器看到匿名类型时，会执行一些后台操作，为你生成这个类，并允许像已经显式声明过它那样使用它。代码清单14-1展示了这样的一个声明。

代码清单14-1 使用匿名类型的隐式局部变量

```
using System;

class Program
{
  static void Main()
  {
      var patent1 =
          new
          {
              Title = "Bifocals",
              YearOfPublication = "1784"
          };
      var patent2 =
          new
          {
              Title = "Phonograph",
              YearOfPublication = "1877"
          };
      var patent3 =
          new
          {
              patent1.Title,
              // Renamed to show property naming.
              Year = patent1.YearOfPublication
          };

      Console.WriteLine("{0} ({1})",
          patent1.Title, patent1.YearOfPublication);
      Console.WriteLine("{0} ({1})",
          patent2.Title, patent2.YearOfPublication);

      Console.WriteLine();
      Console.WriteLine(patent1);
      Console.WriteLine(patent2);

      Console.WriteLine();
      Console.WriteLine(patent3);
  }
}
```

① 事实上，在Silverlight和WinRT中，非泛型集合已被移除。

输出14-1展示了执行这个程序的结果。

输出14-1

```
Bifocals (1784)
Phonograph (1784)

{ Title = Bifocals, YearOfPublication = 1784 }
{ Title = Phonograph, YearOfPublication = 1877 }

{ Title = Bifocals, Year = 1784 }
```

匿名类型纯粹是一个C#语言特性，不是“运行时”中的一种新类型。当编译器遇到匿名类型语法的时侯，会自动生成一个CIL类，其属性对应于在匿名类型声明中命名的值和数据类型。

14.1.2 隐式类型的局部变量

由于根据定义，匿名类型是没有名称的，所以不可能将局部变量显式地声明为匿名类型。相反，局部变量的类型要替换成var。然而，并不是说隐式类型的变量就没有类型了。相反，它们的类型就是赋给它们的值的数据类型。如果将匿名类型赋给隐式类型的变量，那么在为局部变量生成的CIL代码中，它的数据类型就是编译器生成的类型。类似地，将一个string赋给隐式类型的变量，在最终生成的CIL中，变量的数据类型就是string。事实上，对于隐式类型的变量来说，假如赋给它的是一个非匿名的类型（比如string），那么最终生成的CIL代码和直接声明为string类型并无区别。如果声明语句是string text = "This is a test of the...";，那么它最终生成的CIL代码和隐式类型的声明相比：var text = "This is a test of the...";生成的CIL代码是完全一样。遇到一个隐式类型的变量时，编译器根据所赋的数据类型来确定该变量的数据类型。遇到一个带有初始化器的显式类型的局部变量（string s = "hello";）时（而且声明的同时已赋值），编译器首先会根据左侧明确声明的类型来确定s的类型，然后分析右侧的表达式，验证右侧的表达式是否适合赋给那个类型的变量。但是，对于隐式类型的局部变量来说，这个过程是相反的：首先分析右侧的表达式，确定它的类型，再用这个类型替换“var”（在逻辑上）。

虽然C#匿名类型没有名称，但它仍然是强类型的。例如，类型的属性是完全可以访问的。在代码清单14-1中，Console.WriteLine语句在内部调用了patent1.Title和patent2.YearOfPublication。调用不存在的成员会造成编译错误。在Visual Studio 2012 IDE中，就连“智能感知”功能都能很好地支持匿名类型。

隐式类型的变量要少用。匿名类型是不可能指定数据类型的，所以必须使用var。但是，在数据类型并非匿名类型的前提下，首选的做法还是使用显式数据类型。作为一种常规的情况，我们应该在关注于使代码的语义更容易理解的同时，使用编译器验证变量是你希望的类型。为达到这一目的，应该只有在赋给变量的类型是显而易见的前提下，才使用隐式类型的局部变量。例如在语句var items = new Dictionary<string, List<Account>>();中，可以清楚地看出赋给items变量的数据是什么类型，所以像这样编码显得简洁和易读。相反，假如类型不是那么容易看出，例如将方法返回值赋给变量，最好还是使用显式类型的变量：

```
Dictionary<string, List<Account>> dictionary = GetAccounts();
```

> **注意**
>
> 除非赋给变量的类型能一眼看出，否则应该只有在声明匿名类型（具体类型只有在编译时才能确定）的时候，才使用隐式类型的变量。不要不分青红皂白地使用隐式类型的变量。

> **语言对比：C++/Visual Basic/JavaScript——void*、Variant和var**
>
> 必须认识到的是，隐式类型的变量并不等于C++的void*、Visual Basic的Variant或者JavaScript的var。在上述每一种情况下，变量的用声明都不是非常严格的，因为可以将一个不同的类型重新赋给这些变量，这类似于在C#中将变量声明为object类型。相反，C#的var会由编译器确定具体类型。另外，变量一经声明，类型就不能改变，而且类型检查和成员调用会在编译时进行验证。

14.1.3 匿名类型和隐式局部变量的更多注意事项

在代码清单14-1中，匿名类型的成员名称是通过patent1和patent2中的赋值来显式确定的。例如：

```
Title = "Phonograph"
```

但是，假如所赋的值是属性或者字段调用，名称就无需指定。相反，名称将默认为字段或属性的名称。例如，在定义patent3时，我们使用了属性名称“Title”，而不是赋值给一个显式指定的名称。如输出14-1所示，最终获得的属性名称是由编译器决定的，这些名称与从中获取值的属性是匹配的。

patent1和patent2使用了相同的属性名称和相同的数据类型。所以，C#编译器只为这两个匿名类型声明生成一个数据类型。但是，patent3迫使编译器创建第二个匿名类型，因为用于存储专利年份的属性名称（Year）有别于patent1和patent2使用的名称（YearOfPublication）。除此之外，假如patent1和patent2中的两个属性的顺序发生了交换，那么这两个匿名类型就不再是“类型兼容”的。换言之，两个匿名类型要在同一个程序集中做到类型兼容[①]，属性名称、数据类型和属性顺序都必须完全匹配。只要满足这些条件，类型就是兼容的——即使它们出现在不同的方法或类中。代码清单14-2演示了类型不兼容的情况。

代码清单14-2 类型安全性和匿名类型的“不可变”特性

```
class Program
{
  static void Main()
  {
      var patent1 =
          new
          {
              Title = "Bifocals",
              YearOfPublication = "1784"
```

① 换言之，就是最终只生成一个CIL类型。——译者注

```
        };

    var patent2 =
        new
        {
            YearOfPublication = "1877",
            Title = "Phonograph"
        };

    var patent3 =
        new
        {
            patent1.Title,
            Year = patent1.YearOfPublication
        };

    // ERROR: Cannot implicitly convert type
    //        'AnonymousType#1' to 'AnonymousType#2'
    patent1 = patent2;
    // ERROR: Cannot implicitly convert type
    //        'AnonymousType#3' to 'AnonymousType#2'
    patent1 = patent3;

    // ERROR: Property or indexer 'AnonymousType#1.Title'
    //        cannot be assigned to -- it is read only'
    patent1.Title = "Swiss Cheese";
  }
}
```

前两个编译错误证明类型不兼容，所以无法从一个转换成另一个。

第三个错误是由于重新对Title属性进行赋值造成的。匿名类型是不可变的（immutable），所以匿名类型一经实例化，再更改它的某个属性，就会造成编译错误。

虽然代码清单14-2没有显示，但请记住在声明方法时，不可能将它的某个参数声明为隐式数据类型（var）。所以，在创建匿名类型的那个方法的内部，只能以两种方式将匿名类型的实例传到方法外部。首先，如果方法的参数是object类型，则匿名类型的实例可以传到方法外部，因为匿名类型将隐式转换。第二种方式是使用方法类型推导，在这种情况下，匿名类型的实例以一个方法的“类型参数”的形式来传递，编译器能成功推导出具体的类型。所以，使用Function(patent1)调用void Method<T>(T parameter)会成功地通过编译，尽管在Function()内部，parameter允许的操作仅限于object支持的那些操作。

虽然C#支持像代码清单14-1那样的匿名类型，但一般不建议像那样定义它们。匿名类型与C# 3.0支持的“投射”（projections）功能相配合，能提供许多重要的功能，比如本章后面要讨论的集合联接/关联操作。但是，除非真的需要（比如聚合来自多个类型的数据），否则一般不要定义匿名类型。

高级主题：匿名类型的生成

虽然Console.WriteLine()的实现是调用ToString()，但注意在代码清单14-1中，Console.WriteLine()的输出并不是默认的ToString()（默认的ToString()输出的是完全限定的数据类型名称）。相反，现在输出的是一个*PropertyName = value*对的列表，匿名类型的每个属性都有一对这样的输出。之所以会得到这样的结果，是因为编译器在生

成匿名类型的代码时，重写了ToString()方法，像输出14-1展示的那样对输出进行了格式化。类似地，在生成的类型中还重写了Equals()和GetHashCode()的实现。

正是由于ToString()会被重写，所以一旦属性的顺序发生了变化，就会造成最终生成一个全新的数据类型。假如不是这样设计，而是为属性顺序不同的两个匿名类型（它们可能在完全不同的类型中，甚至可能在完全不同的命名空间中）生成同一个CIL类型，那么一个实现在属性顺序上发生的变化，就会对另一个实现的ToString()输出造成显著的、甚至可能是令人无法接受的影响。除此之外，在程序执行的时候，有可能反射一个类型，并检查类型的成员——甚至动态调用其中一个成员（所谓动态调用成员，是指在程序运行期间，根据具体情况来决定调用哪个成员）。如果两个貌似相同的类型在成员顺序上有所区别，就可能造成出乎预料的结果。为了避免这些问题，C#的设计者决定：假如属性的顺序不同，就生成两个不同的类型。

14.2 集合初始化器

C# 3.0新增的另一个特性是**集合初始化器**（collection initializers）[①]。使用集合初始化器，程序员可以采取和数组声明相似的方式，在集合的实例化期间用一组初始成员来构造这个集合。如果没有集合初始化器，就只有在集合实例化好之后将成员显式地添加到集合中，如使用像System.Collections.Generic.ICollection<T>的Add()方法。使用集合初始化器，Add()调用将由C#编译器生成，而不必由开发人员显式编码。代码清单14-3展示了如何用集合初始化器初始化集合。

代码清单14-3 使用集合初始化器

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
      List<string> sevenWorldBlunders;
      sevenWorldBlunders = new List<string>()
      {
          // Quotes from Ghandi
          "Wealth without work",
          "Pleasure without conscience",
          "Knowledge without character",
          "Commerce without morality",
          "Science without humanity",
          "Worship without sacrifice",
          "Politics without principle"
      };

      Print(sevenWorldBlunders);

  }

  private static void Print<T>(IEnumerable<T> items)
  {
```

① 也可翻译为“集合初始化列表”。——译者注

```
        foreach (T item in items)
        {
            Console.WriteLine(item);
        }
    }
}
```

这个语法不仅和数组初始化的语法相似，也和对象初始化器的语法相似，都是在构造器调用的后面添加一对大括号，然后在大括号内添加一个初始化列表。如果调用构造器时没有传递参数，那么数据类型名称之后的圆括号是可要可不要的（这一点与对象初始化器相同）。

集合初始化器要想成功编译，需满足几个基本条件。理想情况下，集合类型应该实现了System.Collections.Generic.ICollection<T>接口。这样就可确保集合包含一个Add()方法，以便由编译器生成的代码可以调用这个Add()方法。但除此之外，这个需求还有一个较宽松的版本，它只要求在实现了IEnumerable<T>的类型上包含一个或多个Add方法，即使该集合类型没有实现ICollection<T>。Add()方法需要接受与集合初始化器中指定的值兼容的参数。

之所以有必要允许为不支持ICollection<T>的集合使用初始化器，是出于两方面的原因。首先，大多数集合（实现了IEnumerable<T>的类型）都没有同时实现ICollection<T>，这使集合初始化器变得用处不大。

其次，由于要匹配方法名称，而且方法的签名要和集合的初始化项兼容，所以可以在集合中初始化更加多样化的数据项。例如，初始化器现在支持new DataStore(){ a, {b, c}}——只要一个Add()方法的签名兼容于a，另一个Add()方法兼容于b，c。

注意，匿名类型是不能使用集合初始化器的，因为集合初始化器要求执行一次构造器调用，但你根本没办法命名这个构造器。解决方案是定义一个像下面这样的方法：

```
static List<T> CreateList<T>(T t) { return new List<T>(); }
```

由于能进行方法类型推导，因此类型参数可以是隐式的，而不用显式指定。所以，采用这个方案，就可以成功创建一个匿名类型的集合。

为了初始化匿名类型的集合，另一个方案是使用数组初始化器。由于不可能在构造器中指定数据类型，所以允许使用new[]，从而为匿名数组初始化器使用数组初始化语法，如代码清单14-4所示。

代码清单14-4 初始化匿名类型数组

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
  static void Main()
  {
      var worldCup2006Finalists = new[]
      {
          new
          {
              TeamName = "France",
              Players = new string[]
              {
```

```
                "Fabien Barthez", "Gregory Coupet",
                "Mickael Landreau", "Eric Abidal",
                // ...
            }
        },
        new
        {
            TeamName = "Italy",
            Players = new string[]
            {
                "Gianluigi Buffon", "Angelo Peruzzi",
                "Marco Amelia", "Cristian Zaccardo",
                // ...
            }
        };

        Print(worldCup2006Finalists);
    }

    private static void Print<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            Console.WriteLine(item);
        }
    }
}
```

最终得到的变量是由匿名类型的项构成的数组。由于是数组，所以每个项的类型都必须相同。

14.3 是什么使类成为集合：IEnumerable<T>

根据定义，.NET中的集合本质上是一个类，它最起码实现了IEnumerable<T>（或非泛型类型IEnumerable）。这个接口非常关键，因为要想支持对集合执行的遍历操作，最起码的要求就是实现由IEnumerable<T>规定的方法。

第3章讲述了如何用foreach语句来遍历由多个元素构成的数组。foreach的语法很简单，而且我们不用事先知道具体有多少个元素。但是，“运行时”不直接支持foreach语句。相反，正如本节要描述的那样，C#编译器会对代码进行必要的转换。

14.3.1 foreach和数组

代码清单14-5演示了一个简单的foreach循环，它遍历一个整数数组，将每个整数都打印到控制台。

代码清单14-5 对数组执行foreach操作

```
int[] array = new int[]{1, 2, 3, 4, 5, 6};

foreach (int item in array)
{
    Console.WriteLine(item);
}
```

基于这段代码，C#编译器会用CIL来创建一个等价的for循环，如代码清单14-6所示。

代码清单14-6 编译器如何实现对数组的foreach操作

```
int number;
int[] tempArray;
int[] array = new int[]{1, 2, 3, 4, 5, 6};

tempArray = array;
for (int counter = 0; (counter < tempArray.Length); counter++)
{
  int item = tempArray[counter];

  Console.WriteLine(item);
}
```

在这个例子中，注意foreach要依赖于对Length属性和数组索引操作符（[]）的支持。知道Length属性的值之后，C#编译器才可以用for语句遍历数组中的每一个元素。

14.3.2 foreach和IEnumerable<T>

代码清单14-6的代码对数组来说没有任何问题，因为数组长度固定，而且肯定支持索引操作符（[]）。然而，并不是所有类型的集合都包含已知数量的元素。除此之外，许多集合类，包括Stack<T>、Queue<T>以及Dictionary<Tkey, Tvalue>，都不支持按照索引获取元素。因此，需要一种更常规的方式遍历元素集合。迭代器（iterator）模式提供了这个能力。只要能确定第一个元素、下一个元素和最后一个元素，就不需要事先知道元素总数，也不需要按照索引获取元素。

System.Collections.Generic.IEnumerator<T>和非泛型System.Collections.IEnumerator接口（参见代码清单14-8）的设计目标就是允许使用迭代器模式来遍历元素集合，而不是使用如代码清单14-6所示的长度—索引（length-index）模式。图14-1展示了它们的类关系图。

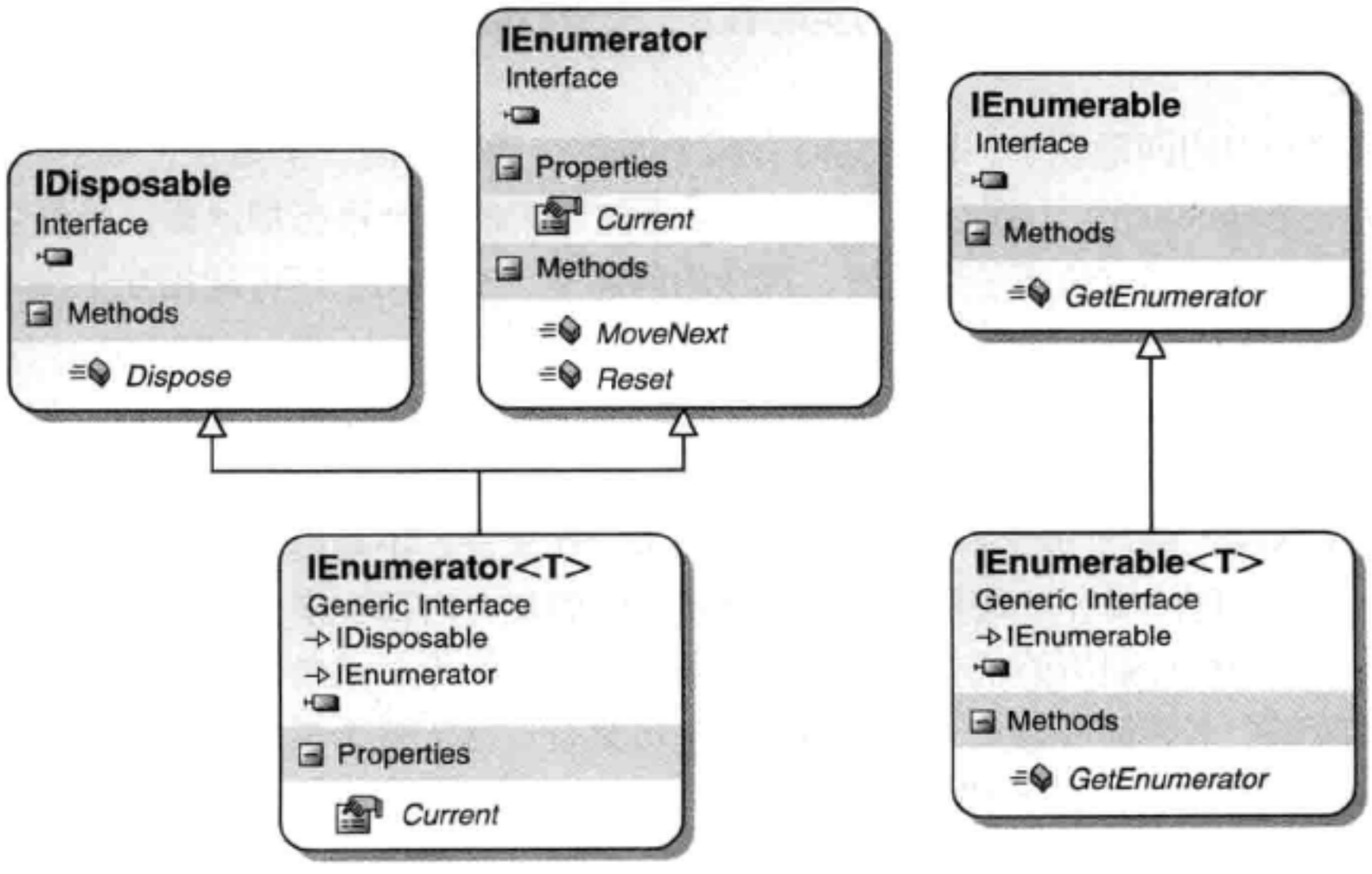


图14-1 IEnumerator<T>和IEnumerator接口

IEnumerator<T>从IEnumerator派生，后者包含3个成员。第一个成员是bool MoveNext()。利用这个方法，可以从集合中的一个元素移动到下一个元素，同时检测是否已枚举完集合中的每个元素。第二个成员是只读属性Current，用于返回当前元素。Current在IEnumerator<T>中进行了重载，提供了类型特有的实现。利用集合类的这两个成员，只需用一个while循环即可遍历集合，如代码清单14-7所示（Reset()方法一般抛出NotImplemented Exception异常，所以永远都不应调用它。如果要重新开始枚举，只需创建一个新的枚举数[①]）。

代码清单14-7 使用while遍历集合

```
System.Collections.Generic.Stack<int> stack =
  new System.Collections.Generic.Stack<int>();
int number;
// ...

// This code is conceptual, not the actual code.
while (stack.MoveNext())
{
  number = stack.Current;
  Console.WriteLine(number);
}
```

在代码清单14-7中，MoveNext()方法在越过集合末尾之后将返回false，从而避免了在循环时统计元素数目的必要。

代码清单14-7使用的集合类型是System.Collections.Generic.Stack<T>。还有其他许多集合类型。Stack<T>的关键特点是它被设计成一个“后入先出”（last in, first out, LIFO）的集合。注意，类型参数T标识了集合内的所有项的类型。泛型集合的一个关键特征就是将一种特定类型的对象全都收集到一个集合中。程序员在添加、删除或访问集合中的项时，必须理解这些项的数据类型是什么。

前面这个例子只是演示了C#编译器生成的代码的要点，但它和真实的CIL代码并不完全一致，因为其中忽略掉了两个重要的实现细节：交错和错误处理。

1. 状态共享

代码清单14-7的问题在于，假如同时有两个循环交错遍历同一个集合（一个foreach中嵌套了另一个foreach），那么集合必须维持当前元素的一个状态指示器，确保当调用MoveNext()时，能正确定位下一个元素。现在的问题是，交错的循环可能相互干扰（假如多个线程都执行循环会发生同样的问题）。

为了解决这个问题，集合类不直接支持IEnumerator<T>和IEnumerator接口。如图14-1所示，还有另一个接口，名为IEnumerable<T>，它唯一的方法就是GetEnumerator()。这个方法的作用是返回支持IEnumerator<T>的一个对象。在这里，不是由集合类来维持状态。相反，是由一个不同的类（通常是嵌套类，以便访问到集合内部）来支持IEnumerator<T>接口，并负责维护循环遍历的状态。枚举数相当于一个“游标”或者“书签”。可以有多个书签，移动每个书签都可独立于其他书签来遍历集合。基于这个模式，代码清单14-8展示了与foreach循环等价的C#代码。

① 根据MSDN文档，enumerator被翻译为“枚举数”。枚举数在最开始的时候定位在集合第一个元素前。——译者注

代码清单14-8 在循环遍历期间，由一个单独的枚举数来维持状态

```
System.Collections.Generic.Stack<int> stack =
  new System.Collections.Generic.Stack<int>();
int number;
System.Collections.Generic.Stack<int>.Enumerator
  enumerator;

// ...

// If IEnumerable<T> is implemented explicitly,
// then a cast is required.
// ((IEnumerable<int>)stack).GetEnumerator();
enumerator = stack.GetEnumerator();
while (enumerator.MoveNext())
{
  number = enumerator.Current;
  Console.WriteLine(number);
}
```

2. 清理状态

由于是由实现了IEnumerator<T>接口的类来维持状态，所以在退出循环之后，有时需要对状态进行清理（之所以退出循环，要么是由于所有循环迭代都已经结束了，要么是中途引发了一个异常）。为此，IEnumerator<T>接口派生于IDisposable。实现IEnumerator的枚举数不一定要实现IDisposable。但是，假如实现了IDisposable，就一样会调用Dispose()方法。这样一来，就可以在foreach循环退出之后调用Dispose()。与最终的CIL代码等价的C#代码如代码清单14-9所示。

代码清单14-9 对集合执行foreach的编译结果

```
System.Collections.Generic.Stack<int> stack =
  new System.Collections.Generic.Stack<int>();
System.Collections.Generic.Stack<int>.Enumerator
  enumerator;
IDisposable disposable;

enumerator = stack.GetEnumerator();
try
{
  int number;
  while (enumerator.MoveNext())
  {
    number = enumerator.Current;
    Console.WriteLine(number);
  }
}
finally
{
  // Explicit cast used for IEnumerator<T>.
  disposable = (IDisposable) enumerator;
  disposable.Dispose();

  // IEnumerator will use the as operator unless IDisposable
  // support is known at compile time.
  // disposable = (enumerator as IDisposable);
```

```
    // if (disposable != null)
    // {
    //     disposable.Dispose();
    // }
}
```

注意，由于IEnumerator<T>支持IDisposable接口，所以可以使用using关键字来简化代码清单14-9中的C#代码，如代码清单14-10所示。

代码清单14-10 使用using的错误处理和资源清理

```
System.Collections.Generic.Stack<int> stack =
  new System.Collections.Generic.Stack<int>();
int number;

using(
  System.Collections.Generic.Stack<int>.Enumerator
      enumerator = stack.GetEnumerator())
{
  while (enumerator.MoveNext())
  {
      number = enumerator.Current;
      Console.WriteLine(number);
  }
}
```

但要记住，using关键字也不是直接由CIL支持的，所以实际上，代码清单14-9的C#代码才更准确地对应于foreach的CIL代码。

> **高级主题：没有IEnumerable的foreach**
>
> C#编译器不要求一定要实现IEnumerable/IEnumerable<T>才能用foreach对一个数据类型进行迭代。相反，编译器采用一个称为“Duck typing”的概念[①]；也就是查找一个GetEnumerator()方法，这个方法返回包含Current属性和MoveNext()方法的一个类型。Duck typing按名称查找方法，而不是依赖接口或显式方法调用。如果找不到可枚举模式的恰当实现，编译器就检查集合是否实现了接口。

14.3.3 foreach循环内不要修改集合

第3章讲过，编译器禁止对foreach变量标识符（number）[②]进行赋值。如代码清单14-10所示，对number进行赋值不会改变集合元素本身。所以，为了避免产生混淆，C#编译器干脆禁止了此类赋值。

除此之外，在foreach循环执行期间，集合中的元素计数是不能变动的，集合项本身一般也不能修改。例如，假定在foreach循环中调用stack.Push(42)，那么迭代器应该忽略还是集成对stack的更改？或者说，迭代器应该遍历新添加的项，还是应该忽略这个项，并假

---

① Duck typing来源于一句英国俗语：If it walks like a duck and quacks like a duck, it must be a duck。（如果它走起路来像一只鸭子，叫起来像一只鸭子，那么肯定是一只鸭子。）对于C#这样的动态语言，它的意思是说，可将一个对象传给正在期待一个特定类型的方法，即使它并非继承自该类型。它唯一要做的就是支持该方法想要使用的、由它原来期待的类型定义的方法和属性。就当前的情况来说，是指一个对象要想被当成是一只鸭子，只需实现Quack()方法，不需要实现IDuck接口。——译者注

② 以foreach (char cell in cells)为例，foreach变量标识符是cell。在代码清单14-10中，这个标识符是number。——译者注

定状态没变（仍为刚刚实例化时的状态）？

正是因为存在上述歧义，所以假如在foreach循环期间对集合进行修改，重新访问枚举数就会引发System.InvalidOperationException异常，指出在枚举数实例化之后，集合已经发生了变化。

14.4 标准查询操作符

如果将System.Object的方法排除在外，那么实现IEnumerable<T>的任何类型就都只需实现GetEnumerator()这一个方法。但看问题不能只看到表面。事实上，任何类型在实现IEnumerable<T>之后，都有超过50个方法可供使用，其中还不包括重载的版本。而为了享受到所有这一切，除了GetEnumerator()之外，根本不需要显式地实现其他任何方法。附加的功能是由C# 3.0开始引入的扩展方法来提供的，所有方法都在System.Linq.Enumerable类中定义。所以，为了用到这些方法，只需简单地添加以下语句：

```
using System.Linq;
```

IEnumerable<T>上的每个方法都是一个**标准查询操作符**（standard query operator），用于为所操作的集合提供查询功能。在后续小节中，我们将探讨一些最重要的标准查询操作符。

许多例子都要依赖Inventor和/或Patent类，两个类的定义如代码清单14-11所示。

代码清单14-11 使用标准查询操作符的示例类

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Patent
{
  // Title of the published application
  public string Title { get; set; }

  // The date the application was officially published
  public string YearOfPublication { get; set; }

  // A unique number assigned to published applications
  public string ApplicationNumber { get; set; }

  public long[] InventorIds { get; set; }

  public override string ToString()
  {
      return string.Format("{0}({1})",
          Title, YearOfPublication);
  }
}

public class Inventor
{
  public long Id { get; set; }
  public string Name { get; set; }
```

```
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public override string ToString()
    {
        return string.Format("{0}({1}, {2})",
            Name, City, State);
    }
}

class Program
{
  static void Main()
  {
    IEnumerable<Patent> patents = PatentData.Patents;
    Print(patents);

    Console.WriteLine();

    IEnumerable<Inventor> inventors = PatentData.Inventors;
    Print(inventors);
  }

  private static void Print<T>(IEnumerable<T> items)
  {
      foreach (T item in items)
      {
          Console.WriteLine(item);
      }
  }
}

public static class PatentData
{
  public static readonly Inventor[] Inventors = new Inventor[]
      {
          new Inventor(){
              Name="Benjamin Franklin", City="Philadelphia",
              State="PA", Country="USA", Id=1 },
          new Inventor(){
              Name="Orville Wright", City="Kitty Hawk",
              State="NC", Country="USA", Id=2},
          new Inventor(){
              Name="Wilbur Wright", City="Kitty Hawk",
              State="NC", Country="USA", Id=3},
          new Inventor(){
              Name="Samuel Morse", City="New York",
              State="NY", Country="USA", Id=4},
          new Inventor(){
              Name="George Stephenson", City="Wylam",
              State="Northumberland", Country="UK", Id=5},
          new Inventor(){
              Name="John Michaelis", City="Chicago",
              State="IL", Country="USA", Id=6},
```

```
        new Inventor(){
            Name="Mary Phelps Jacob", City="New York",
            State="NY", Country="USA", Id=7},
    };

public static readonly Patent[] Patents = new Patent[]
    {
        new Patent(){
            Title="Bifocals", YearOfPublication="1784",
            InventorIds=new long[] {1}},
        new Patent(){
            Title="Phonograph", YearOfPublication="1877",
            InventorIds=new long[] {1}},
        new Patent(){
            Title="Kinetoscope", YearOfPublication="1888",
            InventorIds=new long[] {1}},
        new Patent(){
            Title="Electrical Telegraph",
            YearOfPublication="1837",
            InventorIds=new long[] {4}},
        new Patent(){
            Title="Flying machine", YearOfPublication="1903",
            InventorIds=new long[] {2,3}},
        new Patent(){
            Title="Steam Locomotive",
            YearOfPublication="1815",
            InventorIds=new long[] {5}},
            new Patent(){
                Title="Droplet deposition apparatus",
                YearOfPublication="1989",
                InventorIds=new long[] {6}},
            new Patent(){
                Title="Backless Brassiere",
                YearOfPublication="1914",
                InventorIds=new long[] {7}},
        };
    }
```

代码清单14-11提供了一些示例数据。输出14-2展示了结果。

输出14-2

```
Bifocals(1784)
Phonograph(1877)
Kinetoscope(1888)
Electrical Telegraph(1837)
Flying machine(1903)
Steam Locomotive(1815)
Droplet deposition apparatus(1989)
Backless Brassiere(1914)

Benjamin Franklin(Philadelphia, PA)
Orville Wright(Kitty Hawk, NC)
Wilbur Wright(Kitty Hawk, NC)
Samuel Morse(New York, NY)
George Stephenson(Wylam, Northumberland)
John Michaelis(Chicago, IL)
Mary Phelps Jacob(New York, NY)
```

14.4.1 使用Where()来筛选

为了从集合中筛选出数据，需要提供筛选器方法来返回true或false以指明特定的元素是否应该被包含进来。获取一个实参并返回一个布尔值的委托表达式称为**谓词**[①]。集合的Where()方法依据谓词来确定筛选条件。代码清单14-12展示了一个例子。（从技术上说，Where()方法的结果是一个monad[②]，它封装了根据一个给定的谓词对一个给定的序列进行筛选的操作。）输出14-3展示了结果。

代码清单14-12 使用System.Linq.Enumerable.Where()来筛选

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
  static void Main()
  {
      IEnumerable<Patent> patents = PatentData.Patents;
      patents = patents.Where(
          patent => patent.YearOfPublication.StartsWith("18"));
      Print(patents);
  }

  // ...
}
```

输出14-3

```
Phonograph(1877)
Kinetoscope(1888)
Electrical Telegraph(1837)
Steam Locomotive(1815)
```

注意，代码将Where()的输出结果赋回给了IEnumerable<T>。换言之，IEnumerable<T>.Where()输出的是一个新的IEnumerable<T>集合。具体到代码清单14-12，就是IEnumerable<Patent>。

许多人不知道的是，Where()方法的表达式实参并非一定在赋值时求值。这一点适合许多标准查询操作符。例如，在Where()的情况下，表达式被传给集合，“保存”起来但不马上执行。相反，只有在需要遍历集合中的项时，才会真正对表达式进行求值。例如，foreach循环（例如代码清单14-11的Print()方法中的那一个）会触发表达式针对集合中的每一项进行求值。至少从概念上说，应该认为Where()方法只是描述了集合中应该出现什么，而没有描述具体应该如何遍历数据项并生成一个可能包含更少数据项的新集合。

---

① predicate在.NET Framework SDK文档中翻译成“谓词”。——译者注

② monad一词来源于希腊哲学概念，大致相当于“unit”。在函数式编程中，一个monad代表的是一种用于表示计算（而不是在域模型中表示数据）的抽象数据类型。monad允许程序员将不同的行动（操作）链接到一起来构成一个管道。其中每个行动都用由monad提供的附加处理规则进行修饰。在以函数式风格写的程序中，可利用monad来结构化含有顺序操作的过程，或者定义任意控制流（比如处理并发操作、延续性操作或者异常）。——译者注

缺页

代码清单14-15 投射到匿名类型

```
        // ...
        IEnumerable<string> fileList = Directory.GetFiles(
            rootDirectory, searchPattern);
        var items = fileList.Select(
            file =>
            {
                FileInfo fileInfo = new FileInfo(file);
                return new
                {
                    FileName = fileInfo.Name,
                    Size = fileInfo.Length
                };
            });
        // ...
```

输出14-4

```
{ FileName = AssemblyInfo.cs, Size = 1704 }
{ FileName = CodeAnalysisRules.xml, Size = 735 }
{ FileName = CustomDictionary.xml, Size = 199 }
{ FileName = EssentialCSharp.sln, Size = 40415 }
{ FileName = EssentialCSharp.suo, Size = 454656 }
{ FileName = EssentialCSharp.vsmdi, Size = 499 }
{ FileName = EssentialCSharp.vssscc, Size = 256 }
{ FileName = intelliTechture.ConsoleTester.dll, Size = 24576 }
{ FileName = intelliTechture.ConsoleTester.pdb, Size = 30208 }
{ FileName = LocalTestRun.testrunconfig, Size = 1388 }
```

在为匿名类型生成的ToString()方法中，会自动添加用于显示属性名称及其值的代码。

使用select()进行“投射”，这是非常强大的一个功能。上一节讲述了如何使用Where()标准查询操作符在“垂直”方向上筛选集合[①]（减少集合中项的数量）。现在，使用Select()标准查询操作符，还可以在“水平”方向上减小集合的规模（减少列的数量）或者对数据进行彻底的转换。综合运用Where()和Select()，可以获得原始集合的一个子集，从而满足当前算法的要求。这两个方法各自提供了一个功能强大的、对集合进行操纵的API。以前要获得相同的结果，必须手动写大量难以阅读的代码。

高级主题：LINQ查询的并行运行

随着越来越多的计算机配备了多个处理器，而且每个处理器都有多个内核，人们迫切希望能简单地利用这些额外的处理能力。为此，需要修改程序来支持多线程，使工作可以在计算机的不同CPU上同时进行。代码清单14-16演示了使用并行LINQ（PLING）来达到这个目标的一个途径。

代码清单14-16 并行执行LINQ查询

```
        // ...
        IEnumerable<string> fileList = Directory.GetFiles(
            rootDirectory, searchPattern);
```

① 将集合想象成一个表格就好理解了。在表格的垂直方向上，Where()可以减少集合中的项的数量。——译者注

```
var items = fileList.AsParallel().Select(
    file =>
    {
        FileInfo fileInfo = new FileInfo(file);
        return new
        {
            FileName = fileInfo.Name,
            Size = fileInfo.Length
        };
    });
// ...
```

如代码清单14-16所示，简单修改一下代码就可以支持并行。唯一要做的就是利用.NET Framework 4引入的标准查询操作符AsParallel()，它是静态类System.Linq.ParallelEnumerable的成员。使用这个简单的扩展方法，“运行时”一边遍历fileList中的数据项，一边返回结果对象，这两个操作是并行发生的。在这种情况下进行的每个并行操作的代价并不是特别高（虽然只是相对于正在进行的其他操作而言），但在执行CPU密集型处理的时候（比如在进行加密或压缩的时候），累积起来的代价还是相当高的。在多个CPU之间并行执行，执行时间将根据CPU的数量成比例地缩短。

但有一点非常重要（这也是为什么将AsParallel()放在“高级主题”而不是正文中介绍的原因），并行执行可能引入竞态条件（race condition）。也就是说，一个线程上的一个操作可能会与另一个不同的线程上的一个操作混合，造成数据被破坏。为了避免这个问题，需要向多个线程共享访问的数据应用同步机制，在必要时强迫操作的原子性。然而，同步本身可能引入死锁，造成执行被“冻结”，使并行编程变得更复杂。

要更多地了解这个问题以及其他多线程主题，请参见第18章和第19章。

14.4.3 使用Count()对元素进行计数

对数据项集合执行的另一个常见的操作是获取计数。为了支持这一操作，LINQ提供了Count()扩展方法。

代码清单14-17演示了如何使用重载的Count()来统计所有元素的数量（这个版本是无参数的），或者获取一个谓词作为参数，只对谓词表达式指明的数据项进行计数。

代码清单14-17 用Count()对元素进行计数

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
  static void Main()
  {
      IEnumerable<Patent> patents = PatentData.Patents;
      Console.WriteLine("Patent Count: {0}", patents.Count());
      Console.WriteLine("Patent Count in 1800s: {0}",
          patents.Count(patent =>
              patent.YearOfPublication.StartsWith("18")));
```

```
        }

        // ...
    }
```

虽然Count()语句写起来简单，但IEnumerable<T>没有改变，所以真正执行的代码仍然会遍历集合中的所有项。如果集合直接提供一个Count属性，就应该首选属性，而不要用LINQ的Count()方法（这是一个许多人都没有意识到的差异）。幸好，ICollection<T>包含了Count属性，所以如果集合支持ICollection<T>，那么在它上面调用Count()方法，会对集合进行转型，并直接调用Count。然而，如果不支持ICollection<T>，Enumerable.Count()就会枚举集合中的所有项，而不是调用内建的Count机制。如果计数的目的只是为了看这个计数是否大于0（if(patents.Count() > 0){...}），那么首选的做法是使用Any()操作符（if(patents.Any()) {...}）。Any()只尝试遍历集合中的一个项，成功就返回true，而不会遍历整个序列。

规范

要在检查集合中是否有项的时候使用System.Linq.Enumerable.Any()而不是调用Count()方法。

要使用集合的Count属性（如果有的话），而不是调用System.Linq.Enumerable.Count()方法。

14.4.4 推迟执行

使用LINQ时，要记住的一个重要概念就是推迟执行。来看看代码清单14-18和输出14-5。

代码清单14-18 使用System.Linq.Enumerable.Where()来筛选

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

        IEnumerable<Patent> patents = PatentData.Patents;
bool result;
patents = patents.Where(
    patent =>
    {
        if (result =
            patent.YearOfPublication.StartsWith("18"))
        {
            // Side effects like this in a predicate
            // are used here to demonstrate a
            // principle and should generally be
            // avoided.
            Console.WriteLine("\t" + patent);
```

```
        }
        return result;
    });

Console.WriteLine("1. Patents prior to the 1900s are:");
foreach (Patent patent in patents)
{
}

Console.WriteLine();
Console.WriteLine(
    "2. A second listing of patents prior to the 1900s:");
Console.WriteLine(
    "   There are {0} patents prior to 1900.",
    patents.Count());

Console.WriteLine();
Console.WriteLine(
    "3. A third listing of patents prior to the 1900s:");
patents = patents.ToArray();
Console.Write("   There are ");
Console.WriteLine("{0} patents prior to 1900.",
    patents.Count());

// ...
```

输出14-5

```
1. Patents prior to the 1900s are:
        Phonograph(1877)
        Kinetoscope(1888)
        Electrical Telegraph(1837)
        Steam Locomotive(1815)

2. A second listing of patents prior to the 1900s:
        Phonograph(1877)
        Kinetoscope(1888)
        Electrical Telegraph(1837)
        Steam Locomotive(1815)
   There are 4 patents prior to 1900.

3. A third listing of patents prior to the 1900s:
        Phonograph(1877)
        Kinetoscope(1888)
        Electrical Telegraph(1837)
        Steam Locomotive(1815)
There are 4 patents prior to 1900.
```

注意，Console.WriteLine("1. Patents prior…)是先于Lambda表达式执行的。这是非常重要的一个特点，假如没有意识到它的重要性，就很容易将它忽略[①]。通常，任何谓词都只应做一件事情：对一个条件进行求值。它不应该有任何“副作用”（比如在本例中打印到控制台）。

为了理解背后发生的事情，请记住Lambda表达式是可以四处传递的委托——是方法的

① 这一行代码被故意放到了Lambda表达式的后面，但却先于Lambda表达式执行。——译者注

引用。在LINQ和标准查询操作符的上下文中，每个Lambda表达式都构成了要执行的总体查询的一部分。

Lambda表达式在声明时不执行。Lambda表达式除非被调用，否则其中的代码不会执行。图14-2展示了具体的运算顺序。

如图14-2所示，代码清单14-18中的 3 个调用都触发了Lambda表达式的执行，每次都是隐式触发的。假如Lambda表达式执行的代价比较高（比如查询数据库），那么为了优化代码，很重要的一点就是尽量减少Lambda表达式的执行。

首先，foreach循环会触发Lambda表达式的执行。本章前面说过，foreach循环被分解成一个MoveNext()调用，而且每个调用都会导致为原始集合中的每一项执行Lambda表达式。在循环迭代期间，“运行时”会为每一项调用Lambda表达式，以判断该项是否满足谓词。

其次，调用Enumerable的Count()函数，会再次为每一项触发Lambda表达式。这同样很容易被人忽视，因为在没有用标准查询运算符来查询的集合上，Count属性是非常常见的。

最后，调用ToArray()（或ToList()、ToDictionary()或ToLookup()）会为每一项触发Lambda表达式。但是，使用这些“To*XXX*”方法来转换集合是相当有用的。这样返回的是已由标准查询操作符处理过的集合。在代码清单14-18中，转换成数组意味着在最后一个Console.WriteLine()语句中调用Length时，patents指向的基础对象事实上是一个数组（它显然实现了IEnumerable<T>），所以调用的Length是由System.Array实现的，而不是由System.Linq.Enumerable实现的。因此，在用“To*XXX*”方法转换成集合类型之后，一般可以安全地操纵集合（除非调用了另一个标准查询操作符）。但要注意，这会造成将整个结果集都加载到内存（在此之前可能驻留在一个数据库或文件中）。除此之外，“To*XXX*”方法会创建基础数据的“快照”，确保在重新查询“To*XXX*”方法的结果时，不会返回新的结果。

强烈建议读者好好体会一下图14-2展示的顺序图，拿实际的代码去对照一下，理解由于标准查询操作符存在“推迟执行”的特点，所以可能在不知不觉间触发标准查询操作符。开发者应提高警惕，防止非预期的调用。查询对象代表的是查询，而不是结果。向查询要求结果时，整个查询都会执行（甚至可能是再次执行），因为查询对象不确定结果和上一次执行的结果（如果存在的话）是不是一样。

注意

为了避免这种反复性的执行，一个查询在执行之后，有必要把它获取的数据缓存起来。为此，可以使用一个“To*XXX*”方法（比如ToArray()）将数据赋给一个局部集合。将“To*XXX*”方法的返回结果赋给一个集合，显然会造成查询的执行。但在此之后，对已赋好值的集合进行遍历，就不会再涉及查询表达式。一般情况下，如果“内存中的集合快照”是你所想要的，那么最好的做法就是将查询表达式赋给一个缓存的集合，避免无谓的遍历。

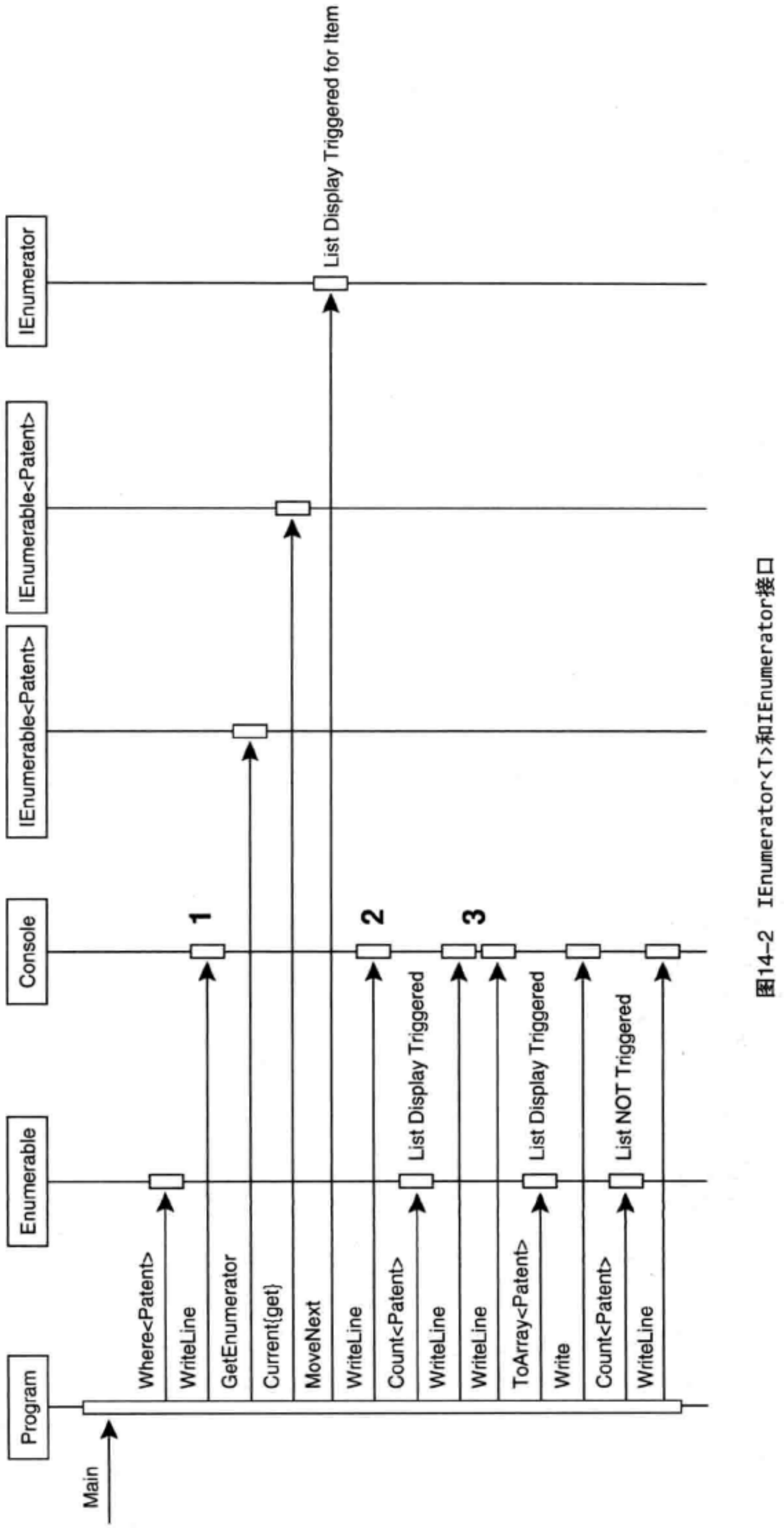


图14-2 IEnumerator<T>和IEnumerator接口

14.4.5 使用OrderBy()和ThenBy()来排序

对集合执行的另一个常见的操作就是对它进行排序。这涉及对System.Linq.Enumerable的OrderBy()的调用，如代码清单14-19和输出14-6所示。

代码清单14-19 使用System.Linq.Enumerable.OrderBy()/ThenBy()排序

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

        IEnumerable<Patent> items;
        Patent[] patents = PatentData.Patents;
        items = patents.OrderBy(
            patent => patent.YearOfPublication).ThenBy(
            patent => patent.Title);
        Print(items);
        Console.WriteLine();

        items = patents.OrderByDescending(
            patent => patent.YearOfPublication).ThenByDescending(
            patent => patent.Title);
        Print(items);

// ...
```

输出14-6

```
Bifocals  (1784)
Steam Locomotive(1815)
Electrical Telegraph(1837)
Phonograph(1877)
Kinetoscope(1888)
Flying machine  (1903)
Backless Brassiere(1914)
Droplet deposition apparatus(1989)

Droplet deposition apparatus(1989)
Backless Brassiere(1914)
Flying machine  (1903)
Kinetoscope(1888)
Phonograph(1877)
Electrical Telegraph(1837)
Steam Locomotive(1815)
Bifocals  (1784)
```

OrderBy()获取一个Lambda表达式，该表达式标识了要据此进行排序的键。在代码清单14-19中，第一次排序使用专利的发布年份（YearOfPublication）作为键。

但要注意，OrderBy()只会获取一个称为keySelector的参数来排序。要依据第二个列来排序，需要使用一个不同的方法ThenBy()。类似地，更多的排序要使用更多的ThenBy()。

OrderBy()返回的是一个IOrderedEnumerable<T>接口，而不是一个IEnumerable<T>。除此之外，IOrderedEnumerable<T>是从IEnumerable<T>派生的，所以能为OrderBy()的返回值使用全部标准查询操作符（包括OrderBy()）。但是，假如重复调用OrderBy()，会撤

销上一个OrderBy()的工作，只有最后一个OrderBy()的keySelector才真正起作用。所以，注意不要在上一个OrderBy()调用的基础上再调用OrderBy()。

为了指定额外的排序条件，应该使用ThenBy()。虽然ThenBy()是一个扩展方法，但它扩展的不是IEnumerable<T>，而是IOrderedEnumerable<T>。该方法也是在System.Linq.Extensions. Enumerable上定义的，声明如下：

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

总之，要先使用OrderBy()，再执行零个或者多个ThenBy()调用来提供额外的排序“列”。OrderByDescending()和ThenByDescending()提供了相同的功能，只是变成按降序排序。升序和降序方法混用是没有问题的，但假如要执行进一步的排序，请使用一个ThenBy()调用（无论升序还是降序）。

关于排序，还有两个重要的问题需要注意。首先，要等到开始访问集合中的成员时，才会实际开始排序，在那个时候，整个查询都会被处理。显然，除非拿到了所有需要排序的项，否则是无法排序的。如果没有得到所有的项，就无法确定自己是否已经获得了第一项。排序被推迟到开始访问成员的时候才开始，这要“归功”于本章前面讨论过的“推迟执行”。其次，执行后续的数据排序调用时（例如，先调用Orderby()，再调用ThenBy()，再调用ThenByDescending()），会再次调用之前的keySelector Lambda表达式。换言之，如果先调用OrderBy()，那么在遍历集合时，会调用对应的keySelector Lambda表达式。如果接着调用ThenBy()，会造成那个OrderBy()的keySelector被再次调用。

规范

不要为OrderBy()的结果再次调用OrderBy()。附加的排序依据用ThenBy()来指定。

初学者主题：联接（join）操作

来考虑两个对象集合，如图14-3的维恩图所示。

左边的圆包含所有发明者，右边的圆包含所有专利。交集中既有发明者也有专利。另外，凡是发明者和专利存在一个匹配，就用一条线来连接。如图14-3所示，每个发明者都可能有多项专利，而每项专利可能有一个或者多个发明者。每项专利都至少有一个发明者，但在某些情况下，一个发明者可能还没有任何专利。

在交集中将发明者与专利匹配，这称为内部联接（inner join）操作。结果是由一对一对的发明者-专利构成的集合，在每一对发明者-专利中，专利和发明者都同时存在。而左外部联接（left outer join）包含左边那个圆中的所有项，不管它们是否有一项对应的专利。在这个特定的例子中，右外部联接（right outer join）和内部联接是一样的，因为所有专利都有发明者。除此之外，谁在左边，谁在右边，这是任意指定的，所以左外联接和右外联接实际并无区别。但是，如果执行完全外部联接（full outer join），就需要同时包含来自两侧的记录，只是极少需要执行这种联接操作罢了。

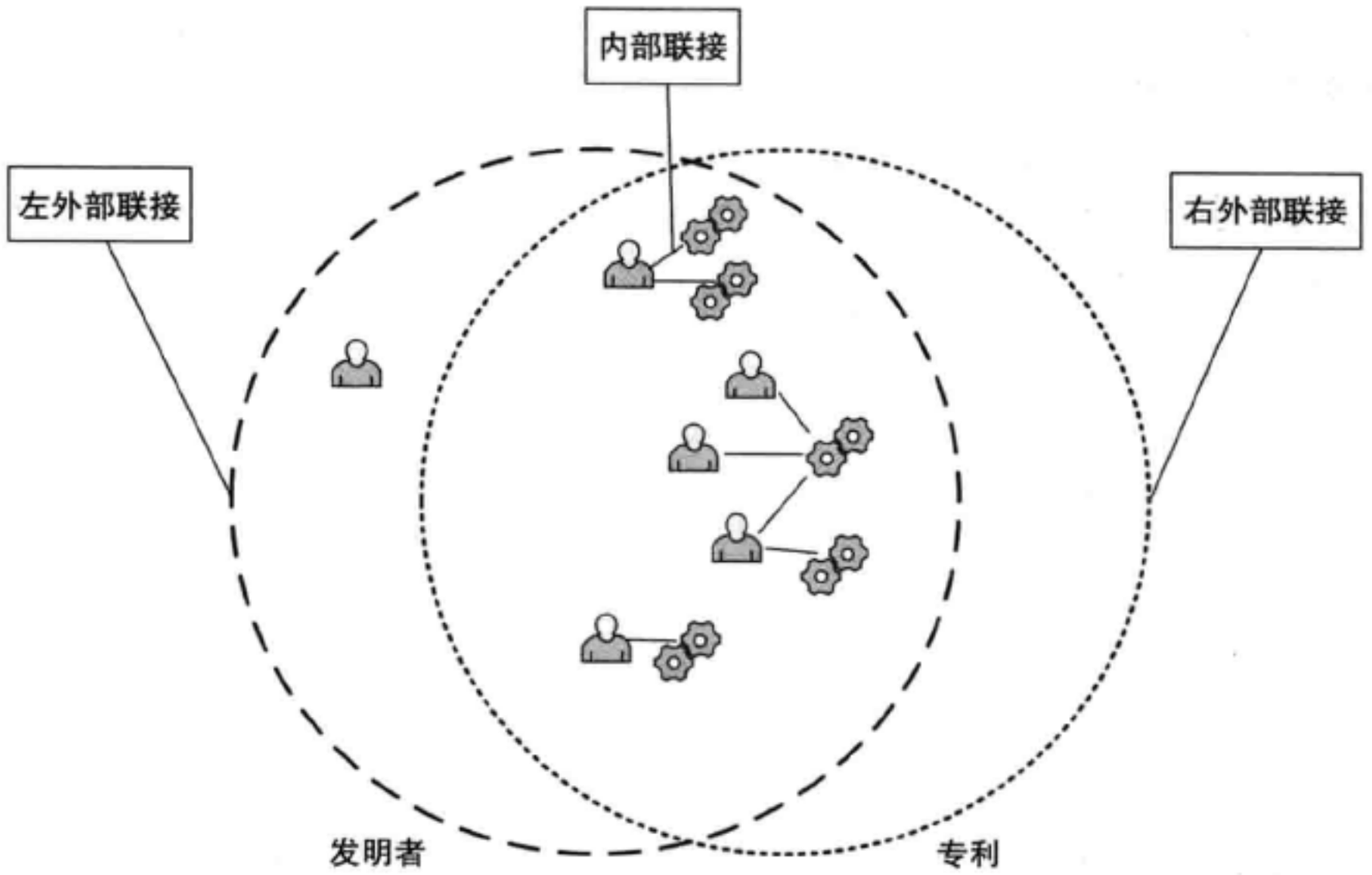


图14-3 发明者和专利集合的维恩图

对于发明者和专利的关系来说，另一个重要特征就是它是一个“多对多”的关系。每一项专利都可能有一个或多个发明者（例如飞机是由莱特兄弟发明的）。除此之外，每个发明者都可能有一项或多项专利（本杰明·富兰克林发明了双焦距眼镜和留声机）。

还有一个常见的关系是“一对多”关系。例如，公司里的一个部门可能有多名员工。然而，每个员工同时只能隶属于一个部门。（然而，在“一对多”关系中引入时间因素，就可以使之变成“多对多”关系。例如，一名特定的员工可能从一个部门调换到另一个部门，所以随着时间的推移，他可能与多个部门关联在一起，形成一种“多对多”的关系。）

在代码清单14-20中，我们对Employee和Department数据进行了演示，输出14-7展示了结果。

代码清单14-20 示例Employee和Department数据

```
public class Department
{
    public long Id { get; set; }
    public string Name { get; set; }
    public override string ToString()
    {
        return string.Format("{0}", Name);
    }
}
```

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Title { get; set; }
    public int DepartmentId { get; set; }
    public override string ToString()
    {
```

```
        return string.Format("{0} ({1})", Name, Title);
    }
}

public static class CorporateData
{
    public static readonly Department[] Departments =
        new Department[]
      {
        new Department(){
            Name="Corporate", Id=0},
        new Department(){
            Name="Finance", Id=1},
        new Department(){
            Name="Engineering", Id=2},
        new Department(){
            Name="Information Technology",
            Id=3},
        new Department(){
            Name="Research",
            Id=4},
        new Department(){
            Name="Marketing",
            Id=5},
    };

    public static readonly Employee[] Employees = new Employee[]
      {
        new Employee(){
            Name="Mark Michaelis",
            Title="Chief Computer Nerd",
            DepartmentId = 0},
        new Employee(){
            Name="Michael Stokesbary",
            Title="Senior Computer Wizard",
            DepartmentId=2},
        new Employee(){
            Name="Brian Jones",
            Title="Enterprise Integration Guru",
            DepartmentId=2},
        new Employee(){
            Name="Jewel Floch",
            Title="Bookkeeper Extraordinaire",
            DepartmentId=1},
        new Employee(){
            Name="Robert Stokesbary",
            Title="Expert Mainframe Engineer",
            DepartmentId = 3},
        new Employee(){
            Name="Paul R. Bramsman",
            Title="Programmer Extraordinaire",
            DepartmentId = 2},
        new Employee(){
            Name="Thomas Heavey",
            Title="Software Architect",
            DepartmentId = 2},
        new Employee(){
```

```
                Name="John Michaelis",
                Title="Inventor",
                DepartmentId = 4}
        };
    }
```

```
class Program
{
    static void Main()
    {
        IEnumerable<Department> departments =
            CorporateData.Departments;
        Print(departments);

        Console.WriteLine();

        IEnumerable<Employee> employees =
            CorporateData.Employees;
        Print(employees);
    }

    private static void Print<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            Console.WriteLine(item);
        }
    }
}
```

输出14-7

```
Corporate
Finance
Engineering
Information Technology
Research
Marketing

Mark Michaelis (Chief Computer Nerd)
Michael Stokesbary (Senior Computer Wizard)
Brian Jones (Enterprise Integration Guru)
Jewel Floch (Bookkeeper Extraordinaire)
Robert Stokesbary (Expert Mainframe Engineer)
Paul R. Bramsman (Programmer Extraordinaire)
Thomas Heavey (Software Architect)
John Michaelis (Inventor)
```

下一节讲述如何对数据执行联接操作时，会用到这些数据。

14.4.6 使用Join()来执行内部联接

在客户端上，对象和对象的关系一般已经建立好了。例如，文件和其所在目录的关系是由DirectoryInfo.GetFiles()方法和FileInfo.Directory方法预先建立好的。但是，从非对象存储（nonobject store）加载的数据则一般不是这种情况。相反，这些数据需要联接到一起，以便能以一种适合于数据的方式，从一种对象类型切换到下一种对象类型。

以员工和公司部门为例。在代码清单14-21中，我们将每个员工都联接到他/她的部

门，然后列出每名员工及其对应的部门。由于每个员工都只隶属于一个（而且只有一个）部门，所以列表中的数据项的总数等于员工总数——每个员工都保证只出现一次（每个员工都被正规化了）。这段代码的结果如输出14-8所示。

代码清单14-21 使用System.Linq.Enumerable.Join()进行内部联接

```
using System;
using System.Linq;

// ...

Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;

var items = employees.Join(
    departments,
    employee => employee.DepartmentId,
    department => department.Id,
    (employee, department) => new
    {
        employee.Id,
        employee.Name,
        employee.Title,
        Department = department
    });

foreach (var item in items)
{
        Console.WriteLine("{0} ({1})",
            item.Name, item.Title);
        Console.WriteLine("\t" + item.Department);
    }

// ...
```

输出14-8

```
Mark Michaelis (Chief Computer Nerd)
        Corporate
Michael Stokesbary (Senior Computer Wizard)
        Engineering
Brian Jones (Enterprise Integration Guru)
        Engineering
Jewel Floch (Bookkeeper Extraordinaire)
        Finance
Robert Stokesbary (Expert Mainframe Engineer)
        Information Technology
Paul R. Bramsman (Programmer Extraordinaire)
        Engineering
Thomas Heavey (Software Architect)
        Engineering
John Michaelis (Inventor)
        Research
```

Join()的第一个参数称为inner，指定了employees要联接到的集合，即departments。接下来的两个参数都是Lambda表达式，指定了两个集合如何联接。第一个Lambda表达式是employee => employee.DepartmentId，这个参数称为outerKeySelector，指出每个员工的键是DepartmentId。下一个Lambda表达式是department => department.Id，它将

Department的Id属性指定为键。换言之，每个员工都联接到employee.DepartmentId等于department.Id的一个部门。最后一个参数在这个例子中是一个匿名类型，它描述了如何对结果项进行选取。在本例声明的匿名类型中，我们添加了一个员工的Id、Name和Title，并用一个Department属性来存储联接的department对象。

注意，在输出结果中，Engineering多次出现，对于工程部的每一个员工都会出现一次。在本例中，Join()调用产生了所有部门和所有员工的一个**笛卡儿积**，换言之，假如一条记录在两个集合中都能找到，而且指定的部门ID相同，就创建一条新记录。这种类型的联接是内部联接。

也可以反向联接数据，即将部门联接到每个员工，从而列出所有“部门—员工”匹配。注意输出的记录数要多于部门数，因为每个部门都可能包含多个员工，而每个匹配的情形都要输出一条记录。和前面一样，Engineering部门会多次显示，对于每个在这个部门的员工显示一次。

代码清单14-22和输出14-9与代码清单14-21和输出14-8是相似的，区别在于两者的对象（Departments和Employees）是相反的。Join()的第一个参数变成了employees，表明departments要联接到employees。接下来的两个参数是Lambda表达式，指定了两个集合如何联接。其中，department => department.Id针对的是departments，而employee => employee.DepartmentId针对的是employees。和以前一样，凡是department.Id等于employee.DepartmentId的情形，都会发生一次联接。最后一个参数是匿名类型，其中包含了int Id、string Name和Employee Employee属性。

代码清单14-22 使用System.Linq.Enumerable.Join()执行的另一个内部联接

```
using System;
using System.Linq;

// ...

        Department[] departments = CorporateData.Departments;
        Employee[] employees = CorporateData.Employees;

        var items = departments.Join(
            employees,
            department => department.Id,
            employee => employee.DepartmentId,
            (department, employee) => new
            {
                department.Id,
                department.Name,
                Employee = employee
            });

        foreach (var item in items)
        {
            Console.WriteLine("{0}",
                item.Name);
            Console.WriteLine("\t" + item.Employee);
        }

// ...
```

输出14-9

```
Corporate
        Mark Michaelis (Chief Computer Nerd)
Finance
        Jewel Floch (Bookkeeper Extraordinaire)
Engineering
        Michael Stokesbary (Senior Computer Wizard)
Engineering
        Brian Jones (Enterprise Integration Guru)
Engineering
        Paul R. Bramsman (Programmer Extraordinaire)
Engineering
        Thomas Heavey (Software Architect)
Information Technology
        Robert Stokesbary (Expert Mainframe Engineer)
Research
        John Michaelis (Inventor)
```

14.4.7 使用GroupBy分组结果

除了排序和联接对象集合，经常还要对具有相似特征的对象进行分组。对于员工数据，可以按部门、地区、职务等对员工进行分组。代码清单14-23展示了如何使用GroupBy()标准查询操作符进行分组，输出14-10展示了结果。

代码清单14-23 使用System.Linq.Enumerable.GroupBy()对数据项进行分组

```
using System;
using System.Linq;

// ...

        IEnumerable<Employee> employees = CorporateData.Employees;

        IEnumerable<IGrouping<int, Employee>> groupedEmployees =
          employees.GroupBy((employee) => employee.DepartmentId);

        foreach(IGrouping<int, Employee> employeeGroup in
            groupedEmployees)
        {
            Console.WriteLine();
            foreach(Employee employee in employeeGroup)
            {
                Console.WriteLine("\t" + employee);
            }
            Console.WriteLine(
                "\tCount: " + employeeGroup.Count());
        }
    // ...
```

输出14-10

```
Mark Michaelis (Chief Computer Nerd)
    Count: 1

Michael Stokesbary (Senior Computer Wizard)
Brian Jones (Enterprise Integration Guru)
Paul R. Bramsman (Programmer Extraordinaire)
Thomas Heavey (Software Architect)
```

```
    Count: 4

Jewel Floch (Bookkeeper Extraordinaire)
    Count: 1

Robert Stokesbary (Expert Mainframe Engineer)
    Count: 1

John Michaelis (Inventor)
    Count: 1
```

注意，GroupBy()返回的是IGrouping<TKey, TElement>类型的数据项，该类型有一个属性指定了作为分组依据的键（employee.DepartmentId）。然而，它没有为组中的数据项准备一个属性。相反，由于IGrouping<TKey, TElement>是从IEnumerable<T>派生的，所以可以用foreach语句枚举组中的项，或者将数据聚合成像计数这样的东西（employeeGroup.Count()）。

14.4.8 使用GroupJoin()实现一对多关系

代码清单14-21和代码清单14-22几乎完全一致。只需改变一下匿名类型的定义，两个Join()调用就可产生相同的输出。如果想要创建的是员工列表，代码清单14-21提供的是正确的结果。department最终成为代表已联接员工的一个匿名类型的属性。但是，要想表示部门，代码清单14-22并不合适。由于支持集合，表示部门的理想方案是显示一个部门的所有员工的集合，而不是为每个部门—员工关系都创建一条匿名类型的记录。代码清单14-24展示了具体如何做，输出14-11展示了结果。

代码清单14-24 使用System.Linq.Enumerable.GroupJoin()创建子集合

```
using System;
using System.Linq;

// ...

        Department[] departments = CorporateData.Departments;
        Employee[] employees = CorporateData.Employees;

        var items = departments.GroupJoin(
            employees,
            department => department.Id,
            employee => employee.DepartmentId,
            (department, departmentEmployees) => new
            {
                department.Id,
                department.Name,
                Employees = departmentEmployees
            });

    foreach (var item in items)
    {
        Console.WriteLine("{0}",
            item.Name);
        foreach (Employee employee in item.Employees)
        {
            Console.WriteLine("\t" + employee);
```

```
            }
        }

    // ...
```

输出14-11

```
Corporate
        Mark Michaelis (Chief Computer Nerd)
Finance
        Jewel Floch (Bookkeeper Extraordinaire)
Engineering
        Michael Stokesbary (Senior Computer Wizard)
        Brian Jones (Enterprise Integration Guru)
        Paul R. Bramsman (Programmer Extraordinaire)
        Thomas Heavey (Software Architect)
Information Technology
        Robert Stokesbary (Expert Mainframe Engineer)
Research
        John Michaelis (Inventor)
```

为了获得理想的结果，我们使用了System.Linq.Enumerable的GroupJoin()方法。参数和代码清单14-21的差不多，区别在于最后一个以匿名类型形式传递的参数。在代码清单14-24中，Lambda表达式的类型是Func<Department, IEnumerable<Employee>, TResult>，其中的TResult是所选的匿名类型。注意，使用第二个类型参数（IEnumerable<Employee>）将每个部门的员工集合投射到结果的部门匿名类型中。

（熟悉SQL的读者会注意到，和Join()不同，SQL中没有与GroupJoin()等价的东西，这是由于SQL返回的数据是基于记录的，而且是不分层次结构的。）

高级主题：使用GroupJoin()实现外部联接

我们前面描述的内部联接称为同等联接（equi-joins），因为它们基于同等的键求值——只有在两个集合中都有的对象，它们的记录才会出现在结果集中。但在某些情况下，即使对应的对象不存在，也有必要创建一条记录。例如，虽然Marketing部门还没有员工，但在最终的部门列表中，是不应该把它忽略掉的。更好的做法是列出这个部门的名称，同时显示一个空白的员工列表。为此，可以执行一次左外部联接，这是通过组合使用GroupJoin()、SelectMany()和DefaultIfEmpty()来实现的。如代码清单14-25和输出14-12所示。

代码清单14-25 使用GroupJoin()和SelectMany()实现外部联接

```
using System;
using System.Linq;

// ...

        Department[] departments = CorporateData.Departments;
        Employee[] employees = CorporateData.Employees;

        var items = departments.GroupJoin(
            employees,
            department => department.Id,
            employee => employee.DepartmentId,
```

```
(department, departmentEmployees) => new
{
    department.Id,
    department.Name,
    Employees = departmentEmployees
}).SelectMany(
    departmentRecord =>
        departmentRecord.Employees.DefaultIfEmpty(),
    (departmentRecord, employee) => new
        {
            departmentRecord.Id,
            departmentRecord.Name,
            Employees =
                departmentRecord.Employees
        }).Distinct();

foreach (var item in items)
{
    Console.WriteLine("{0}",
        item.Name);
    foreach (Employee employee in item.Employees)
    {
        Console.WriteLine("\t" + employee);
    }
}

// ...
```

输出14-12

```
Corporate
        Mark Michaelis (Chief Computer Nerd)
Finance
        Jewel Floch (Bookkeeper Extraordinaire)
Engineering
        Michael Stokesbary (Senior Computer Wizard)
        Brian Jones (Enterprise Integration Guru)
        Paul R. Bramsman (Programmer Extraordinaire)
        Thomas Heavey (Software Architect)
Information Technology
        Robert Stokesbary (Expert Mainframe Engineer)
Research
        John Michaelis (Inventor)
Marketing
```

14.4.9 调用SelectMany()

我们偶尔需要处理由集合构成的集合。代码清单14-26展示了一个例子。在teams数组中包含两个球队，每个球队都有一个包含了球员的字符串数组。

代码清单14-26 调用SelectMany()

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...
```

```
var worldCup2006Finalists = new[]
{
    new
    {
        TeamName = "France",
        Players = new string[]
        {
            "Fabien Barthez", "Gregory Coupet",
            "Mickael Landreau", "Eric Abidal",
            "Jean-Alain Boumsong", "Pascal Chimbonda",
            "William Gallas", "Gael Givet",
            "Willy Sagnol", "Mikael Silvestre",
            "Lilian Thuram", "Vikash Dhorasoo",
            "Alou Diarra", "Claude Makelele",
            "Florent Malouda", "Patrick Vieira",
            "Zinedine Zidane", "Djibril Cisse",
            "Thierry Henry", "Franck Ribery",
            "Louis Saha", "David Trezeguet",
            "Sylvain Wiltord",
        }
    },
    new
    {
        TeamName = "Italy",
        Players = new string[]
        {
            "Gianluigi Buffon", "Angelo Peruzzi",
            "Marco Amelia", "Cristian Zaccardo",
            "Alessandro Nesta", "Gianluca Zambrotta",
            "Fabio Cannavaro", "Marco Materazzi",
            "Fabio Grosso", "Massimo Oddo",
            "Andrea Barzagli", "Andrea Pirlo",
            "Gennaro Gattuso", "Daniele De Rossi",
            "Mauro Camoranesi", "Simone Perrotta",
            "Simone Barone", "Luca Toni",
            "Alessandro Del Piero", "Francesco Totti",
            "Alberto Gilardino", "Filippo Inzaghi",
            "Vincenzo Iaquinta",
        }
    }
};

IEnumerable<string> players =
    worldCup2006Finalists.SelectMany(
        team => team.Players);

Print(players);

// ...
```

在这个代码清单的输出中，会按照每个球员在代码中出现的顺序逐行显示他们的姓名。Select()和SelectMany()的区别在于，Select()会返回两个球员数组，每个都对应于原始集合中的球员数组。Select()可以一边投射一边转换，但项的数量不会发生改变。例如，teams.Select (team => team.Players)会返回一个IEnumerable<string[]>。

而SelectMany()会遍历由Lambda表达式（早先由Select()选择的数组）标识的每一项，并将每一项都放到一个新集合中。新集合整合了子集合中的所有项。所以，不像Select()那样返回两个球员数组，SelectMany()是将选择的每个数组都整合起来，把其中的项整合到一个集合中[①]。

14.4.10 更多标准查询操作符

代码清单14-27展示了如何使用由Enumerable提供的一些更简单的API；输出14-13展示了结果。

代码清单14-27 更多的System.Linq.Enumerable方法调用

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

class Program
{
  static void Main()
  {
      IEnumerable<object> stuff =
        new object[] { new object(), 1, 3, 5, 7, 9,
            "\"thing\"", Guid.NewGuid() };
      Print("Stuff: {0}", stuff);
      IEnumerable<int> even = new int[] { 0, 2, 4, 6, 8 };
      Print("Even integers: {0}", even);

      IEnumerable<int> odd = stuff.OfType<int>();
      Print("Odd integers: {0}", odd);

      IEnumerable<int> numbers = even.Union(odd);
      Print("Union of odd and even: {0}", numbers);

      Print("Union with even: {0}", numbers.Union(even));
      Print("Concat with odd: {0}", numbers.Concat(odd));
      Print("Intersection with even: {0}",
          numbers.Intersect(even));
      Print("Distinct: {0}", numbers.Concat(odd).Distinct());
      if (!numbers.SequenceEqual(
          numbers.Concat(odd).Distinct()))
      {
```

---

① 作者的这个例子过于复杂以至于可能不好理解。下面是一个简单的例子：

```
string[] text = { "Zhou Jing",  "walking in",  "the sideway" };
var tokens = text.Select(s => s.Split(' '));
foreach (string[] line in tokens)
    foreach (string token in line)
        Console.Write("{0}-", token);
```

如果使用Select()，那么获得的tokens是一个string[]数组，还需要遍历每个string[]，才能解析出其中的string。相反，如果换用SelectMany()，那么不仅代码量减少了，也更容易理解：

```
string[] text = { "Zhou Jing",  "walking in",  "the sideway" };
var tokens = text.SelectMany(s => s.Split(' '));
foreach (string  token in tokens)
        Console.Write("{0}-", token);
```

——译者注

```
        throw new Exception("Unexpectedly unequal");
    }
    else
    {
        Console.WriteLine(
            @"Collection ""SequenceEquals""" +
            " collection.Concat(odd).Distinct())");
        Print("Reverse: {0}", numbers.Reverse());

        Print("Average: {0}", numbers.Average());
        Print("Sum: {0}", numbers.Sum());
        Print("Max: {0}", numbers.Max());
        Print("Min: {0}", numbers.Min());
    }
}

private static void Print<T>(
    string format, IEnumerable<T> items)
{
    StringBuilder text = new StringBuilder();
    foreach (T item in items.Take(items.Count()-1))
    {
        text.Append(item + ", ");
    }
    text.Append(items.Last());

    Console.WriteLine(format, text);
}

private static void Print<T>(string format, T item)
{
    Console.WriteLine(format, item);
}
}
```

输出14–13

```
Stuff: System.Object, 1, 3, 5, 7, 9, "thing"
24c24a41-ee05-41b9-958e-50dd12e3981e
Even integers: 0, 2, 4, 6, 8
Odd integers: 1, 3, 5, 7, 9
Union of odd and even: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Union with even: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Concat with odd: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9, 1, 3, 5, 7, 9
Intersection with even: 0, 2, 4, 6, 8
Distinct: 0, 2, 4, 6, 8, 1, 3, 5, 7, 9
Collection "SequenceEquals"collection.Concat(odd).Distinct())
Reverse: 9, 7, 5, 3, 1, 8, 6, 4, 2, 0
Average: 4.5
Sum: 45
Max: 9
Min: 0
```

在代码清单14-27中，所有API调用都不需要Lambda表达式。表14-1和表14-2对这些方法进行了总结。

System.Linq.Enumerable提供了一系列聚合函数，它们能枚举集合并计算结果。本章已经利用了其中一个聚合函数Count()。

注意，表14-1和表14-2列出的所有方法都会触发推迟执行。

表14-1 更简单的查询操作符

方 法	描 述
OfType<T>()	构造一个查询来查询集合，只返回特定类型的项，类型由OfType<T>()的类型参数指定
Union()	合并两个集合，生成两个集合中的所有项的超集。结果集合不包含重复的项，即使同一个项在两个集合中都存在
Concat()	合并两个集合，生成两个集合的超集。重复的项不从结果集中删除。Concat()会保留原先的顺序。也就是说，{A，B}和{C，D}连接，结果是{A，B，C，D}
Intersect()	在结果集合中填充两个原始集合中都有的项
Distinct()	筛选集合中重复的项，使结果集合中的每一项都是不重复的
SequenceEquals()	比较两个集合，返回一个布尔值来指出集合是否一致，项的顺序也是作比较依据之一（这有助于测试结果是否符合预期）
Reverse()	反转集合中各个项的顺序，以便在遍历集合时，以相反的顺序进行遍历

表14-2 System.Linq.Enumerable提供的聚合函数

聚合函数	描 述
Count()	返回集合中的项的总数
Average()	计算数值集合（由一个数值键选择器指定）的平均值
Sum()	数值集合求和
Max()	判断数值集合的最大值
Min()	判断数值集合的最小值

高级主题：IQuerable<T>的Queryable扩展

有一个接口几乎和IEnumerable<T>完全一样，这就是IQueryable<T>。由于IQueryable<T>是从IEnumerable<T>派生的，所以它有IEnumerable<T>的“全部”成员——但只有那些直接声明的，例如GetEnumerator()。扩展方法是不会继承的。所以，IQueryable<T>没有Enumerable的任何扩展方法。然而，它有一个类似的扩展类，称为System.Linq.Queryable。Enumerable为IEnumerable<T>添加的几乎所有方法，Queryable都会为IQueryable<T>添加。所以，IQueryable<T>提供了一个与IEnumerable<T>非常相似的编程接口。

那么，是什么使IQueryable<T>显得与众不同呢？答案是可通过它实现自定义的LINQ Provider（LINQ提供程序）。LINQ Provider的作用是将表达式分解成各个组成部分。一经分解，表达式就可以转换成另一种语言，可以序列化以便在远程执行，可以通过一个异步执行模式来注入，等等。简单地说，LINQ Provider为一个标准集合API提供了一种“解释”机制。利用这种几乎没有任何限制的功能，可以注入与查询和集合有关的行为。

例如，利用LINQ Provider，可以将查询表达式从C#转换成SQL，然后在远程数据库中执行。这样一来，C#程序员就可以继续使用他熟悉的面向对象语言，将向SQL的转换留给底层的LINQ Provider去完成。通过这种类型的表达式，程序语言可以轻松协调“面向对象编程”和“关系数据库”。

在IQueryable<T>的情况下，尤其要警惕推迟执行的问题。例如，假定一个LINQ Provider负责从数据库返回数据，那么不是不顾选择条件，直接从数据库中获取数据。相反，Lambda表达式应提供IQueryable<T>的一个实现，其中可能要包含上下文信息，比如连接字符串，但不要包含数据。除非调用GetEnumerator()（甚至MoveNext()），否则不会发生真正的数据获取操作。然而，GetEnumerator()调用一般是隐式发生的，比如在用foreach遍历集合，或者调用一个Enumerable方法时（比如Count<T>()或Cast<T>()）。很明显，针对此类情况，开发人员需要提防在不知不觉中反复调用代价高昂的、可能涉及推迟执行的操作。例如，假定调用GetEnumerator()会造成通过网络来分布式地访问数据库，就应避免因调用Count()或foreach而不经意地造成对集合的重复遍历。

14.5 小结

在介绍了匿名类型、隐式变量和集合初始化器之后，本章描述了foreach循环的内部工作机制以及为了执行它需要什么接口。除此之外，开发者经常都要筛选集合来减少数据项的数量。另外，还经常需要对集合进行投射，使数据项改换一种不同的格式。针对这些问题，本章讨论了如何使用标准查询操作符——由System.Linq.Enumerable类提供的一些常用的集合API——来执行集合处理。

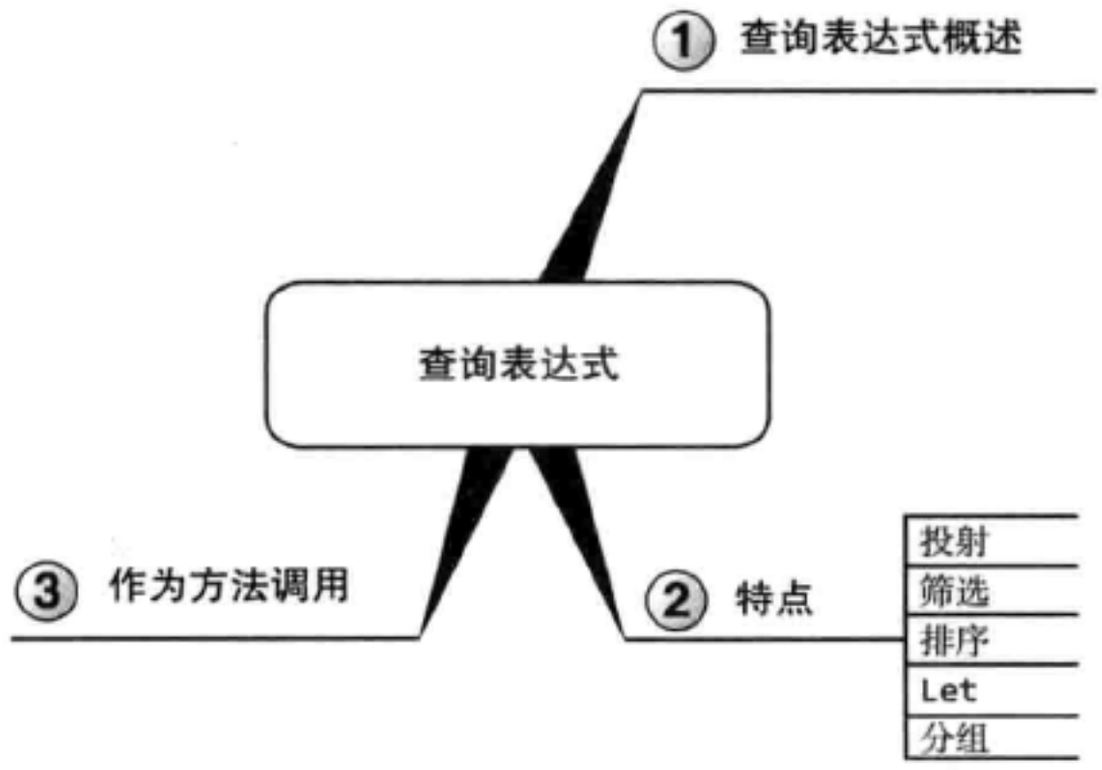
在介绍标准查询操作符的时候，本章用几页的篇幅描述了推迟执行的原理，并提醒开发者注意这个问题。一个不经意的调用可能会枚举集合内容，造成表达式被反复执行。推迟执行会造成标准查询操作符被“隐式”地执行。开发者一定要留意这个问题，尤其是在查询的开销比较大的时候。程序员应将查询对象视为查询而不是结果。而且，即使查询已经执行过，它也有可能要再次从头执行。查询对象不知道结果和上一次执行的结果（如果存在的话）是否一样。

之所以将代码清单14-25放到一个“高级主题”中，是因为连续调用多个标准查询操作符有一定的复杂性。虽然可能经常会遇到这样的需求，但并非一定要直接依靠标准查询操作符。C# 3.0支持一个名为“查询表达式”的概念。这是一种类似于SQL的语法，可以利用它来更方便地操纵集合，而且写出来的代码显得更容易理解。这将是下一章要讲述的主题。

第15章

使用查询表达式的LINQ

在第14章展示的一个查询中（如代码清单14-25所示）使用了标准查询操作符GroupJoin()、SelectMany()和Distinct()——另外还创建了两个匿名类型。结果是一个跨越许多行的语句，它很难理解。事实上，这比以前用C# 2.0写的语句都要复杂得多，尽管它表面上是和C# 2.0的语法完全兼容的。不过，有时也确实需要这种复杂的语句。在这种情况下，使用标准查询操作符显得更方便，即使结果代码可能比较复杂和难以理解。另外，标准查询操作符所实现的查询在功能上与SQL中实现的查询非常相似。



考虑到这两方面的因素，C#语言的设计者决定在C# 3.0中添加一个新的语法：**查询表达式**。使用查询表达式，许多标准查询操作符都能转换成更容易理解的代码，也就是和SQL的风格非常接近的代码。

本章将介绍新的查询表达式语法，并利用这个语法 改写上一章的许多查询。

15.1 查询表达式概述

除了遍历集合中的所有项，开发者经常还要对集合进行**筛选**来删除不想要的项，以及对集合进行**投射**将其中的项变成其他形式。例如，假定有一个文件集合，我们可以筛选它来创建一个新集合，使新集合中只包含.cs文件，或者只包含超过1 MB的文件。我们还可以投射文件集合来创建一个新集合，其中只包含两项数据：文件的目录路径以及目录的大小。查询表达式为这两种常规操作提供了直观易懂的语法。代码清单15-1展示了筛选字符串集合的查询表达式，结果如输出15-1所示。

代码清单15-1 简单的查询表达式

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

  static string[] Keywords = {
      "abstract", "add*", "alias*", "as", "ascending*",
      "async*", "await*", "base","bool", "break",
      "by*", "byte", "case", "catch", "char", "checked",
      "class", "const", "continue", "decimal", "default",
      "delegate", "descending*", "do", "double",
      "dynamic*", "else", "enum", "event", "equals*",
      "explicit", "extern", "false", "finally", "fixed",
      "from*", "float", "for", "foreach", "get*", "global*",
      "group*", "goto", "if", "implicit", "in", "int",
      "into*", "interface", "internal", "is", "lock", "long",
      "join*", "let*", "namespace", "new", "null", "object",
      "on*", "operator", "orderby*", "out", "override",
      "params", "partial*", "private", "protected", "public",
      "readonly", "ref", "remove*", "return", "sbyte", "sealed",
      "select*", "set*", "short", "sizeof", "stackalloc",
      "static", "string", "struct", "switch", "this", "throw",
      "true", "try", "typeof", "uint", "ulong", "unchecked",
      "unsafe", "ushort", "using", "value*", "var*", "virtual",
      "void", "volatile", "where*", "while", "yield*"};

private static void ShowContextualKeywords1()
{
    IEnumerable<string> selection =
        from word in Keywords
            where !word.Contains('*')
            select word;

    foreach (string keyword in selection)
    {
        Console.Write(keyword + " ");
    }
}

// ...
```

输出15-1

```
abstract as base bool break byte case catch char checked class const
continue decimal default delegate do double else enum event explicit
extern false finally fixed float for foreach goto if implicit in int
interface internal is lock long namespace new null object operator out
override params private protected public readonly ref return sbyte
sealed short sizeof stackalloc static string struct switch this throw
true try typeof uint ulong unchecked unsafe ushort using virtual void
volatile while
```

在这个查询表达式中，我们将C#关键字集合赋给selection。where子句筛选出非上下文关键字（不含星号的那些）。

查询表达式总是以“from子句”开始，以“select子句”或者“groupby子句”结束。这些子句分别用上下文关键字from、select或group来标识。from子句中的标识符word称为**范围变量**（range variable），代表集合中的每一项。这就像是foreach循环中的循环变量代表集合中的每一项。

熟悉SQL的开发者会发现，查询表达式采用了和SQL非常相似的语法。这个设计是故意的，目的是使SQL的熟手很容易掌握LINQ。但是，两者又存在一些明显的区别。其中最明显的是C#查询表达式的子句顺序首先是from子句，然后是where子句，最后才是select子句。而对应的SQL查询首先是SELECT子句，然后是FROM子句，最后是WHERE子句。

之所以要采用这个设计，目的是为了方便实现智能感知功能，或者说是为了方便预测被选择的对象的成员。例如，由于最开始出现的是from，并将字符串数组Keywords指定为数据源，所以代码编辑器知道word是string类型。这样一来，Visual Studio IDE的智能感知功能就可以立即发挥作用——在word之后输入一个成员访问操作符（一个圆点符号），将只显示出string的成员。相反，如果from子句出现在select之后（就像SQL那样），那么from子句之前的任何圆点操作符都无法确定word的数据类型是什么，所以无法显示word的成员列表。例如，在代码清单15-1中，就不可能预测Contains()是word的一个成员。

C#查询表达式的顺序其实更接近各个操作在逻辑上的顺序。对查询进行求值时，首先指定集合（from子句），再筛选出想要的项（where子句），最后描述希望的结果（select子句）。

最后，C#查询表达式的顺序确保（范围）变量的作用域规则与局部变量的规则一致。例如，子句（通常是from子句）必须先声明范围变量，然后才能使用这个变量。这类似于局部变量必须先声明再使用。

15.1.1 投射

查询表达式的结果是IEnumerable<T>或IQueryable<T>类型的集合[①]。T的实际类型是从select或group by子句推导的。例如在代码清单15-1中，编译器知道keywords是string[]类型，能转换成IEnumerable<string>，所以推导出word是string类型。查询以select word结尾，所以查询表达式的结果肯定是字符串集合，所以查询表达式的类型是IEnumerable<string>。

① 在实际应用中，查询表达式输出的几乎总是IEnumerable<T>或者它的派生类型。但理论上不一定非要这样。完全可以实现查询操作符方法来返回其他类型。语言不要求查询表达式的结果必须能转换成操作符IEnumerable<T>。

在这个例子中，查询的“输入”和“输出”都是字符串集合。但是，如果select子句中的表达式是完全不同的类型，那么“输出”类型可以有别于“输入”类型。例如代码清单15-2的查询表达式，结果如代码清单15-2所示。

代码清单15-2 使用查询表达式来投射

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

// ...

    static void List1(string rootDirectory, string searchPattern)
    {
        IEnumerable<string> fileNames = Directory.GetFiles(
            rootDirectory, searchPattern);
        IEnumerable<FileInfo> fileInfos =
            from fileName in fileNames
            select new FileInfo(fileName);

        foreach (FileInfo fileInfo in fileInfos)
        {
              Console.WriteLine(".{0}({1})",
                  fileInfo.Name, fileInfo.LastWriteTime);
        }
    }

// ...
```

输出15-2

```
Account.cs(11/22/2011 11:56:11 AM)
Bill.cs(8/10/2011 9:33:55 PM)
Contact.cs(8/19/2011 11:40:30 PM)
Customer.cs(11/17/2011 2:02:52 AM)
Employee.cs(8/17/2011 1:33:22 AM)
Person.cs(10/22/2011 10:00:03 PM)
```

这个查询表达式的结果是一个IEnumerable<FileInfo>，而不是Directory.GetFiles()返回的IEnumerable<string>数据类型。查询表达式的select子句可以将from子句的表达式所收集到的东西投射到完全不同的数据类型中。

注意在这个例子中，选择FileInfo是因为它恰好有输出所需的两个字段，即文件名和上一次写入时间。如果需要FileInfo对象没有捕捉到的信息，那么或许没有现成类型可供使用。这时需要使用匿名类型，它可以简单、方便地投射你需要的数据，同时不必寻找或创建一个显式的类型。（事实上，这正是在语言中加入匿名类型的关键原因。）代码清单15-3 产生和代码清单15-2相似的输出，但它是通过匿名类型，而不是通过FileInfo。

代码清单15-3 在查询表达式中使用匿名类型

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
```

```
// ...

  static void List2(string rootDirectory, string searchPattern)
  {
        var fileNames =Directory.GetFiles(
            rootDirectory, searchPattern)
        var fileResults =
            from fileName in fileNames
            select new
            {
                Name = fileName,
                LastWriteTime = File.GetLastWriteTime(fileName)
            };

        foreach (var fileResult in fileResults)
        {
            Console.WriteLine("{0}({1})",
                fileResult.Name, fileResult.LastWriteTime);
        }
    }

  // ...
```

在这个例子中，我们在查询结果中只投射出了文件名和它的最后一次写入时间。假如处理的数据规模不大（如FileInfo），像代码清单15-3这样的投射对程序的增强并不是特别明显。然而，假如数据量非常大，而且检索这些数据的代价非常高（例如要通过Internet从一台远程计算机上获取），那么像这样在“水平”方向上进行投射，从而减少与集合中的每一项关联的数据量，就可以发挥出非常积极的作用。利用匿名类型，执行查询时可以不必获取全部数据，而是只在集合中存储和获取需要的数据。

例如，假定一个大型数据库的某个表包含30个以上的列。没有匿名类型，开发人员要么使用含有不需要信息的对象，要么定义一些小的专用类，这些类只存储需要的特定数据。而匿名类型允许由编译器（动态）定义类型。在这种类型中，只包含当前情况所需的数据。在其他情况下，则可以采用不同的投射方式来投射仅供那个情况需要的属性。

初学者主题：查询表达式和推迟执行

上一章已讲过“推迟执行”的主题，同样的道理也适用于查询表达式。来考虑代码清单15-1中对selection的赋值。创建查询和向变量赋值不会执行查询，而只是生成代表查询的对象。换言之，查询对象创建时不会调用word.Contains("*")方法。查询表达式只是存储了一个选择条件（查询标准）。以后在遍历由selection变量所标识的集合时会用到这个条件。

为了体验这具体是如何发生的，来看看代码清单15-4和输出15-3。

代码清单15-4 推迟执行和查询表达式（示例1）

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...
```

```
private static void ShowContextualKeywords2()
{
    IEnumerable<string> selection = from word in Keywords
                                    where IsKeyword(word)
                                    select word;
    Console.WriteLine("Query created.");
    foreach (string keyword in selection)
    {
        // No space output here.
        Console.Write(keyword);
    }
}

// The side effect of console output is included
// in the predicate to demonstrate deferred execution;
// predicates with side effects are a poor practice in
// production code.
private static bool IsKeyword(string word)
{
    if (word.Contains('*'))
    {
        Console.Write(" ");
        return true;
    }
    else
    {
        return false;
    }
}
// ...
```

输出15-3

```
Query created.
 add* alias* ascending* by* descending* dynamic* equals* from* get*
global* group* into* join* let* on* orderby* partial* remove* select*
set* value* var* where* yield*
```

注意在代码清单15-4中，foreach循环内部是没有输出空格的。上下文关键字之间的空格是在IsKeyword()方法中输出的，这证明了在代码遍历selection的时候IsKeyword()方法才会得到调用，而不是在对selection赋值的时候就调用。

总之，虽然selection是集合（毕竟它的类型是IEnumerable<T>），但在赋值时，from子句之后的一切都构成了选择条件。遍历selection时才会真正应用这些条件。

来考虑第二个例子，如代码清单15-5和输出15-4所示。

代码清单15-5 推迟执行和查询表达式（示例2）

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

  private static void CountContextualKeywords()
  {
      int delegateInvocations = 0;
```

```
Func<string, string> func =
    text=>
    {
        delegateInvocations++;
        return text;
    };

IEnumerable<string> selection =
    from keyword in Keywords
    where keyword.Contains('*')
    select func(keyword);


Console.WriteLine(
    "1. delegateInvocations={0}", delegateInvocations);

// Executing count should invoke func once for
// each item selected.
Console.WriteLine(
    "2. Contextual keyword count={0}", selection.Count());

Console.WriteLine(
    "3. delegateInvocations={0}", delegateInvocations);

// Executing count should invoke func once for
// each item selected.
Console.WriteLine(
    "4. Contextual keyword count={0}", selection.Count());

Console.WriteLine(
    "5. delegateInvocations={0}", delegateInvocations);

// Cache the value so future counts will not trigger
// another invocation of the query.
List<string> selectionCache = selection.ToList();

Console.WriteLine(
    "6. delegateInvocations={0}", delegateInvocations);

// Retrieve the count from the cached collection.
Console.WriteLine(
    "7. selectionCache count={0}",selectionCache.Count());

Console.WriteLine(
    "8. delegateInvocations={0}", delegateInvocations);

}

// ...
```

输出15-4

```
1. delegateInvocations=0
2. Contextual keyword count=15
3. delegateInvocations=15
4. Contextual keyword count=15
5. delegateInvocations=30
6. delegateInvocations=45
7. selectionCache count=15
8. delegateInvocations=45
```

代码清单15-5不是定义一个单独的方法，而是用一个语句Lambda来统计方法的调用次数。

输出中有3个地方值得注意。首先，注意在selection被赋值之后，delegateInvocations保持为零。在对selection进行赋值时，还没有发生对Keywords的遍历。如果Keywords是属性，那么属性调用会发生。换言之，from子句会在赋值时执行。但是，除非代码开始遍历selection中的值，否则无论投射、筛选，还是from子句之后的一切都不会执行。好像在赋值的时候，selection的一个更恰当的称呼是“查询”。

然而，一旦调用Count()，选择或者项等暗示容器或集合的称呼就显得恰当了，因为我们要开始对集合中的项进行计数。换言之，变量selection扮演着双重角色，第一是保存查询，第二是作为可从中获取数据的容器。

第二个要注意的地方是，每次为selection调用Count()，都会对每个被选择的项执行func。由于selection同时扮演了查询和集合两个角色，所以请求计数要求再次执行查询，遍历selection引用的IEnumerable<string>集合，并统计数据项的个数。C#编译器不知道是否有人修改了数组中的字符串而造成计数发生变化，所以每次都要重新计数，保证答案总是正确的和最新的。类似地，如果对selection执行foreach循环，那么也会针对selection中的每一项调用一次func。调用System.Linq.Enumerable提供的其他所有扩展方法时，这个道理都是适用的。

高级主题：实现推迟执行

推迟执行通过委托和表达式树来实现。委托允许创建和操纵方法的引用，方法中含有可在以后调用的表达式。类似地，利用表达式树，可创建和操纵与表达式有关的信息，这种表达式能在以后检查和处理。

在代码清单15.5中，where子句的谓词表达式和select子句的投射表达式由编译器转换成Lambda表达式，再转换成委托创建。查询表达式的结果是包含了委托引用的对象。只有在遍历查询结果时，查询对象才实际地执行委托。

15.1.2 筛选

代码清单15-1中用一个where子句来筛选出除了上下文关键字之外的其他C#关键字。where子句在“垂直”方向上筛选集合，结果集合将包含较少的项（每个数据项都相当于数据表中的一行记录）。筛选条件（filter criteria）用谓词表示。所谓谓词，本质上就是返回布尔值的Lambda表达式，比如word.Contains()（如代码清单15-1所示）或者File.GetLastWriteTime(file) <DateTime.Now.AddMonths(-1)（如代码清单15-6和输出15-5所示）。

代码清单15-6 用where子句进行筛选的查询表达式

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

// ...

  static void FindMonthOldFiles(
```

```
        string rootDirectory, string searchPattern)
    {
        IEnumerable<FileInfo> files =
            from fileName in Directory.GetFiles(
                rootDirectory, searchPattern)
            where File.GetLastWriteTime(fileName) <
                DateTime.Now.AddMonths(-1)
            select new FileInfo(fileName);

        foreach (FileInfo file in files)
        {
            //  As simplification, current directory is
            //  assumed to be a subdirectory of
            //  rootDirectory
            string relativePath = file.FullName.Substring(
                    Environment.CurrentDirectory.Length);
            Console.WriteLine(".{0}({1})",
                relativePath, file.LastWriteTime);
        }
    }

  // ...
```

输出15–5

```
.\TestData\Bill.cs(8/10/2011 9:33:55 PM)
.\TestData\Contact.cs(8/19/2011 11:40:30 PM)
.\TestData\Employee.cs(8/17/2011 1:33:22 AM)
.\TestData\Person.cs(10/22/2011 10:00:03 PM)
```

15.1.3 排序

在查询表达式中对数据项进行排序的是orderby子句，如代码清单15-7所示。

代码清单15–7 使用orderby子句进行排序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

// ...
  static void ListByFileSize1(
      string rootDirectory, string searchPattern)
  {
      IEnumerable<string> fileNames =
          from fileName in Directory.GetFiles(
              rootDirectory, searchPattern)
          orderby (new FileInfo(fileName)).Length descending,
              fileName
          select fileName;

      foreach (string fileName in fileNames)
      {
          Console.WriteLine("{0}", fileName);
      }
  }
// ...
```

代码清单15-7使用orderby子句来排序由Directory.GetFiles()返回的文件，首先按文件长度降序排序，然后按文件名升序排序。多个排序条件以逗号分隔。在这个例子中，如果两个文件的长度相同，就再按文件名排序。ascending和descending是上下文关键字，分别指定以升序或降序排序。将排序顺序指定为升序或降序是可选的（例如，filename就没有指定）。如果没有指定排序指定，就默认为ascending。

15.1.4 let子句

代码清单15-8的查询与代码清单15-7的查询非常相似，只是IEnumerable<T>的类型参数是FileInfo。但代码清单15-8的一个问题在于： FileInfo要创建两次，分别在orderby和select子句中创建。

代码清单15-8 投射一个FileInfo集合并按文件大小排序

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;

// ...
  static void ListByFileSize2(
      string rootDirectory, string searchPattern)
  {
      IEnumerable<FileInfo> files =
          from fileName in Directory.GetFiles(
              rootDirectory, searchPattern)
          orderby new FileInfo(fileName).Length, fileName
          select new FileInfo(fileName);

      foreach (FileInfo file in files)
      {
          //  As simplification, current directory is
          //  assumed to be a subdirectory of
          //  rootDirectory
          string relativePath = file.FullName.Substring(
              Environment.CurrentDirectory.Length);
          Console.WriteLine(".{0}({1})",
              relativePath, file.Length);
      }
  }
// ...
```

遗憾的是，虽然结果正确，但代码清单15-8会为源集合中的每一项都实例化FileInfo对象两次。可以用let子句避免这种无谓的、可能非常昂贵的开销，如代码清单15-9所示。

代码清单15-9 用let避免多余的实例化

```
        // ...
        IEnumerable<FileInfo> files =
            from fileName in Directory.GetFiles(
                rootDirectory, searchPattern)
            let file = new FileInfo(fileName)
            orderby file.Length, fileName
            select file;
         // ...
```

let子句引入了一个新的范围变量，它容纳的表达式值可以在查询表达式剩余的部分使用。可添加任意数量的let表达式，只需把它们每一个作为一个附加的子句，放在第一个from子句之后、最后一个select/group by子句之前，加入查询即可。

15.1.5 分组

另一个常见的数据处理情形是对相关的数据项进行分组。在SQL中，这通常涉及对数据项进行聚合以生成一个汇总或合计或其他聚合值（aggregate value）。然而，LINQ的表达力更强一些。LINQ表达式允许将单独的项分组到一系列子集合中，还允许那些组与所查询的集合中的项关联。例如，可以将C#的上下文关键字和普通关键字分成两组，并自动将单独的单词与这两个组关联。代码清单15-10和输出15-6对此进行了演示。

代码清单15-10 对查询结果进行分组

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

  private static void GroupKeywords1()
  {
      IEnumerable<IGrouping<bool, string>> selection =
          from word in Keywords
          group word by word.Contains('*');

      foreach (IGrouping<bool, string> wordGroup
          in selection)
      {
          Console.WriteLine(Environment.NewLine + "{0}:",
              wordGroup.Key ?
                  "Contextual Keywords" : "Keywords");
          foreach (string keyword in wordGroup)
          {
              Console.Write(" " +
                  (wordGroup.Key ?
                      keyword.Replace("*", null) : keyword));
          }
      }
  }

// ...
```

输出15-6

```
Keywords:
 abstract as base bool break byte case catch char checked class const
continue decimal default delegate do double else enum event explicit
extern false finally fixed float for foreach goto if implicit in int
interface internal is lock long namespace new null object operator out
override params private protected public readonly ref return sbyte
sealed short sizeof stackalloc static string struct switch this throw
true try typeof uint ulong unchecked unsafe ushort using virtual void
volatile while
Contextual Keywords:
 add alias ascending async await by descending dynamic equals from
get global group into join let on orderby partial remove select
set value var where yield
```

这个代码清单有几个地方需要注意。首先，查询结果是一系列IGrouping<bool, string>类型的元素。第一个类型实参指出by关键字后的“group key”（分组依据）表达式是bool类型，第二个类型实参指出group关键字后的“group element”（分组元素）表达式是string类型。也就是说，查询将生成一系列分组，将相同的布尔类型键应用于组内的每个string。

由于含有groupby子句的查询会产生一系列集合，所以对结果进行迭代的常用模式是创建嵌套的foreach循环。在代码清单15-10中，外层循环遍历各个分组，打印关键字的类型（上下文还是普通关键字）作为标题。嵌套的foreach循环则在标题下打印组中的每个关键字。

由于查询表达式的结果本身是一个序列，所以可以像查询其他任何序列那样查询结果序列。代码清单15-11和输出15-7 展示了如何创建附加的查询，为生成一系列分组的查询增加投射。（下一节将讨论查询延续，将展示如何采用更顺眼的语法为一个完整的查询添加额外的查询子句。）

代码清单15-11 在group子句后面选择一个匿名类型

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

private static void GroupKeywords1()
{
    IEnumerable<IGrouping<bool, string>> keywordGroups =
        from word in Keywords
        group word by word.Contains('*');

    var selection =
        from groups in keywordGroups
        select new
        {
            IsContextualKeyword = groups.Key,
            Items = groups
        };

    foreach (var wordGroup in selection)
    {
        Console.WriteLine(Environment.NewLine + "{0}:",
            wordGroup.IsContextualKeyword ?
                "Contextual Keywords" : "Keywords");
        foreach (var keyword in wordGroup.Items)
        {
            Console.Write(" " +
                keyword.Replace("*", null));
        }
    }
}

// ...
```

输出15-7

```
Keywords:
 abstract as base bool break byte case catch char checked class const
continue decimal default delegate do double else enum event explicit
extern false finally fixed float for foreach goto if implicit in int
interface internal is lock long namespace new null object operator out
override params private protected public readonly ref return sbyte
sealed short sizeof stackalloc static string struct switch this throw
true try typeof uint ulong unchecked unsafe ushort using virtual void
volatile while
Contextual Keywords:
 add alias ascending async await by descending dynamic equals from
get global group into join let on orderby partial remove select
set value var where yield
```

group子句使查询返回由IGouping<TKey, TElement>对象构成的集合——这和第14章讲过的标准查询操作符GroupBy()是一样的。select子句定义了一个匿名类型，将IGrouping<TKey, TElement>.Key重命名为IsContextualKeyword，并命名了子集合属性Items。在进行了这些修改之后，嵌套的foreach循环就可以使用wordGroup.Items，而不必像代码清单15-10那样直接使用wordGroup。另外，有的人认为还可以在匿名类型中添加一个属性来表示子集合中的数据项的个数。然而，这个功能已经由wordGroup.Items.Count()提供，所以直接把它添加到匿名类型中是没有必要的。

15.1.6 使用into进行查询延续

如代码清单15-11所示，一个现有的查询可作为另一个查询的输入。然而，要将一个查询的结果作为另一个的输入，没必要写一个全新的查询表达式，而是可以使用上下文关键字into，通过**查询延续子句**来扩展任何查询。查询延续是一个语法糖，能简单地表示“创建两个查询并将第一个用作第二个的输入”。into子句引入的范围变量（代码清单15-11中的groups）成为查询剩余部分的范围变量；之前的任何范围变量在逻辑上成为之前查询的一部分，不可在查询延续中使用。代码清单15-12展示了如何重写代码清单15-11来使用查询延续，而不是使用两个查询。

代码清单15-12 用查询延续子句扩展查询

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...

private static void GroupKeywords1()
{
    var selection =
        from word in Keywords
        group word by word.Contains('*')
        into groups
            select new
            {
                IsContextualKeyword = groups.Key,
                Items = groups
            };

    // ...
```

```
    }

// ...
```

使用into在现有查询结果上运行附加的查询，这不是只有以group子句结尾的查询才有的功能，而是所有查询表达式都可用的功能。查询延续简单地实现了在查询表达式中使用其他查询表达式的结果。into相当于一个管道操作符，它将第一个查询的结果“管道传送”给第二个查询。用这种方式可以链接起任意数量的查询。

15.1.7 用多个from子句“平整”序列的序列

经常需要将一个序列的序列“平整”（flatten）成单个序列。例如，一系列客户中的每个客户都可能关联了一系列订单，或者一系列目录中的每个目录都关联了一系列文件。SelectMany序列操作符（在第14章讨论）可以连接所有子序列；要用查询表达式语法做相同的事情，可以使用多个from子句，如代码清单15-13所示。

代码清单15-13 多个选择

```
var selection =
    from word in Keywords
    from character in word
    select character;
```

上述查询生成字符序列a, b, s, t, r, a, c, t, a, s, b, a, s, e, …。

还可用多个from子句生成笛卡儿积——几个序列所有可能的组合，如代码清单15-14所示。

代码清单15-14 笛卡儿积

```
var numbers = new[] { 1, 2, 3 };
var product =
    from word in Keywords
    from number in numbers
    select new {word, number};
```

这样生成一系列对(abstract, 1)、(abstract, 2)、(abstract, 3)、(as, 1)、(as, 2)，等等。

初学者主题：不重复的成员

我们经常希望在集合中只保留不重复的项——丢弃重复的项。查询表达式没有专门的语法来做到这一点。但是，如上一章所述，可以通过查询操作符Distinct()来实现这个功能。为了向查询表达式应用查询操作符，表达式必须放到圆括号中，防止编译器以为对Distinct()的调用是select子句的一部分。代码清单15-15给出了一个例子，输出15-8是结果。

代码清单15-15 从查询表达式获取不重复的成员

```
using System;
using System.Collections.Generic;
using System.Linq;

// ...
```

```
public static void ListMemberNames()
{
    IEnumerable<string> enumerableMethodNames = (
        from method in typeof(Enumerable).GetMembers(
            System.Reflection.BindingFlags.Static |
            System.Reflection.BindingFlags.Public)
        select method.Name).Distinct();
    foreach(string method in enumerableMethodNames)
    {
        Console.Write(" {0},", method);
    }
}

// ...
```

输出15–8

```
Enumerable methods are: First, FirstOrDefault, Last, LastOrDefault,
Single, SingleOrDefault, ElementAt, ElementAtOrDefault, Repeat,
Empty, Any, All, Count, LongCount, Contains, Aggregate, Sum, Min, Max,
Average, Where, Select, SelectMany, Take, TakeWhile, Skip, SkipWhile,
Join, GroupJoin, OrderBy, OrderByDescending, ThenBy, ThenByDescending,
GroupBy, Concat, Distinct, Union, Intersect, Except, Reverse,
SequenceEqual, AsEnumerable, ToArray, ToList, ToDictionary, ToLookup,
DefaultIfEmpty, OfType, Cast, Range
```

在这个例子中，typeof(Enumerable).GetMembers()返回System.Linq.Enumerable的所有成员（方法、属性等）的一个列表。然而，许多成员都是重载的，有的甚至还不止一次重载。所以，为了避免同一个成员多次显示，我们在查询表达式中调用了Distinct()。这样就可以避免在列表中显示重复的名称。（typeof()和如GetMember()方法使用的反射的详情将在第17章讲述。）

15.2 查询表达式作为方法调用

令人惊讶的是，C# 3引入查询表达式并未对CLR或CIL语言进行任何改动。相反，是由C#编译器将查询表达式转换成一系列方法调用。以代码清单15-1的查询表达式为例，代码清单15-16部分重现了它。

代码清单15–16 简单的查询表达式

```
    private static void ShowContextualKeywords1()
    {
        IEnumerable<string> selection =
            from word in Keywords
                where word.Contains('*')
                select word;
        // ...
    }

// ...
```

编译之后，代码清单15-16的表达式会转换成一个由System.Linq.Enumerable提供的IEnumerable<T>扩展方法调用，如代码清单15-17所示。

代码清单15-17 查询表达式转换成标准查询操作符语法

```
    private static void ShowContextualKeywords3()
    {
        IEnumerable<string> selection =
            Keywords.Where(word => word.Contains('*'));

        // ...
    }

// ...
```

如第14章所述，Lambda表达式随后由编译器进行转换来生成一个带有Lambda的主体的方法。Lambda表达式的作用变成分配一个对该方法的委托。

每个查询表达式都能（而且必须能）转换成方法调用，但不是每一系列的方法调用都有对应的查询表达式。例如，扩展方法TakeWhile<T>(Func<T, bool> predicate)就没有与之等价的查询表达式。这个扩展方法只要谓词返回true，就反复返回集合中的项。

如果一个查询既有方法调用形式，也有查询表达式形式，那么哪个更好一些呢？这个问题没有固定答案，有的更适合使用查询表达式，有的使用方法调用反而更易读。

规范

要用查询表达式语法使查询更易读，尤其是涉及复杂的from、let、join或group子句时。

考虑在查询所涉及的操作没有查询表达式语法时，使用标准查询操作符（方法调用形式），例如Count()、TakeWhile()或者Distinct()。

15.3 小结

本章介绍了一种新的语法，即查询表达式。熟悉SQL的读者很快就会看出查询表达式和SQL共通的地方。然而，查询表达式还引入了一些附加的功能，比如分组成一套层次化的新对象集合，这在SQL中是不支持的。查询表达式的所有功能都可通过标准查询操作符来提供，但查询表达式往往提供了更简单的语法。然而，无论标准查询操作符还是查询表达式，结果都是显著改进了开发者编码集合API的方式。现在，熟悉面向对象编程的开发者可以更顺畅地同关系数据库打交道了。

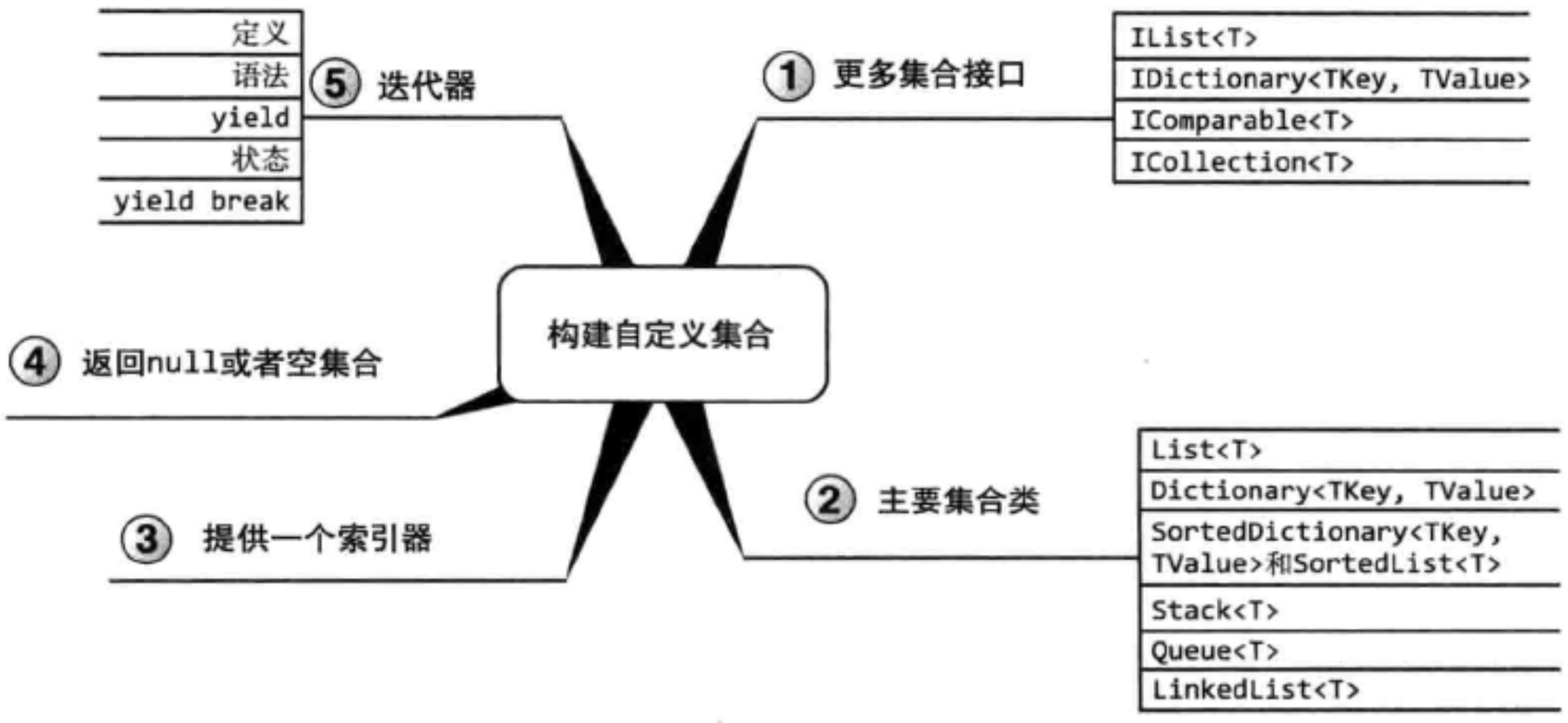
下一章将继续讨论集合。我们将具体讨论一些.NET Framework集合类型，并解释如何创建自定义集合。

第16章

构建自定义集合

第14章讨论了标准查询操作符，它们是由IEnumerable<T>提供的一套扩展方法，适合所有集合 。但是，这并不能使所有集合都适合所有任务。我们仍然需要不同的集合类型。有的集合适合根据键来搜索，有的则适合根据位置来访问。有的集合具有“先入先出”的队列行为，有的则更像“先入后出”的栈。还有一些根本没有顺序。

.NET Framework针对多种不同的使用场合定义了一系列集合。本章将介绍部分集合类型及其实现的接口。本章还介绍了如何创建自定义集合来支持标准集合功能（比如索引）。除此之外，本章还讨论了如何用yield return语句创建类和方法来实现IEnumerable<T>。利用这个C# 2.0引入的特性，可以简单地实现能由foreach语句遍历的集合。



.NET Framework中有许多非泛型集合类和接口，但它们主要是为了向后兼容。泛型集合类不仅更快（因为避免了装箱开销），还更加类型安全。所以，新代码应该总是使用泛型集合类。本书假定你主要使用泛型集合类型。

16.1 更多集合接口

前面讨论了集合如何实现IEnumerable<T>，这是实现集合元素遍历功能的主要接口。更复杂的集合实现了其他许多接口。图16-1展示了集合类实现的接口的层次结构。

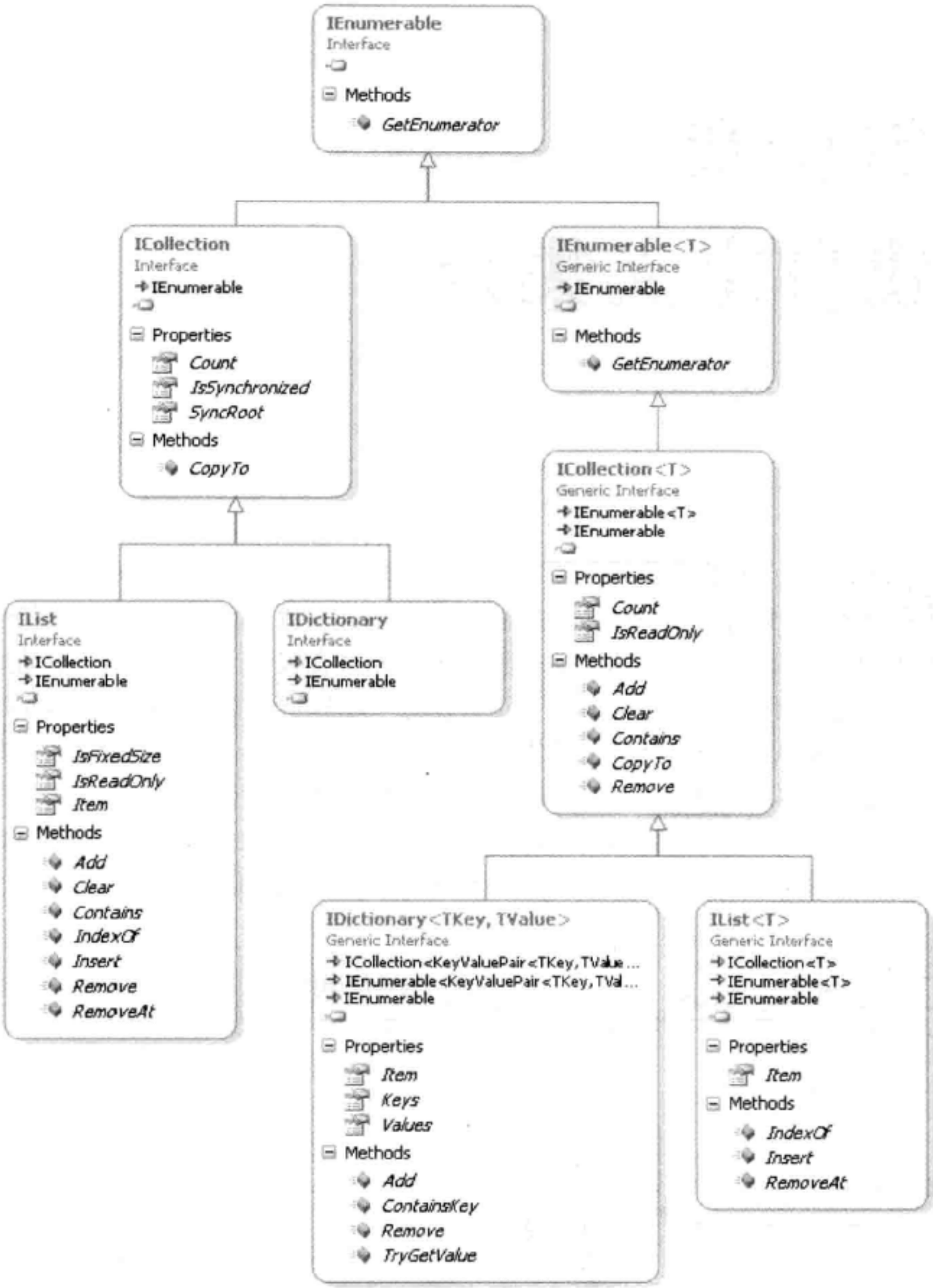


图16-1 泛型集合接口层次结构

这些接口提供了一种标准的方式来执行常规任务，包括遍历、索引和计数。本节将讨论这些接口（至少所有泛型版本），先从图16-1底部的接口开始，然后逐渐上移。

16.1.1 IList<T>与IDictionary<TKey, TValue>

可将英语字典看成是一个定义集合；查找“键”（被定义的单词））即可快速找到定

义。类似地，字典集合类是值的集合；每个值都可通过关联的、唯一的键来快速访问。但要注意，英语字典一般按照“键”的字母顺序存储定义。字典类也可选择这样，但一般都没有。最好将字典集合看成是键及其关联值的无序列表，除非文档专门指定了排序方式。类似地，一般不说查找“字典的第6个定义”；字典类一般只按照键进行索引，而不按位置索引。

列表则相反，它按特定顺序存储值并按位置访问它们。从某种意义上说，列表是字典的一个特例，其中“键”总是一个整数，“键集”总是从0开始的非负整数的一个连续集合。但是，由于两者存在重大差异，所以有必要用一个完全不同的类来表示列表。

所以，选择集合类来解决数据存储或者数据获取问题时，首先要考虑的两个接口就是IList<T>和IDictionary<TKey, TValue>。这两个接口决定了集合类型是侧重于通过位置索引来获取值，还是侧重于通过键来获取值。

实现这两个接口的类都必须提供索引器。对于IList<T>来说，索引器[①]的操作数是要获取的元素的位置：索引器获取一个整数，以便你访问列表中的第*n*个元素。对于IDictionary<TKey, TValue>来说，索引器的操作数是和一个值关联的键，以便你访问那个值。

16.1.2 ICollection<T>

IList<T>和IDictionary<TKey, TValue>都实现了ICollection<T>。此外，即使集合没有实现IList<T>或IDictionary<TKey, TValue>，也极有可能实现了ICollection<T>（但并非绝对，因为集合可以实现要求较少的IEnumerable或IEnumerable<T>）。ICollection<T>是从IEnumerable<T>派生的，包含两个成员：Count和CopyTo()。

- Count属性返回集合中的元素总数。从表面来看，只需要用一个for循环就可以遍历集合的每个元素。但是，要真正做到这一点，集合还必须支持按索引来获取（检索）值，而这个功能是ICollection<T>接口不包括的（虽然IList<T>包括）。
- CopyTo()方法允许将集合转换成数组。该方法包含一个index参数，允许指定在目标数组的什么位置插入元素。注意，要使用这个方法，必须初始化目标数组并使其具有足够大的容量：从index开始，一直到能装下ICollection<T>中的所有元素。

16.2 主要集合类

共有5类关键的集合类，它们的区别在于数据的插入、存储以及获取的方式。所有泛型类都位于System.Collections.Generic命名空间，等价的非泛型版本位于System.Collections命名空间。

16.2.1 列表集合：List<T>

List<T>类具有与数组相似的属性。关键区别是随着元素数量的增多，这种类会自动扩展（与之相反，数组的长度是固定的）。此外，列表可以通过显式调用TrimToSize()或Capacity来缩小（参见图16-2）。

---

① 索引器即索引操作符，即[]。——译者注

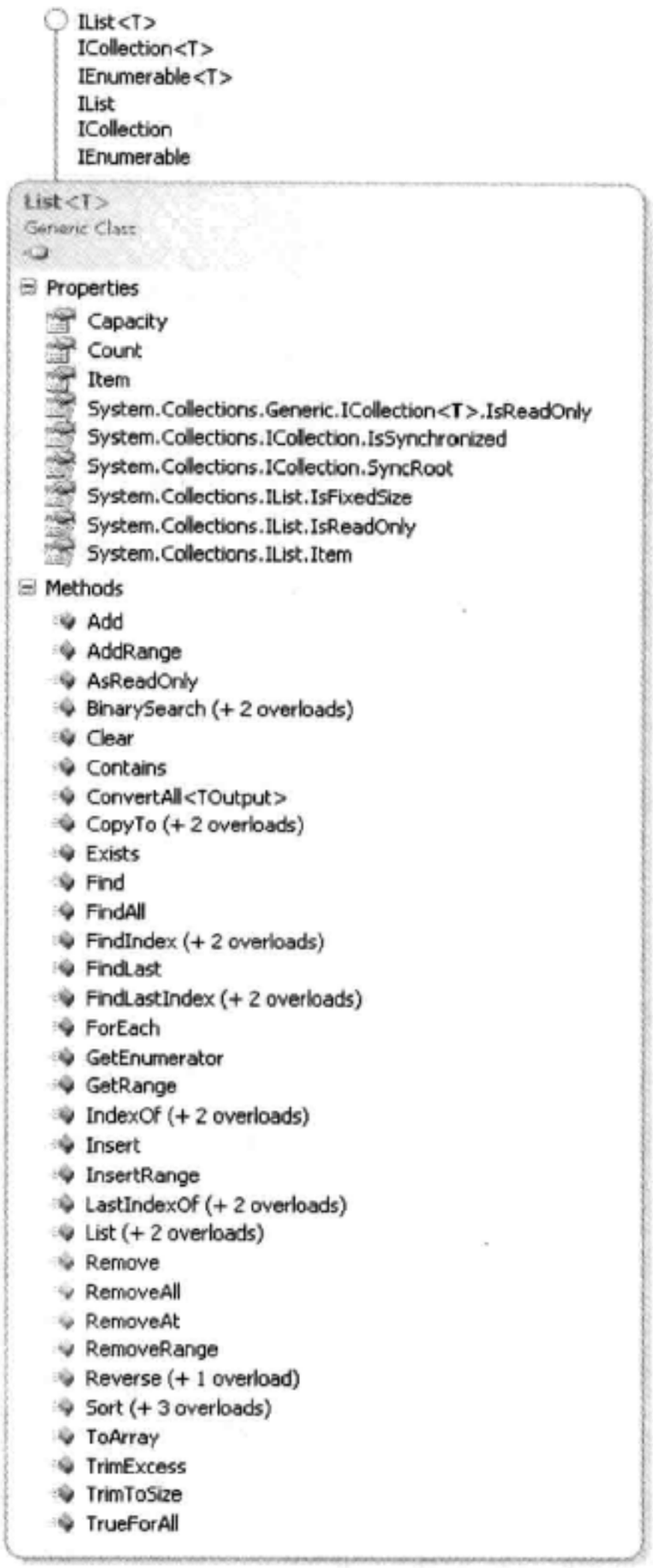


图16-2 List<>类图

这种类称为**列表集合**，其独特功能是每个元素都可根据索引来单独访问，这和数组一样。因此，可以使用索引器来设置和访问列表集合类中的元素，索引参数值对应于元素在集合中的位置。代码清单16-1展示了一个例子，输出16-1展示了结果。

代码清单16-1 使用List<T>

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
      List<string> list = new List<string>();

      // Lists automatically expand as elements
```

```
        // are added.
        list.Add("Sneezy");
        list.Add("Happy");
        list.Add("Dopey");
        list.Add("Doc");
        list.Add("Sleepy");
        list.Add("Bashful");
        list.Add("Grumpy");

        list.Sort();

        Console.WriteLine(
            "In alphabetical order {0} is the "
            + "first dwarf while {1} is the last.",
            list[0], list[6]);

        list.Remove("Grumpy");
    }
}
```

输出16-1

```
In alphabetical order Bashful is the first dwarf while Sneezy is the last.
```

C#的索引是基于零的，代码清单16-2中的索引0对应第一个元素，而索引6对应第七个元素。按索引获取元素不会造成搜索操作，而只需快速和简单地“跳”到一个内存位置。

List<T>是有序集合。Add()方法将指定的项添加到列表末尾。所以在代码清单16-1中，在调用Sort()之前，"Sneezy"是第一项，而"Grumpy"是最后一项。调用Sort()之后，列表按字母顺序排序，而不是按添加时的顺序。有的集合能在元素添加时自动排序，但List<T>不能；显式调用Sort()方能使元素得以排序。

移除元素使用的是Remove()或者RemoveAt()方法，它们用于移除指定元素，或者移除指定索引位置的元素。

高级主题：自定义集合排序

你可能好奇代码清单16-1的List<T>.Sort()方法是如何知道按字母顺序对列表元素进行排序的呢？string类型实现了IComparable<string>接口，它有一个CompareTo()方法。这个方法返回一个整数，指出所传递的元素是大于、小于还是等于当前元素。如果元素类型实现了泛型IComparable<T>接口或者非泛型IComparable接口，排序算法默认就用它来决定排序顺序。

但是，假如元素类型没有实现IComparable<T>，或者默认的比较逻辑不符合要求，又该怎么办呢？为了指定非默认的排序顺序，可以调用List<T>.Sort()的一个重载版本，它获取一个IComparer<T>作为实参。

IComparable<T>和IComparer<T>的区别很细微，但却很重要。前者说“我知道如何将我自己和我的类型的另一个实例进行比较”，后者说“我知道如何比较给定类型的两个实例”。

如果可以采取多种方式对一个数据类型进行排序，但没有哪一种占有绝对优势，就适合使用IComparer<T>接口。例如， Contact对象集合可能按姓名、地

点、生日、地区或者其他许多条件来排序。这时不该选择让一个Contact类来实现IComparable<Contact>，而应该创建多个不同的类来实现IComparer<Contact>。代码清单16-2展示了比较LastName和FirstName的一个示例实现。

代码清单16-2 实现IComparer<T>

```
class Contact
{
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public Contact(string firstName, string lastName)
  {
    this.FirstName = firstName;
    this.LastName = lastName;
  }
}
```

```
using System;
using System.Collections.Generic;

class NameComparison : IComparer<Contact>
{
  public int Compare(Contact x, Contact y)
  {
      if (Object.ReferenceEquals(x, y))
          return 0;
      if (x == null)
          return 1;
      if (y == null)
          return -1;
      int result = StringCompare(x.LastName, y.LastName);
      if (result == 0)
          result = StringCompare(x.FirstName, y.FirstName);
      return result;
  }

  private static int StringCompare(string x, string y)
  {
      if (Object.ReferenceEquals(x, y))
          return 0;
      if (x == null)
          return 1;
      if (y == null)
          return -1;
      return x.CompareTo(y);
    }
  }
```

若要先按LastName排序，再按FirstName排序，调用contactList.Sort(new NameComparer())即可。

16.2.2 全序

实现IComparable<T>或IComparer<T>时必须生成一个**全序**（total order）。CompareTo的实现必须为任何可能的数据项排列组合提供完全一致的排序结果。排序结果必须满足一些

基本条件。例如，每个元素都和它自己相等。如果元素X等于元素Y，元素Y等于元素Z，那么全部3个元素（X、Y、Z）都必须互等。如果元素X大于元素Y，那么元素Y必须小于元素X。而且不能存在“传递悖论”，即不能元素X大于元素Y，元素Y大于元素Z，但元素Z却大于元素X。如果不能提供全序的结果，排序算法的行为就是“未定义”的，可能会产生令人不解的排序结果，可能会崩溃，也可能会进入无限循环，等等。

例如，观察代码清单16-2的比较器（comparer）是如何确保全序的，它连实参是null的情况都考虑到了。不能“任何一个元素为null就返回零”，否则两个非null的元素可能等于null，但相互不等。

规范

要确保自定义比较逻辑产生一致的“全序”。

16.2.3 搜索List<T>

要在List<T>中查找特定的元素，可以使用Contains()、IndexOf()、LastIndexOf()和BinarySearch()方法。前三个除了LastIndexOf()从最后一个元素开始之外，其他方法都从第一个元素开始搜索，检查每个元素，直到发现目标元素。这些算法的执行时间与发现匹配项之前实际搜索的元素数量成正比。注意，集合类不要求集合中的所有元素都是唯一的。假如集合中有两个或者更多的元素相同，则IndexOf()返回的是第一个索引，LastIndexOf()返回的是最后一个索引。

BinarySearch()采用的是快得多的二分搜索算法，但它要求元素已经排好序。BinarySearch()方法的一个有用的功能是假如元素没有找到，会返回一个负整数。该值的按位取反（~）结果是“大于被查找元素的下一个元素”的索引，如果没有更大的值，则是元素的总数。这样一来，就可以在列表的特定位置方便地插入新值，同时还能保持已排序状态，如代码清单16-3所示。

代码清单16-3 对BinarySearch()的结果进行按位取反

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
    List<string> list = new List<string>();
    int search;

    list.Add("public");
    list.Add("protected");
    list.Add("private");

    list.Sort();

    search = list.BinarySearch("protected internal");
    if (search < 0)
```

```
        {
          list.Insert(~search, "protected internal");
        }

        foreach (string accessModifier in list)
        {
            Console.WriteLine(accessModifier);
        }
    }
}
```

要注意的是，假如列表事先没有排好序，那么不一定能找到指定的元素，即使它确实在列表中。代码清单16-3的结果如输出16-2所示。

输出16-2

```
private
protected
protected internal
public
```

高级主题：使用FindAll()查找多个数据项

有时必须在列表中找到多个项，而且搜索条件要比搜索一个特定的值复杂得多。为此，System.Collections.Generic.List<T>包含了一个FindAll()方法。FindAll()获取Predicate<T>类型的一个参数，它是对称为“委托”的一个方法的引用。代码清单16-4演示了如何使用FindAll()方法。

代码清单16-4 演示FindAll()及其谓词参数

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
      List<int> list = new List<int>();
      list.Add(1);
      list.Add(2);
      list.Add(3);
      list.Add(2);

      List<int> results = list.FindAll(Even);

      foreach(int number in results)
      {
          Console.WriteLine(number);
      }
  }

  public static bool Even(int value)
  {
      return (value % 2) == 0;
  }
```

```
    }
```

代码清单16-4在调用FindAll()时，传递了一个委托实例Even()。若整数实参值是偶数，该方法就返回true。FindAll()获取委托实例，并为列表中的每一项调用Even()（这个代码清单利用了委托类型推断）。每次返回值为true时，相应的项就被添加到一个新的List<T>实例中。检查完列表中的每一项之后，这个实例会被返回。第12章已详细讨论了委托。

16.2.4 字典集合：Dictionary<TKey, TValue>

另一种集合类是字典类，具体就是Dictionary<Tkey, Tvalue>（参见图16-3）。和列表集合不同，字典类存储的是“名称/值”对。其中，名称相当于独一无二的键，可以利用它像在数据库中利用主键来访问一条记录那样查找对应的元素。这会为访问字典元素带来一定的复杂性，但由于利用键来进行查找效率非常高，所以这是一个有用的集合。注意，键可为任意数据类型，而非仅能为字符串或数值。

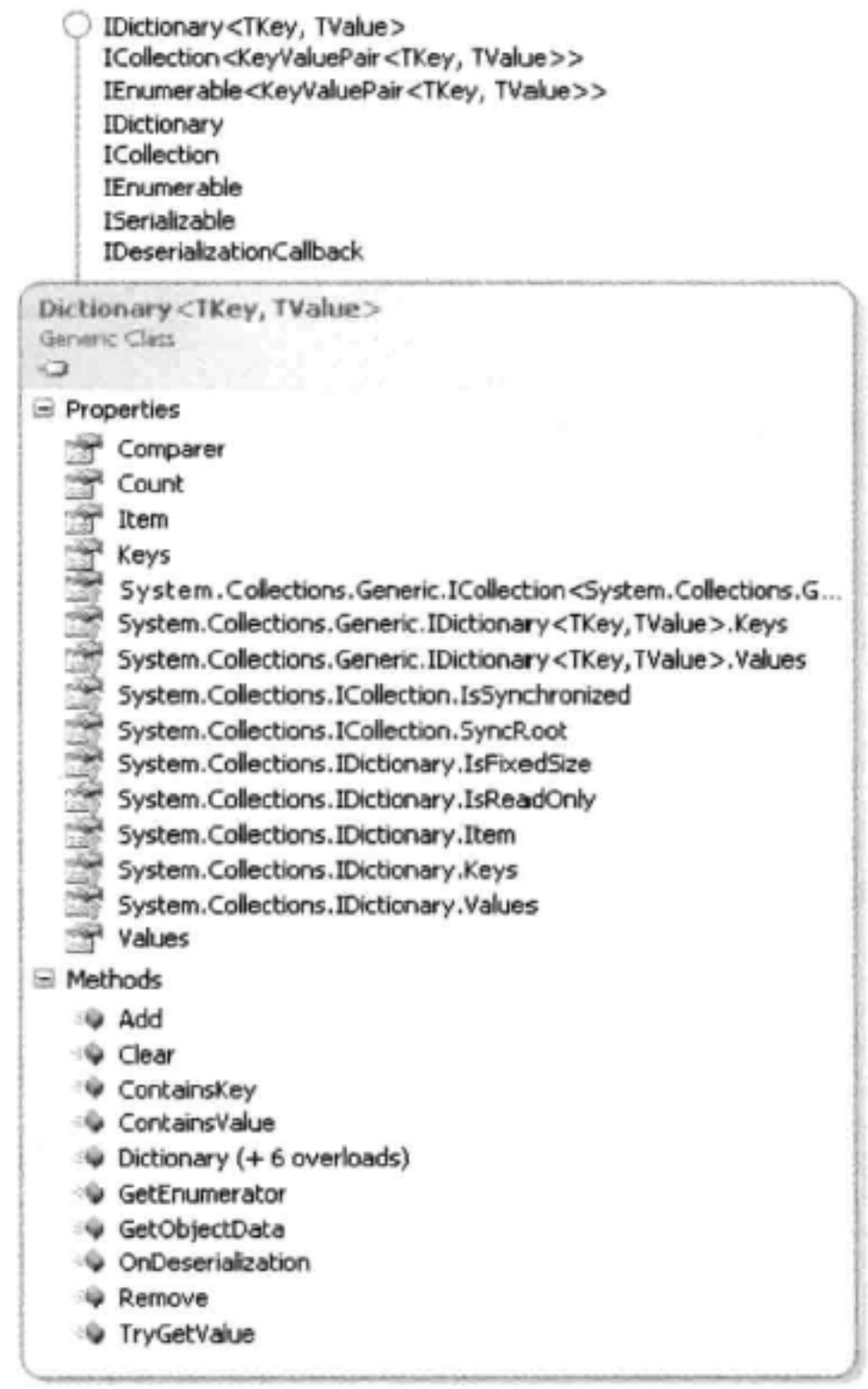


图16-3 Dictionary类图

要在字典中插入元素，一个选择是使用Add()方法，向它传递键和值，如代码清单16-5所示。

代码清单16-5 向Dictionary<TKey, TValue>添加项

```
using System;
using System.Collections.Generic;
```

```
class Program
{
  static void Main()
  {
      Dictionary<Guid,string> dictionary =
          new Dictionary<Guid, string>();
      Guid key = Guid.NewGuid();

      dictionary.Add(key, "hello");
  }
```

代码清单16-5使用一个Guid作为键来插入字符串"hello"。假如已添加了具有相同键的元素，就会引发异常。

还有一个选择是使用索引器，如代码清单16-6所示。

代码清单16-6 使用索引操作符在Dictionary<TKey, TValue>中插入项

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
      Dictionary<Guid, string> dictionary =
          new Dictionary<Guid, string>();
      Guid key = Guid.NewGuid();

      dictionary[key] = "hello";
      dictionary[key] = "goodbye";
  }
}
```

在代码清单16-6中，要注意的第一件事情是索引操作符不要求整数。相反，索引操作数的类型是由第一个类型实参（Guid）指定的。索引器设置或获取的值的类型由第二个类型实参（string）指定。

在代码清单16-6中，要注意的第二件事情是同一个键使用了两次。第一次赋值时，没有与指定的键对应的字典值，所以字典集合类插入具有指定键的新值。第二次赋值时，具有指定键的元素已经存在，所以此时不是插入新元素，而是删除值"hello"，将值"goodbye"与键关联。

如果试图从字典读取键不存在的值，就会引发KeyNotFoundException异常。利用ContainsKey()方法，可以在访问与一个键对应的值之前检查这个键是否存在，从而避免引发异常。

Dictionary<TKey, TValue>是作为“散列表”实现的；这种数据结构在根据键来查找值时速度非常快——无论字典中存储了多少值。相反，检查特定值是否在字典集合中相当花时间，性能和搜索无序列表一样是“线性”的。该操作使用的是ContainsValue()方法，它顺序搜索集合中的每个元素。

移除字典元素是用Remove()方法，向它传递键，而不是元素的值。

由于键和值都要添加到字典中，所以用于枚举字典中的元素的foreach循环的循环变量必须是KeyValuePair<TKey, TValue>。代码清单16-7演示了如何用foreach循环来枚举字典中的键和值，输出16-3展示了结果。

代码清单16-7 使用foreach遍历Dictionary<TKey, TValue>

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
      var dictionary = newDictionary<string,string>();
      dictionary.Add("do", "a deer");
      dictionary.Add("re", "a drop");
      dictionary.Add("mi", "a name");
      dictionary.Add("fa", "a long way");
      dictionary.Add("so", "a needle");
      dictionary.Add("la", "a note");
      dictionary.Add("ti", "a drink");
      Console.WriteLine("Key  Value   ");
      Console.WriteLine("---  ------- ");
      foreach (KeyValuePair<string, string> i in dictionary)
      {
          Console.WriteLine("{0}   {1}", i.Key, i.Value);
      }
  }
}
```

输出16-3

```
Key  Value
---  -------
do   a deer
re   a drop
mi   a name
fa   a long way
so   a needle
la   a note
ti   a drink
```

注意，此处显示的数据项的顺序是它们添加到字典的顺序，这和添加到列表是一样的。字典的实现通常按照添加时的顺序枚举键和值，但这既不是必须的，也没有编入文档，所以不要依赖它。

规范

不要对集合元素的顺序进行任何假定。如果集合的说明文档没有指明它是按特定顺序枚举的，就不能保证以任何特定顺序生成元素。

如果只想处理字典类中的键或值，那么可以利用Keys和Values属性。这些属性返回的数据类型是ICollection<T>。它们返回的是对原始字典集合中的数据的引用，而不是返回副本。字典中的变化会在Keys和Values属性返回的集合中自动反映出来。

高级主题：自定义的字典相等性

为了判断一个指定的键是否与字典中现有的键匹配，字典必须能比较两个键的相等性。这就像列表必须能比较两个项来判断其顺序（参见本章前面的“高级主题：自定义的集合排序”）。对值类型的两个实例进行比较，默认是检查两者是否包含了一样的数据。对于引用类型的实例的比较，则是检查是否引用了同一个对象。然而，即使两个实例不包含相同的值，或者不引用同一个对象，偶尔也需要比较它们是否相等。

例如，你可能想用代码清单16-2的Contact类型创建一个Dictionary<Contact, string>，并希望假如两个Contact对象具有相同的FirstName和LastName，就判定两者相等，而无论两个对象是否引用相等。与之前通过实现IComparer<T>对列表进行排序类似，可以提供IEqualityComparer<T>的一个实现来判断两个键是否相等。这个接口要求两个方法：一个方法返回bool值以指出两个项是否相等，一个方法返回一个“散列码”来实现字典的快速索引。代码清单16-8展示了一个例子。

代码清单16-8 实现IEqualityComparer<T>

```
using System;
using System.Collections.Generic;

class ContactEquality : IEqualityComparer<Contact>
{
  public bool Equals(Contact x, Contact y)
  {
      if (Object.ReferenceEquals(x, y))
          return true;
      if (x == null || y == null)
          return false;
      return x.LastName == y.LastName &&
          x.FirstName == y.FirstName;
  }

  public int GetHashCode(Contact x)
  {
      if (Object.ReferenceEquals(x, null))
          return 0;
      int h1 = x.FirstName == null ? 0 : x.FirstName.GetHashCode();
      int h2 = x.LastName == null ? 0 : x.LastName.GetHashCode();
      return h1 * 23 + h2;
  }
}
```

你可以使用构造器new Dictionary<Contact, string>(new ContactEquality)来创建一个利用了这个相等性比较器的字典。

初学者主题：相等性比较的要求

如第9章所述，相等性和散列码算法有几条重要的规则。集合更需要遵守这些规则。就像列表排序要求自定义的排序器提供“全序”，散列表也对自定义的相等性比较提出了一些要求。其中最重要的要求就是，如果Equals()对于两个对象返回的都是true，那么GetHashCode()对这两个对象的返回值也必须是true。但反之则不然：两个不相等的项可能具有相同的散列码。（事实上必然有两个不相等的项具有相同的散列码，因为总共只有232个可能的散列码，不相等的对象数量比这多！）

第二个重要的要求是，至少当数据项在散列表中的时候，对同一个数据项的多次GetHashCode()调用都必须生成相同的结果。但要注意的是，在程序两次运行期间，两个“看起来相等”的对象并不一定产生相同的散列码。例如，可以今天为指定联系人分配一个散列码，两周后当程序再次运行时，为该联系人分配不同的散列码，这是完全合法的。不要将散列码持久存储在数据库中，并指望每次运行程序时的散列码都保持不变。

在理想情况下，GetHashCode()的结果应该是“随机”的。也就是说，输入的小变化应造成输出的大变化，得到的结果应该在所有可能的整数值之间大致均匀地分布。但是，很难设计出既非常快又产生非常良好分布的输出；尽量尝试在两者之间取得平衡。

最后，GetHashCode()和Equals()千万不能引发异常；例如，代码清单16-8就非常小心地避免了对空引用进行解引用。

下面总结了一些关键原则。

- 相等的对象必然有相等的散列码。
- 实例生存期间（至少当其在散列表中的时候），其散列码应一直不变。
- 散列算法应快速生成良好分布的散列码。
- 散列算法应避免在所有可能的对象状态中引发异常。

16.2.5 已排序集合：SortedDictionary<TKey, TValue>和SortedList<T>

已排序集合类（参见图16-4）的元素是已经排好序的。具体地说，对于SortedDictionary<TKey, TValue>，元素是按照键排序的；对于SortedList<T>，元素则是按照值排序的（还有一个非泛型的SortedList实现）。更改代码清单16-7的代码来使用SortedDictionary<string, string>而不是Dictionary<string, string>，结果如输出16-4所示。

输出16-4

```
Key  Value
---  -------
do   a deer
fa   a long way
la   a note
mi   a name
re   a drop
so   a needle
ti   a drink
```

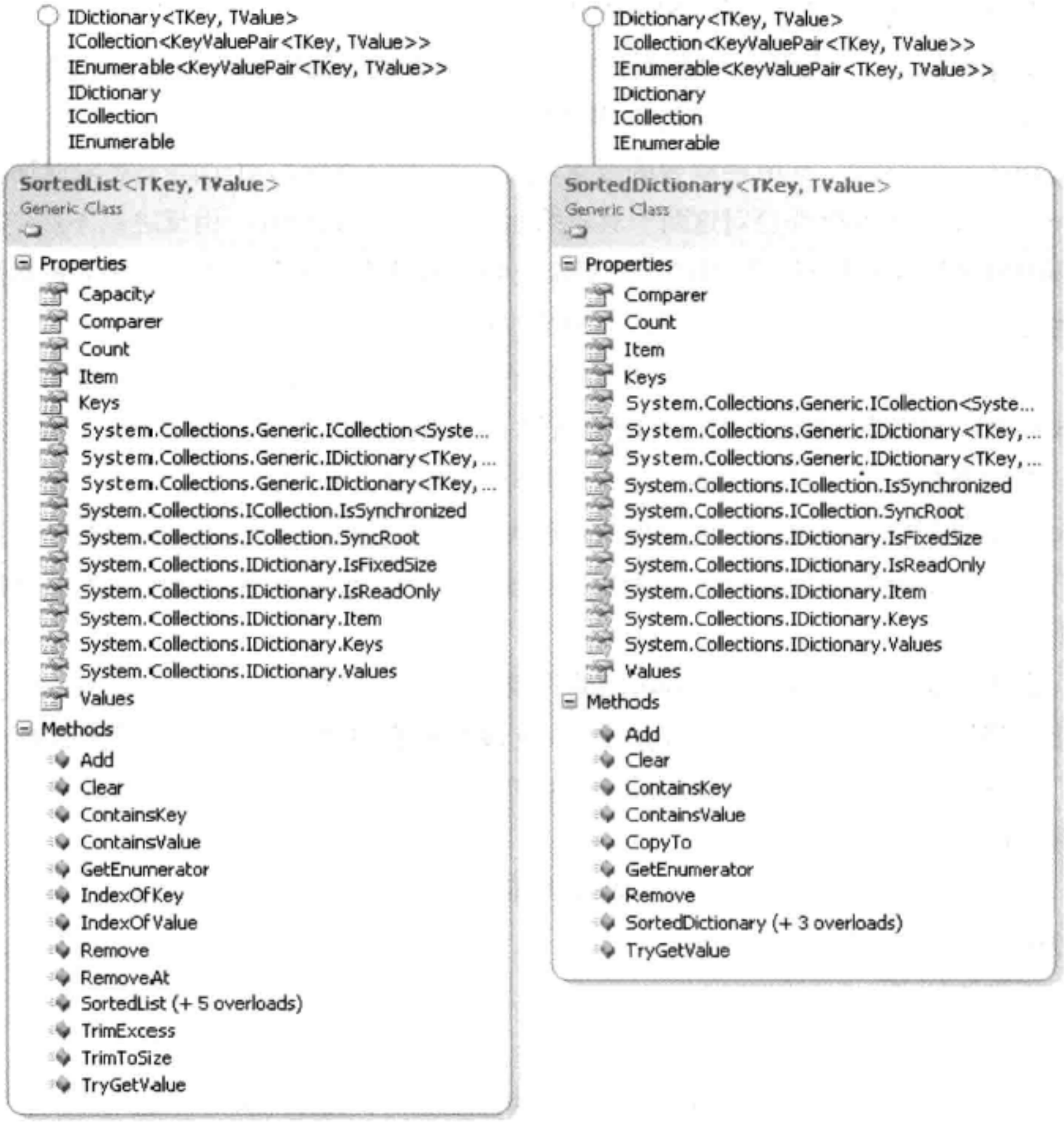


图16-4 SortedList<>和SortedDictionary<>类图

注意，元素是根据键来排序的，而不是根据值来排序。

由于要保持集合中元素的顺序，所以相对于无序字典来说，在一个已排序集合中插入和删除元素要稍慢一些。

由于已排序集合必须按特定顺序存储数据项，所以既可按键访问值，也可按索引访问值。要根据索引访问键或者值，请使用 Keys和Values属性。 它们分别返回IList<TKey>和IList<TValue>实例；结果集合可像其他列表那样索引。

16.2.6 栈集合：Stack<T>

第11章讨论了栈集合类（如图16-5所示）。栈集合类被设计成后入先出集合。两个关键的方法是Push()和Pop()。

- Push()将元素送入集合。元素不必是唯一的。
- Pop()按照与添加时相反的顺序获取并删除元素。



图16-5 Stack<T>类图

为了在不修改栈的前提下访问栈中的元素，要使用Peek()和Contains()方法。Peek()方法返回Pop()将获取的下一个元素。

和大多数集合类一样，Contains()方法用于判断一个元素是否存在于栈的某个地方。和所有集合一样，还可以用foreach循环来遍历栈中的元素。这样就可以访问栈中任何地方的值。但要注意，用foreach循环来访问值不会把它从栈中删除。只有Pop()才会。

16.2.7 队列集合：Queue<T>

如图16-6所示，队列集合类与栈集合类基本相同，只是遵循先入先出排序模式。代替Pop()和Push()的分别是Enqueue()和Dequeue()方法，前者称为入队方法，后者称为出队方法。队列集合的行为像是一根“管子”。Enqueue()方法在队列的一端将对象放入队列，Dequeue()方法从另一端移除它们。和栈集合类一样，对象不必是唯一的。另外，队列集合类会根据需要自动增大。队列缩小时不一定回收之前使用的存储空间，因为这会使插入新元素的动作变得很昂贵。如果确定队列将长时间大小不变，可以使用TrimToSize()方法提醒队列集合你希望回收存储空间。



图16-6 Queue<T>类图