第3章 集合与泛型

.NET 下提供了包含数组在内的多种集合类型，为应用程序中建立各种复杂数据结构提供了基础性的支持。一般情况下，程序员只要在.NET 现有的各种集合类型中作出选择，就可以满足大部分应用中的需求。特殊情形下，还可以利用.NET 中有关的基类派生出自定义集合类型。

3.1 数组和数组列表

数组是一组相同类型的变量的集合，其中的每个变量称为该数组的元素。一个数组中的所有元素位于一个连续的内存块中，并且可以通过使用整数索引来访问它们。当程序需要处理一组有序的相似元素时，数组就显得非常有用。在这种情况下，与创建许多独立的变量相比，使用数组更为灵活且有效。C＃中的数组可分为一维数组和多维数组。只包含单个序列的数组称为一维数组，而多维数组需要用一个或多个整数值对其进行索引。

在.NET 中通过 System.Array 类来支持数组。System.Array 类是一个抽象类。它提供了一些用来操作数组的通用方法。表 3-1 列举了其中的一部分。

表 3-1 System.Array 类中的常用方法

名　　称	描　　述
Sort	对数组元素进行排序
Clear	将一系列元素设置为空引用或零(取决于元素类型)
GetLength	返回数组指定维数的长度
IndexOf	返回指定值在数组中首次出现的索引位置
Clone	创建数组的副本
CopyTo	将当前一维数组的所有元素复制到指定的一维数组中

C＃下常用的数组声明的语法为：

```
数组类型[] 数组名;
```

例如：

```
int[] arr;
```

.NET 的数组属于引用类型，因此需要为数组变量创建实例。C＃创建数组实例的语法为：

```
数组名 = new 数组类型[数组长度];
```

这里数组的长度必须是预先设定的，往往不够灵活。如果需要可变长度的数组，则可以用数组列表。数组列表属于在 System.Collection 命名空间中定义的 ArrayList 类的对象，在用法上和数组非常类似，但是其容量可以按照需要动态增长。当把元素添加到一个 ArrayList 列表时，该列表的容量会自动增加。

可以像访问数组那样使用索引来访问 ArrayList 中的元素。通过用 Add 或 Remove 方法可以向 ArrayList 中添加元素或者从其中移除元素。需要特别注意的是，所有的 ArrayList 元素都是 System.Object 类型的。因此，当从 ArrayList 中访问元素时，往往需要执行类型转换。

表 3-2 列出了 ArrayList 类的一些常用方法和属性。

表 3-2　ArrayList 类的常用方法和属性

	名　　称	描　　述
属性	Count	该属性值代表 ArrayList 中实际包含的元素数
	Item	获取或设置指定索引处的元素
方法	Add	将元素添加到 ArrayList 的末端
	Clear	清空 ArrayList 中的所有元素
	IndexOf	返回 ArrayList 或它的一部分中某个值的第一个匹配项的从零开始的索引
	Insert	将元素插入 ArrayList 的指定索引处
	Remove	从 ArrayList 中移除指定元素的第一个匹配项
	RemoveAt	在 ArrayList 中移除指定索引处的元素
	RemoveRange	在 ArrayList 中移除从指定索引处开始的若干个元素
	Reverse	将 ArrayList 元素的顺序逆置
	Sort	对 ArrayList 中的元素执行排序
	TrimToSize	将容量设置为 ArrayList 元素实际数量

下面是一个简单的例子。

【例 3-1】 本例示范如何使用 ArrayList 的一些基本操作。

```
using System;
using System.Collections;
public class SamplesArrayList {
   public static void Main(){
      ArrayList myAL = new ArrayList(); //创建一个 ArrayList 列表的实例
      myAL.Add("The");                  //从本行开始连续在该列表内添加若干元素
      myAL.Add("quick");
      myAL.Add("brown");
      myAL.Add("fox");
      myAL.Add("jumped");
      myAL.Add("over");
      myAL.Add("the");
      myAL.Add("lazy");
      myAL.Add("dog");
      Console.WriteLine("The ArrayList initially contains the following:");
      PrintValues(myAL);                //显示该列表
      myAL.Remove("lazy");              //删除列表中包含"lazy"的元素
      Console.WriteLine("After removing \"lazy\":");
```

```
        PrintValues(myAL);                    //显示该列表当前状态
        myAL.RemoveAt(5);                     //删除列表中索引为 5 的元素
        Console.WriteLine("After removing the element at index 5:");
        PrintValues(myAL);                    //显示该列表当前状态
        myAL.RemoveRange(4,3);                //从索引为 4 的位置开始删除连续 3 个元素
        Console.WriteLine("After removing three elements starting at index 4:");
        PrintValues(myAL);                    //显示该列表最终的状态
    }

    public static void PrintValues(IEnumerable myList)                    //显示列表中所有元素
    {
        foreach (Object obj in myList)    //foreach 循环访问列表中每一个元素
            Console.Write( " {0}",obj );
        Console.WriteLine( );
    }
}
```

如前所述，ArrayList 是可以像数组那样通过索引进行访问的，因此在本例的 PrintValues 方法中，也可以用

```
for(int i = 0; i < myAL.Count; i++)
    Console.Write(" {0}",myAL[i]);
```

代替 foreach 语句对列表进行遍历访问并输出。

3.2 队　　列

队列(Queue)类实现了一种“先进先出”的数据结构，用于描述需要顺序处理的一组对象。队列可以按照对象插入的顺序进行存储，并且总是在队列的尾端插入对象，在队列的首端移除对象。

表 3-3 列出了 Queue 类的一些常用属性和方法。

表 3-3　Queue 类的一些常用属性和方法

	名　　称	描　　述
属性	Count	指示 Queue 中包含的元素数
方法	Clear	清空队列中所有的元素
	Contains	如果队列中存在某个指定元素返回 true，否则返回 false
	Dequeue	移除并返回队列首端的元素
	Enqueue	将一个新元素添加到队列的尾端
	Peek	返回队列首端的元素，但是并不移除它
	ToArray	将 Queue 元素复制到新数组

这里需要注意的是，在空的队列上调用 Dequeue 或 Peek 方法，将引发 InvalidOperationException 异常。

一个队列中的所有元素存储在一个缓冲区中，这个缓冲区可以按照需要扩展，以便为附加的对象提供空间。如果队列缓冲区不需要扩展，那么在相应队列上执行操作的开销就相

对较小。调整队列的大小是需要付出一定性能代价的，所以应尽量避免频繁调整缓冲区的大小。C#中默认的缓冲区大小为32。即队列中最多容纳32个元素。可以通过队列构造函数来设定缓冲区的大小，例如：

```
Queue q = new Queue (100);
```

在队列对象的构造过程中，还可以设置队列对象的增长因子。当缓冲区大小需要调整时，可以使用增长因子来设定新建的队列缓冲区的大小。增长因子默认为2.0，即每当队列缓冲区必须增长时，它的大小将增长到原来的2.0倍。Queue类有一个允许同时初始化队列大小和增长因子的构造函数。例如：

```
Queue q = new Queue(100,10.0);
```

通过设置大于默认值的增长因子，可以减少Queue对象调整大小的次数。

下面给出使用队列类的示例。

【例3-2】 在本例中调用了队列元素的入列(即Enqueue)和出列(即Dequeue)以及Peek等方法，从中可了解它们的用法。

```
using System;
using System.Collections;
public class SamplesQueue {
  public static void Main(){
    Queue myQ = new Queue();                                    //创建队列对象的实例
    myQ.Enqueue( "测" );
    myQ.Enqueue( "试" );
    myQ.Enqueue( "成" );
    myQ.Enqueue( "功" );
    Console.Write( "Queue values:" );
    ShowValues( myQ,'\t');                                      //显示当前队列中的元素
    Console.WriteLine( "(Dequeue)\t{0}",myQ.Dequeue() );        //执行出列
    Console.Write( "Queue values:" );
    ShowValues( myQ,'\t');                                      //显示当前队列中的元素
    Console.WriteLine( "(Dequeue)\t{0}",myQ.Dequeue() );        //执行出列
    Console.Write( "Queue values:" );
    ShowValues( myQ,'\t');                                      //显示当前队列中的元素
    Console.WriteLine( "(Peek) \t{0}",myQ.Peek() );             //Peek时无须出列
    Console.Write( "Queue values:" );
    ShowValues( myQ,'\t');                                      //显示当前队列中的元素
  }
  public static void ShowValues( Queue q,char mySeparator )     //显示队列中的元素
  {
       IEnumerator myEnumerator = q.GetEnumerator();
       while (myEnumerator.MoveNext())
         Console.Write("{0}{1}",mySeparator,myEnumerator.Current);
       Console.WriteLine( );
    }
}
```

本例运行输出的结果为：

```
Queue values: 测  试  成功
(Dequeue)     测
Queue values: 试  成  功
(Dequeue)     试
Queue values: 成  功
(Peek)        成
Queue values: 成  功
```

本例的 ShowValues 方法采用了不同于例 3-1 中的方式对集合中的元素进行遍历访问，这样做的目的是为了更全面地揭示. NET 下与集合类型相关的一些特性：System. Collections. Queue类实现了三个与集合相关的接口：ICollection、IEnumerable、ICloneable。因此可通过 GetEnumerator 方法得到 IEnumerator 的实例，然后只要连续执行 MoveNext 方法就可以遍历整个集合。就本例来说，也可以用

```
foreach (object s in q )
    Console.Write("{0}\t ",(string) s);
```

来遍历该队列，因为 C# 中的 foreach 一般就是转换为利用 IEnumerator 接口实现遍历的。此外，ArrayList 也是实现了 IEnumerable 接口的，因此可以像本例中那样对数组列表进行遍历。顺便提一下，队列的元素是不能像数组中那样使用索引来访问的。

表 3-4 给出了 IEnumerator 接口中主要成员的说明，供读者参考。

表 3-4　IEnumerator 接口的主要成员

	名　　称	描　　述
属性	Current	获取集合中的当前元素
方法	MoveNext	将当前位置推进到集合的下一个元素
	Reset	重新开始一轮遍历，当前元素设置为初始位置

3.3　栈

栈(Stack)类实现了一种“后进先出”的数据结构，在这种数据结构中，最后入栈的元素位于栈的顶部。和 Queue 类一样，栈类也实现了 ICollection、IEnumerable、ICloneable 这三个接口。表 3-5 列出了 Stack 类的一些常用属性和方法。

表 3-5　Stack 类的一些常用属性和方法

	名　　称	描　　述
属性	Count	获取 Stack 中包含的元素数
方法	Clear	清空栈中所有的对象
	Contains	如果栈中存在某个指定对象返回 true，否则返回 false
	Peek	返回栈顶部的元素，但是并不从栈中移除该元素
	Pop	移除并返回栈顶部的元素
	Push	将新元素插入到栈的顶部
	ToArray	将 Stack 复制到新数组中

下面给出一个如何使用 Stack 类的示例。

【例 3-3】 在本例中给出了堆栈类的压入和弹出元素的方法调用。

```
using System;
using System.Collections;
class Test {
    static void Main(){
        Stack s = new Stack();                                  //创建一个栈
            //以下连续对该栈压入 6 个元素
        s.Push(1);
        s.Push(2);
        s.Push(3);
        s.Push(4);
        s.Push(5);
        s.Push(6);
        Console.WriteLine("栈中的元素有: ");
        foreach(int index in s){
            Console.WriteLine("{0}",index);
        }
        while(s.Count > 0) {
            int tmp = (int) s.Pop();                            //逐个元素出栈
            Console.WriteLine(" 弹出栈元素: {0}",tmp);
        }
        Console.WriteLine("栈目前为空");
    }
}
```

本例运行时输出结果如下。

```
堆栈中的元素有:
6
5
4
3
2
1
弹出栈元素: 6
弹出栈元素: 5
弹出栈元素: 4
弹出栈元素: 3
弹出栈元素: 2
弹出栈元素: 1
栈目前为空
```

3.4 哈希表和有序表

哈希表(Hashtable)是一种很有用但也是相对比较复杂的数据结构,在哈希表集合中的每一个元素中都是以键、值对的形式保存数据的。其中的“键”和“值”都是字符串,这些数据

保存在内存中。由于使用独特的地址算法，对其插入、查找和其他基本操作的速度都很快，在数据量较大的情形下这一点更明显。哈希表的查找方式主要为通过已知的键找出其对应值。

.NET 的 Hashtable 实现了哈希表。表 3-6 为该类的主要属性和方法。

表 3-6　Hashtable 类的主要属性和方法

	名　　称	描　　述
属性	Count	指示 Hashtable 中包含的元素的个数
	Item	表示与指定的键相关联的值
	Keys	表示此 Hashtable 中的键的 ICollection
	Values	表示此 Hashtable 中的值的 ICollection
方法	Add	将带有指定键和值的元素添加到 Hashtable 中，Add 的参数 Key 和 Value 被定义为 Object 类型的，但一般使用时用字符串作为 Key 和 Value 的值
	Clear	从 Hashtable 中移除所有元素
	ContainsKey	确定 Hashtable 是否包含特定键
	ContainsValue	确定 Hashtable 是否包含特定值
	Remove	从 Hashtable 中移除带有指定键的元素

下面是一个使用 Hashtable 类的简单示例。

【例 3-4】 本例中可看到如何在.NET 程序中对哈希表进行存储和检索，其源代码如下：

```
using System;
using System.Collections;
class Program
{
      static void Main(string[] args){
         Hashtable currencies = new Hashtable();              //新建一个哈希表
         currencies.Add("US","Dollar");                       //添加一个表元素(键值对)
         currencies.Add("Japan","Yen");
         currencies.Add("France","Euro");
         Console.Write("US Currency: {0}",currencies["US"]);
            //检索已知键为"US"的哈希值并显示
         Console.ReadLine();
      }
}
```

本例运行输出的结果为：

```
US Currency: Dollar
```

与哈希表类似，.NET 下还有一种 SortedList 的集合类型，它的元素也是"键"、"值"对。SortedList 的"键"、"值"对可按键排序并可按照键或索引进行访问。SortedList 表由于是排好序的，因此查找速度可以相当快。但对其插入操作时一般需要大量移动元素，因此比较慢。

SortedList 的属性和方法与 Hashtable 的相近，这里不再予以列出。下面给出一个示例。

【例 3-5】 本例介绍 SortedList 类对象的基本用法，其源代码如下：

```
using System;
using System.Collections;
class Program
{
    static void Main(string[] args){
        SortedList slColors = new SortedList();
        slColors.Add("forecolor","black");
        slColors.Add("backcolor","white");
        slColors.Add("errorcolor","red");
        slColors.Add("infocolor","blue");
        foreach (DictionaryEntry de in slColors){
            Console.WriteLine("{0} = {1}",de.Key,de.Value);
        }
        Console.WriteLine("Press Enter to exit...");
        Console.ReadLine();
    }
}
```

本例运行输出的结果为：

```
backcolor = white
errorcolor = red
forecolor = black
infocolor = blue
Press Enter to exit...
```

从输出中可以看到该表已按照键值排过序。

注：.NET 中 DictionaryEntry 类的对象用于表示一个“键”、“值”对。因此 Hashtable 和 SortedList 都是 DictionaryEntry 类的对象的集合。

3.5 专 用 集 合

.NET 中还定义了一些特定类型的集合，如 StringCollection 等。这种类型在对特定类型的对象操作时往往有更好的效率。下面是一个例子。

【例 3-6】 本例介绍 StringCollection 类的用法，其源代码如下：

```
using System;
using System.Collections;
using System.Collections.Specialized;
  //System.Collections.Specialized 命名空间中包含这些特定类型的集合
public class Program
{
    public static void Main(){
       StringCollection sc = new StringCollection();
       sc.Add("first");
       sc.Add("second");
       sc.Add("third");
```

```
        for (int i = 0; i < sc.Count; i++){
            Console.WriteLine(sc[i]);                    //可对该集合类型使用索引
         }
         Console.WriteLine("Press Enter to exit...");
         Console.ReadLine();
    }
}
```

3.6 使 用 泛 型

泛型是.NET 中特有的类型,定义泛型时用 list<…>语句表示。它很像 ArrayList 类,但可以在创建对象实例时限定列表中元素的类型,这样在使用时更方便和高效。程序中使用泛型需要在代码文件中添加对 System.Collections.Generic 命名空间的引用。

以下为一个使用泛型的示例。

【例 3-7】 本例中使用的 list<string>泛型是一个由 string 元素构成的列表,其源代码如下:

```
using System;
using System.Collections;
using System.Collections.Generic;
public class Program
{
    public static void Main( ){
       List < string > names = new List < string >();
       names.Add("Michael Patten");
       names.Add("Simon Pearson");
       names.Add("David Pelton");
       names.Add("Thomas Andersen");
       foreach(string str in names){              //可以用 foreach 对泛型进行遍历
          Console.WriteLine(str);
       }
       names.Remove("David Pelton");
       for(int i = 0;i < names.Count;i++)         //也可以用索引对泛型元素进行访问
          Console.WriteLine(names[i]);
       Console.ReadLine();
    }
}
```

本例运行输出的结果为:

```
Michael Patten
Simon Pearson
David Pelton
Thomas Andersen
Michael Patten
Simon Pearson
Thomas Andersen
```

从中看出,该泛型的用法与 ArrayList 基本是一样的。为了验证在泛型定义时< >中

可以输入不同的类型，下面再看一个元素为整型的泛型的示例。

【例 3-8】 本例中使用的 List＜int＞泛型是一个由 int 元素构成的列表，其源代码如下：

```
using System;
using System.Collections;
using System.Collections.Generic;
public class Program
{
    public static void Main( ){
       List < int > numbers = new List < int >();
       numbers.Add(3);
       numbers.Add(6);
       numbers.Add(4);
       numbers.Add(2);
       foreach(int k in numbers){
          Console.WriteLine(k);
       }
       numbers.Remove(3);
       for(int i = 0; i < numbers.Count; i++){
          Console.WriteLine(numbers [i]);
       }
       Console.ReadLine();
    }
}
```

.NET 中还有针对 Stack 和 Queue 等类的泛型，它们用 Stack＜…＞和 Queue＜…＞等表示，用法与 Stack 和 Queue 等类相似，但同样可以限定其元素的类型。

请看下面的示例。

【例 3-9】 本例中使用的 Stack＜int＞泛型，是一个由 int 元素构成的栈(Stack)，其源代码如下：

```
using System;
using System.Collections;
using System.Collections.Generic;                    //添加该命名空间
public class Program
{
    public static void Main(){
       Stack < int > numbers = new Stack < int >();
       numbers.Push(3);
       numbers.Push(6);
       numbers.Push(4);
       numbers.Push(2);
       int count = numbers.Count;
       for (int i = 0; i < count; i++){
          Console.WriteLine("Popping: {0}",numbers.Pop());
       }
       Console.ReadLine();
    }
}
```

本例运行输出的结果为：

```
Popping: 2
Popping: 4
Popping: 6
Popping: 3
```


3.7 自定义集合类

上面介绍的几种集合类已能满足大多数的应用，但有时程序中仍需要自定义集合类型。

3.7.1 实现 IEnumerable 接口

.NET 提供了 ICollection、IEnumerable、ICloneable 等与集合操作有关的接口，自定义集合类通常需要实现其中的一个或几个接口。

下面是一个自定义集合类的示例。

【例 3-10】 本例中定义了一个类，它包含一个 ArrayList 类型的成员用于容纳集合中的元素，并提供一些方法以实现 IEnumerable 接口，其源代码如下：

```
using System;
using System.Collections;
public class Contractor
    //定义 Contractor 类,它的实例将作为此后定义的 Contractors 集合类型中的元素
{
    private string name;
    private int rate;
    public Contractor(string Name,int Rate)        //定义构造函数
    {
        this.name = Name;
        this.rate = Rate;
    }
    public override string ToString()              //重载 ToString 方法定义输出格式
    {
        return string.Format("{0} [ $ {1:.00}]",this.name,this.rate);
    }
}

public class Contractors : IEnumerable             //自定义的集合类
{
    private ArrayList items = new ArrayList();
    public IEnumerator GetEnumerator()             //IEnumerable 接口中规定要实现的方法
    {
        for (int index = 0; index < this.Count; index++){
            yield return this[ index];
        }//yield return 语句使得可以多次返回值(集合类型 IEnumerator)
    }

    public int Add(string Name,int Rate)
```

```
    {
        return items.Add(new Contractor(Name,Rate));
    }

    public Contractor this[int Index]
    { //允许在实例名之后带含有索引的中括号访问 Items 中的成员
        get{
            return (Contractor)items[Index];
        }
    }

    public int Count                                    //Count 被定义为属性
    {
        get { return items.Count; }
    }
}

public class contractorTest
{
   static void Main(){
      Contractors myContractors = new Contractors();
      myContractors.Add("Thomas Andersen",12);   //注意,自定义的 Add 用两个参数
      myContractors.Add("Carole Poland",25);
      myContractors.Add("Nancy Anderson",65);
      myContractors.Add("Sidney Higa",48);
      foreach (Contractor c in myContractors)     //支持用 foreach 遍历
         Console.WriteLine(c);              //WriteLine 调用自定义 ToString 实现特殊输出格式
      for (int i = 0; i < myContractors.Count; i++)//也支持用索引器访问
         Console.WriteLine(myContractors[i]);
      Console.ReadLine( );
   }
}
```

本例运行输出的结果为：

```
Thomas Andersen [ $ 12.00]
Carole Poland [ $ 25.00]
Nancy Anderson [ $ 65.00]
Sidney Higa [ $ 48.00]
Thomas Andersen [ $ 12.00]
Carole Poland [ $ 25.00]
Nancy Anderson [ $ 65.00]
Sidney Higa [ $ 48.00]
```

说明：

(1) foreach 的执行是建立在该类型提供 IEnumerable 接口的基础上。IEnumerable 接口规定必须实现 GetEnumerator 方法，该方法返回一个实现了 IEnumerator 接口的对象实例。通过 IEnumerator 接口就能进行枚举操作(即遍历该集合)。

(2) public Contractor this[int Index]为集合类型定义索引器，使得可以用 myContractors[i]来表示 myContractors.Items[i]。索引器按习惯使用一对中括号([])，因

此也可以看作是对[]运算符的重载。此外注意,虽然索引器给人感觉可以用数组的方式对待集合,但本例中的索引器是作为只读属性定义的,因此并不能对被索引的元素赋值,与真正的数组在用法上仍有差距,不过读者也许能修改其代码使该索引器成为可读且可写的属性。

(3) yield return this[index]语句中的 yield 是 C# 的关键字;在有关说明文档中的解释是"在迭代器块中用于向枚举数对象提供值或发出迭代结束信号"。

(4) 因为 Console.WriteLine(c)是按对象 c 的 ToString 方法输出的,所以显示中出现[$12.00]。

(5) 本例中集合类 Contractors 中定义了 Add 方法,但没有定义 Remove 方法。事实上,Add 也不是实现 IEnumerable 接口所必需的。

3.7.2 继承 CollectionBase 类

CollectionBase 是.NET 下定义的一个抽象基类,具有一般集合类型中应有的一些基本特征。程序中可以通过继承 CollectionBase 类来实现自定义集合类型。一般来说,这种做法比较适合强类型对象的集合。表 3-7 介绍了 CollectionBase 类的常用属性和方法。

表 3-7 CollectionBase 类的常用属性和方法

	名 称	描 述
属性	Capacity	该属性表示 CollectionBase 集合中可存储的元素数目(应理解为利用已分配到的内存块最多就可存储这么多的元素,必要时系统会自动追加分配内存)
	Count	该属性表示集合中实际包含的元素数目
	List	这是一个受保护的属性,类型为 ArrayList。用于储存 CollectionBase 对象中的元素
方法	Clear	删除 CollectionBase 对象的所有元素
	GetEnumerator	返回一个对枚举数对象的引用,该对象用于循环访问
	RemoveAt	从 CollectionBase 对象中移除指定索引处的元素

下面提供一个示例。

【例 3-11】 本例模拟一个宠物诊所的运行,其中的自定义集合类型 Patients 是由 CollectionBase 派生的,其源代码如下:

```
using System;
using System.Collections;
public class Mammal { }                          //定义哺乳动物类,以下三行定义它的派生类
public class Dog : Mammal { }
public class Cat : Mammal { }
public class Whale : Mammal { }
public class Fish { }                            //定义鱼类
public class Reptile { }                         //定义爬行类
public class Patients : CollectionBase           //Patients 为自定义集合类
{
    public void AdmitPatient(object patient)
        //该方法用于收治一个宠物,相当于执行 Add,但添加了一些条件
        //仅当满足这些条件时才能实际完成添加
```

```
    {
        if ((patient) is Whale)
            Console.WriteLine(@"Can not add a whale as a patient.
                Who do you think we are Sea World!");
        else
            if ((patient)is Mammal || (patient)is Fish || (patient)is Reptile)
                this.List.Add(patient);
    }

    public void DischargePatient(object patient)
        //该方法相当于对集合执行的移除(Remove)
    {
        this.List.Remove(patient);
    }
}

public class VeterinaryClinic
{
    public static void Main(){
      Cat fluffy = new Cat();                          //创建一个 Cat 类实例
      Whale shamu = new Whale();                       //创建一个 Whale 类实例
      Fish goldy = new Fish();                         //创建一个 Fish 类实例
      Patients SickPets = new Patients();
        //SickPets 为 Patients 集合类的实例,它的元素是该宠物诊所中的病员
      SickPets.AdmitPatient(fluffy);                   //一个 Cat 被收治
      SickPets.AdmitPatient(shamu);                    //一个 Whale 来求治,但未被收治
      SickPets.AdmitPatient(goldy);                    //一个 Fish 被收治
      foreach (object pet in SickPets){
          Console.WriteLine(pet.ToString());           //输出对象名字
      }
      Console.WriteLine("Press Enter to exit...");
      Console.ReadLine();
  }
}
```

本例运行时,按规定婉拒了前来求治的鲸鱼 shamu,而其余一只猫和一条鱼均得以收治;最后再通过 foreach 语句显示全体病员名册。

说明:

(1) 由于继承了 CollectionBase 类,本例中的自定义集合类只需编写很少的代码。

(2) AdmitPatient 方法中调用 this.List.Add 方法进行添加,其中 List 是 CollectionBase 的被保护属性(只能在该类自身及其派生类中访问的属性),无法在 Main 方法中直接调用 List.Add。可以比较有效地对数据实行控制(例如,鲸鱼 shamu 被拒绝收治)。

(3) 本例定义的集合的元素不能用索引来访问,如果要使用索引访问这些元素,必须像例 3-10 中那样为 Patients 类定义一个索引器(即重载操作符[])。