细细品味C#

——泛型系列专题

精

华

集

锦

csAxp

虾皮工作室

http://www.cnblogs.com/xia520pi/

2011年7月29日

目录

1、关于java、C#泛型的看法

1.1、版权声明

文章出处：http://www.cnblogs.com/jobs/archive/2007/11/15/959802.html
文章作者：温少

1.2、内容详情

过去曾经有很长一段时间，直至现在，存在这样的一种观点，就是 C#比 Java 的实现更漂亮。《Think in java》的作者 Bruce Eckel 曾经公开质疑过 Java 5 提供的泛型。不过说实在，我一直不喜欢看 Bruce Eckel 的书，感觉上他不是一个有经验有深度的技术人员。

我也很长一段实现认同这样的观点，因为人云亦云！

在 C# 2.0 支持泛型，而且在虚拟机级别支持，一开始接触时，感觉是很震撼的，感觉到泛型从此走入主流应用开发了。和 C++相比，没有 C++模板那样强的功能，完全做不到产生式编程的效果，也做不到编译期计算的效果，但是它简单实用。

Java 5 也开始支持泛型，而且最终正式发行比 C# 2.0 要早，我之前就使用过 Beta 版本 C#的泛型，也熟悉 C++的模板语法，可能是内心的傲慢，或者是懒惰，开始时只是将就着按照传统的经验使用 Java 5 提供的泛型。

对事物的一知半解总是令人困扰的，在阅读分析 JDK 源码时，总会遇到一些 Java 5 额外提供的泛型用法，一开始忽略不计，但是看多了总会注意到的。

例如 java.util.Collections 类中的 sort 方法和 binarySearch 方法的接口：

```
public static <T> void sort(List<T> list, Comparator<? super T> c);
public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key);
```

extends 和 super 这两个关键字是 C#和 C++的泛型中都没有的，为什么需要这样的功能呢？

例如如下情形：

```
class A { }
class B extends A {}
void addAll(List<A> items) {}
```

如下代码：

```
List<A> aList = ;
List<B> bList = ;
addAll(aList); //可以
addAll(bList); //编译不通过
```

addAll(bList)是无法编译通过的，这一点在 Java、C#、C++中都是如此，怎么办呢？在 java 中如下处理，修改 addAll 的接口，改为：

```
void addAll(List<? extends A> items) {}
```

这样，addAll(aList)和 addAll(bList)都能够编译通过了。

另外 super 关键在算法中更是好用，如上面介绍的 Collections.sort 方法。如果你想在 C# 中实现一个和 java.util.Collections.sort 一样的方法，你会发现那是做不到的！

为什么 C#和 C++无法提供这样的功能呢？因为 C#和 C++都是运行时的泛型支持，bList 和 aList 的类型是不一样的，List<A>和 List<B>的实际类型都是不一样的，运行时对泛型的支持目前还无法象处理数组参数那样具备协变能力。而 Java 的实现是编译器的特性，这样做的缺点就是性能没有得到提升，但是可以提供更好的语法糖。

想起 ajoo 以前发表的一个观点，就是在应用开发中，泛型提供的关键是类型安全，性能反而是其次。我对此十分认同，重新审视 java 的泛型，我们会发现其设计颇具创新，而且向后兼容良好！

总结一下我的观点：

Java 的泛型，语法有创新，更好用，向后兼容，编写泛型算法更方便，但是没有带来性能提升。

C#泛型，实现有创新，在虚拟机级别支持，运行时支持泛型，性能有提升，但是不好编写泛型算法，不向后兼容。

2、C# 泛型的优点

2.1、版权声明

文章出处：http://www.cnblogs.com/pegger/archive/2008/05/11/1192348.html
文章作者：pegger

2.2、内容详情

2.2.1、泛型介绍

泛型是用来做什么的？答案是：没有泛型，将会很难创建类型安全的集合。

C# 是一个类型安全的语言，类型安全允许编译器(可信赖地)捕获潜在的错误，而不是在程序运行时才发现(不可信赖地，往往发生在你将产品出售了以后！)。因此，在 C#中，所有的变量都有一个定义了的类型；当你将一个对象赋值给那个变量的时候，编译器检查这个赋值是否正确，如果有问题，将会给出错误信息。

在 .Net 1.1 版本(2003)中，当你在使用集合时，这种类型安全就失效了。由.Net 类库提供的所有关于集合的类全是用来存储基类型(Object)的，而.Net 中所有的一切都是由 Object 基类继承下来的，因此所有类型都可以放到一个集合中。于是，相当于根本就没有了类型检测。

更糟的是，每一次你从集合中取出一个 Object，你都必须将它强制转换成正确的类型，这一转换将对性能造成影响，并且产生冗长的代码(如果你忘了进行转换，将会抛出异常)。更进一步地讲，如果你给集合中添加一个值类型(比如，一个整型变量)，这个整型变量就被隐式地装箱了(再一次降低了性能)，而当你从集合中取出它的时候，又会进行一次显式地拆箱(又一次性能的降低和类型转换)。

在公共语言运行库和 C# 语言的早期版本中，通用化是通过在类型与通用基类型 Object 之间进行强制转换来实现的，泛型提供了针对这种限制的解决方案。通过创建泛型类，您可以创建一个在编译时类型安全的集合。

使用非泛型集合类的限制可以通过编写一小段程序来演示，该程序利用 .NET Framework 基类库中的 ArrayList 集合类。ArrayList 是一个使用起来非常方便的集合类，无需进行修改即可用来存储任何引用或值类型。

```
System.Collections.ArrayList list1 = new System.Collections.ArrayList();
list1.Add(3);
```

```
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayList();
list2.Add("It is raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

但这种方便是需要付出代价的。添加到 ArrayList 中的任何引用或值类型都将隐式地向上强制转换为 Object。如果项是值类型，则必须在将其添加到列表中时进行装箱操作，在检索时进行取消装箱操作。强制转换以及装箱和取消装箱操作都会降低性能；在必须对大型集合进行循环访问的情况下，装箱和取消装箱的影响非常明显。

另一个限制是缺少编译时类型检查；因为 ArrayList 将把所有项都强制转换为 Object，所以在编译时无法防止客户端代码执行以下操作：

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an error later.
list.Add("It is raining in Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
    t += x;
}
```

尽管将字符串和 ints 组合在一个 ArrayList 中的做法在创建异类集合时是完全合法的，有时是有意图的，但这种做法更可能产生编程错误，并且直到运行时才能检测到此错误。

在 C# 语言的 1.0 和 1.1 版本中，只能通过编写自己的特定于类型的集合来避免 .NET Framework 基类库集合类中的通用代码的危险。当然，由于此类不可对多个数据类型重用，因此将丧失通用化的优点，并且您必须对要存储的每个类型重新编写该类。

ArrayList 和其他相似类真正需要的是：客户端代码基于每个实例指定这些类要使用的具体数据类型的方式。这样将不再需要向上强制转换为 T:System.Object，同时，也使得编译器可以进行类型检查。换句话说，ArrayList 需要一个 type parameter。这正是泛型所能提供的。在 N:System.Collections.Generic 命名空间的泛型 List<T> 集合中，向该集合添加项的操作类似于以下形式：

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();
```

```
// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

对于客户端代码，与 ArrayList 相比，使用 List<T> 时添加的唯一语法是声明和实例化中的类型参数。虽然这稍微增加了些编码的复杂性，但好处是您可以创建一个比 ArrayList 更安全并且速度更快的列表，特别适用于列表项是值类型的情况。

归纳起来，泛型比非泛型具有下面两个优点：

1）更加安全

在非泛型编程中，虽然所有的东西都可以作为 Object 传递，但是在传递的过程中免不了要进行类型转换。而类型转换在运行时是不安全的。使用泛型编程将可以减少不必要的类型转换，从而提高安全性。不仅是值类型，引用类型也存在这样的问题，因此有必要的尽量的去使用泛型集合。

2）效率更高

在非泛型编程中，将简单类型作为 Object 传递时会引起装箱和拆箱的操作，这两个过程都是具有很大开销的。使用泛型编程就不必进行装箱和拆箱操作了。

2.2.2、泛型集合

通常情况下，建议您使用泛型集合，因为这样可以获得类型安全的直接优点而不需要从基集合类型派生并实现类型特定的成员。下面的泛型类型对应于现有的集合类型：

1）List 是对应于 ArrayList 的泛型类。
2）Dictionary 是对应于 Hashtable 的泛型类。
3）Collection 是对应于 CollectionBase 的泛型类。
4）ReadOnlyCollection 是对应于 ReadOnlyCollectionBase 的泛型类。
5）Queue、Stack 和 SortedList 泛型类分别对应于与其同名的非泛型类。
6）LinkedList 是一个通用链接列表，它提供运算复杂度为 O(1) 的插入和移除操作。
7）SortedDictionary 是一个排序的字典，其插入和检索操作的运算复杂度为 O(log n)，这使得它成为 SortedList 的十分有用的替代类型。
8）KeyedCollection 是介于列表和字典之间的混合类型，它提供了一种存储包含自己键的对象的方法。

3、C# 泛型秘诀

3.1、版权声明

文章出处：http://www.cnblogs.com/abatei/archive/2008/02/20/1075760.html
文章作者：abatei

3.2、内容详情

3.2.1、理解泛型

介绍

泛型，一个期待已久的功能，随着 C# 2.0 版本编译器的到来最终出现。泛型是一个非常有用的功能，它使得您的代码变得精简而富有效率。这些将在秘诀 4.1 进行详细讲述。泛型的到来使得您可以编写更为强大的应用程序，但这需要正确地使用它。如果您考虑把 ArrayList，Queue，Stack 和 Hashtable 对象转变为使用相应的泛型版本，可以阅读秘诀 4.4，4.5 和 4.10。当您阅读过后，会发现这种转变不一定简单，甚至有可能会不再打算进行转变。

本章的另外一些秘诀涉及到.NET Framework 2.0 所包含的其他泛型类，如秘诀 4.6。其他秘诀讲述一些泛型类的操作，如秘诀 4.2，4.8 和 4.13。

决定在何时何地使用泛型

问题

您希望在一个新工程内使用泛型，或者想把已有项目中的非泛型类转换为等价的泛型版本。但您并非了解为何要这样做，也不知道哪个非泛型类应该被转换为泛型类。

解决方案

决定在何时何地使用泛型，您需要考虑以下几件事件：

- 您所使用的类型是否包含或操作未指定的数据类型（如集合类型）？如果是这样，如果是这样，创建泛型类型将能提供更多的好处。如果您的类型只操作单一的指定类型，那么就没有必要去创建一个泛型类。
- 如果您的类型将操作值类型，那么就会产生装箱和拆箱操作，就应该考虑使用泛型来防止装箱和拆箱操作。
- 泛型的强类型检查有助于快速查找错误（也就是编译期而非运行期），从而缩短 bug 修复周期。
- 在编写多个类操作多个数据类型时是否遭遇到“代码膨胀”问题（如一个 ArrayList 只存储 StreamReaders 而另一个存储 StreamWriters）？其实编写一次代码并让它工作于多个数据类型非常简单。
- 泛型使得代码更为清晰。通过消除代码膨胀并进行强制检查，您的代码将变得更易

于阅读和理解。

讨论

很多时候，使用泛型类型将使您受益。泛型将使得代码重用更有效率，具有更快的执行速度，进行强制类型检查，获得更易读的代码。

阅读参考

MSDN 文档中的“Generics Overview”和“Benefits of Generics”主题。

理解泛型类型

问题

您需要理解泛型类型在.NET 中是如何工作的，它跟一般的.NET 类型有什么不同。

解决方案

几个小实验就可以演示一般类型和泛型类型之间的区别。例 4-1 中的 StandardClass 类就是一个般类型。

例 4-1 StandardClass：一般的.NET 类型

```
public class StandardClass
{
    static int _count = 0; //StandardClass 类的对象的表态计数器
    int _maxItemCount; //项数目的最大值
    object[] _items; //保存项的数组
    int _currentItem = 0; //当前项数目
    public StandardClass(int items) //构造函数
    {
        _count++; //对象数目加
        _maxItemCount = items;
        _items = new object[_maxItemCount]; //数组初始化
    }
    //用于添加项，为了适用于任何类型，使用了 object 类型
    public int AddItem(object item)
    {
        if (_currentItem < _maxItemCount)
        {
            _items[_currentItem] = item;
            return _currentItem++; //返回添加的项的索引
        }
        else
            throw new Exception("Item queue is full");
    }
    //用于从类中取得指定项
    public object GetItem(int index)
    {
        Debug.Assert(index < _maxItemCount); //设置断言
        if (index >= _maxItemCount)
```

```
                throw new ArgumentOutOfRangeException("index");
            return _items[index]; //返回指定项
        }
        public int ItemCount //属性，指示当前项数目
        {
            get { return _currentItem; }
        }
        public override string ToString( )
        {    //重载 ToString 方法，用于介绍类的情况
            return "There are " + _count.ToString( ) +
                " instances of " + this.GetType( ).ToString( ) +
                " which contains " + _currentItem + " items of type " +
                _items.GetType( ).ToString( ) + "";
        }
}
```

StandardClass 类有一个整型静态成员变量_count，用于在实例构造器中计数。重载的ToString()方法打印在这个应用程序域中 StandardClass 类实例的数目。StandardClass 类还包括一个 object 数组（_item），它的长度由构造方法中的传递的参数来决定。它实现了添加和获得项的方法（AddItem， GetItem），还有一个只读属性来获取数组中的项的数目（ItemCount）。

GenericClass<T>类是一个泛型类型，同样有静态成员变量_count，实例构造器中对实例数目进行计算，重载的 ToString()方法告诉您有多少 GenericClass<T>类的实例存在。GenericClass<T>也有一个_itmes 数组和 StandardClass 类中的相应方法，请参考例 4-2。

Example4-2 GenericClass<T>：泛型类

```
public class GenericClass<T>
{
    static int _count = 0;
    int _maxItemCount;
    T[] _items;
    int _currentItem = 0;
    public GenericClass(int items)
    {
        _count++;
        _ _maxItemCount = items;
        _items = new T[_maxItemCount];
    }
    public int AddItem(T item)
    {
        if (_currentItem < _maxItemCount)
        {
            _items[_currentItem] = item;
            return _currentItem++;
```

```
            }
            else
                throw new Exception("Item queue is full");
        }
        public T GetItem(int index)
        {
            Debug.Assert(index < _maxItemCount);
            if (index >= _maxItemCount)
                throw new ArgumentOutOfRangeException("index");
            return _items[index];
        }
        public int ItemCount
        {
            get { return _currentItem; }
        }
        public override string ToString()
        {
            return "There are " + _count.ToString() +
                " instances of " + this.GetType().ToString() +
                " which contains " + _currentItem + " items of type " +
                _items.GetType().ToString() + "";
        }
}
```

从 GenericClass<T>中的少许不同点开始，看看_items 数组的声明。它声明为：

```
T[] _items;
```

而不是

```
object[] _items;
```

_items 数组使用泛型类（<T>）做为类型参数以决定在_itmes 数组中接收哪种类型的项。StandarClass 在_itmes 数组中使用 Objcec 以使得所有类型都可以做为项存储在数组中（因为所有类型都继承自 object）。而 GenericClass<T>通过使用类型参数指示允许使用的对象类型来提供类型安全。

下一个不同在于 AddItem 和 GetItem 方法的声明。AddItem 现在使用一个类型 T 做为参数，而在 StandardClass 中使用 object 类型做为参数。GetItem 现在的返回值类型 T，StandardClass 返回值为 object 类型。这个改变允许 GenericClass<T>中的方法在数组中存储和获得具体的类型而非 StandardClass 中的允许存储所有的 object 类型。

```
public int AddItem(T item)
    {
```

```
        if (_currentItem < _maxItemCount)
        {
            _items[_currentItem] = item;
            return _currentItem++;
        }
        else
            throw new Exception("Item queue is full");
    }
    public T GetItem(int index)
    {
        Debug.Assert(index < _maxItemCount);
        if (index >= _maxItemCount)
            throw new ArgumentOutOfRangeException("index");
        return _items[index];
    }
```

这样做的优势在于，首先通过 GenericClass<T>为数组中的项提供了类型安全。在 StandardClass 中可能会这样写代码：

```
// 一般类
    StandardClass C = new StandardClass(5);
    Console.WriteLine(C);
    string s1 = "s1";
    string s2 = "s2";
    string s3 = "s3";
    int i1 = 1;
    // 在一般类中以 object 的形式添加项
    C.AddItem(s1);
    C.AddItem(s2);
    C.AddItem(s3);
    // 在字符串数组中添加一个整数，也被允许
    C.AddItem(i1);
```

但在 GenericClass<T>中做同样的事情将导致编译错误：

```
    // 泛型类
    GenericClass<string> gC = new GenericClass<string>(5);
    Console.WriteLine(gC);
    string s1 = "s1";
    string s2 = "s2";
    string s3 = "s3";
    int i1 = 1;
    // 把字符串添加进泛型类.
```

```
gC.AddItem(s1);
gC.AddItem(s2);
gC.AddItem(s3);
// 尝试在字符串实例中添加整数，将被编译器拒绝
// error CS1503: Argument '1': cannot convert from 'int' to 'string'
//GC.AddItem(i1);
```

编译器防止它成为运行时源码的 bug，这是一件非常美妙的事情。
虽然并非显而易见，但在 StandardClass 中把整数添加进 object 数组会导致装箱操作，这一点可以 StandardClass 调用 GetItem 方法时的 IL 代码：

```
IL_0170: ldloc.2
IL_0171: ldloc.s i1
IL_0173: box [mscorlib]System.Int32
IL_0178: callvirt instance int32  CSharpRecipes.Generics/StandardClass::AddItem(object)
```

这个装箱操作把做为值类型的整数转换为引用类型（object），从而可以在数组中存储。这导致了在 object 数组中存储值类型时需要增加额外的工作。

当您在运行 StandardClass 并从类中返回一个项时，还会产生一个问题，来看看 StandardClass.GetItem 如何返回一个项：

```
// 存储返回的字符串.
string sHolder;
// 发生错误 CS0266:
// Cannot implicitly convert type 'object' to 'string'…
sHolder = (string)C.GetItem(1);
```

因为 StandardClass.GetItem 返回的是 object 类型，而您希望通过索引 1 获得一个字符串类型，所以需要把它转换为字符串类型。然而它有可能并非字符串-----只能确定它是一个 object-----但为了赋值正确，您不得不把它转换为更明确的类型。字符串比较特殊，所有对象都可以自行提供一个字符串描述，但当数组接收一个 double 类型并把它赋给一个布尔类型就会出问题。

这两个问题在 GenericClass<T>中被全部解决。无需再进行拆箱，因为 GetItem 所返回的是一个具体类型，编译器会检查返回值以强近它执行。

```
string sHolder;
int iHolder;
// 不需要再进行转换
sHolder = gC.GetItem(1);
// 尝试把字符串变为整数.将出现
// 错误 CS0029: Cannot implicitly convert type 'string' to 'int'
//iHolder = gC.GetItem(1);
```

为了了解两种类型的其他不同点，分别给出它们的示例代码：

```
// 一般类
StandardClass A = new StandardClass();
Console.WriteLine(A);
StandardClass B = new StandardClass();
Console.WriteLine(B);
StandardClass C = new StandardClass();
Console.WriteLine(C);
// 泛型类
GenericClass<bool> gA = new GenericClass<bool>();
Console.WriteLine(gA);
GenericClass<int> gB = new GenericClass<int>();
Console.WriteLine(gB);
GenericClass<string> gC = new GenericClass<string>();
Console.WriteLine(gC);
GenericClass<string> gD = new GenericClass<string>();
Console.WriteLine(gD);
```

上述代码输出结果如下：

```
There are 1 instances of CSharpRecipes.Generics+StandardClass which contains 0 items of type System.Object[]...
There are 2 instances of CSharpRecipes.Generics+StandardClass which contains 0 items of type System.Object[]...
There are 3 instances of CSharpRecipes.Generics+StandardClass which contains 0 items of type System.Object[]...
There are 1 instances of CSharpRecipes.Generics+GenericClass`1[System.Boolean] which contains 0 items of type System.Boolean[]...
There are 1 instances of CSharpRecipes.Generics+GenericClass`1[System.Int32] which contains 0 items of type System.Int32[]...
There are 1 instances of CSharpRecipes.Generics+GenericClass`1[System.String] which contains 0 items of type System.String[]...
There are 2 instances of CSharpRecipes.Generics+GenericClass`1[System.String] which contains 0 items of type System.String[]...
```

讨论

泛型中的类型参数允许您在不知道使用何种类型的情况下提供类型安全的代码。在很多场合下，您希望类型具有某些指定的特征，这可以通过使用类型约束（秘诀 4.12）来实现。方法在类本身不是泛型的情况下也可以拥有泛型类型的参数。秘诀 4.9 为此演示了一个例子。

注意当 StandardClass 拥有三个实例，GenericClass 有一个声明为<bool>类型的实例，一个声明为<int>类型的实例，两个声明为<string>类型的实例。这意味着所有非泛型类都创建同一.NET 类型对象，而所有泛型类都为指定类型实例创建自己的.NET 类型对象。

示例代码中的 StandardClass 类有三个实例，因为 CLR 中只维护一个 StandardClass 类型。

而在泛型中，每种类型都被相应的类型模板所维护，当创建一个类型实例时，类型实参被传入。说得更清楚一些就是为 GenericClass<bool>产生一个类型，为 GenericClass<int>产生一个类型，为 GenericClass<string>产生第三个类型。

内部静态成员_count 可以帮助说明这一点，一个类的静态成员实际上是跟 CLR 中的类型相连的。CLR 对于给定的类型只会创建一次，并维护它一直到应用程序域卸载。这也是为什么在调用 ToString 方法时，输出显示有 StandardClass 的三个实例，而 GenericClass<T>类型有 1 和 2 个实例。

3.2.2、获取泛型类型和使用相应的泛型版本替换ArrayList

获取泛型的类型

问题

您需要在运行时获得一个泛型类型实例的 Type 对象。

解决方案

在使用 typeof 操作符时提供类型参数；使用类型参数实例化的泛型类型，用 GetType()方法。

声明一个一般类型和一个泛型类型如下：

```
public class Simple
{
    public Simple()
    {
    }
}
public class SimpleGeneric<T>
{
    public SimpleGeneric(){}
}
```

使用 typeof 操作符和简单类型的名称就可以在运行时获得简单类型的类型。对于泛型类型来说，在调用 typeof 时类型参数必须要提供，但是简单类型实例和泛型类型实例都可以使用相同的方式来调用 GetType()。

```
Simple s = new Simple();
    Type t = typeof(Simple);
    Type alsoT = s.GetType();
    //提供类型参数就才可以获得类型实例
    Type gtInt = typeof(SimpleGeneric<int>);
    Type gtBool = typeof(SimpleGeneric<bool>);
    Type gtString = typeof(SimpleGeneric<string>);
    // 当有一个泛型类实例时，您也可以使用 GetType 的旧的方式去调用一个实例。.
    SimpleGeneric<int> sgI = new SimpleGeneric<int>();
```

```
Type alsoGT = sgI.GetType();
```

讨论

不能直接获取泛型类的类型，因为如果不提供一个类型参数，泛型类将没有类型（参考秘诀 4.2 获得更多信息）。只有通过类型参数实例化的泛型类才有 Type。
如果您在使用 typeof 操作符时，只提供泛型类型定义而不提供类型参数，将得到下面的错误：

```
// 这产生一个错误：
// Error 26 Using the generic type 'CSharpRecipes.Generics.SimpleGeneric<T>'
// requires '1' type arguments
Type gt = typeof(SimpleGeneric);
```

使用相应的泛型版本替换 ArrayList

问题

您希望通过将所有 ArrayList 对象替换为相应的泛型版本以提高应用程序的效率，并使得代码更易于使用。当结构体或其他值类型存储在这些数据结构中时，会导致装箱/拆箱操作，这时就需要这么做。

解决方案

使用更有效率的泛型类 System.Collections.Generic.List 来替换已存在的 System.Collection.ArrayList 类。
下面是使用 System.Collection.ArrayList 对象的简单例子：

```
public static void UseNonGenericArrayList()
    {
        // 创建一个 ArrayList.
        ArrayList numbers = new ArrayList();
        numbers.Add(1); // 导致装箱操作
        numbers.Add(2); // 导致装箱操作
        // 显示 ArrayList 内的所有整数
        // 每次迭代都导致拆箱操作
        foreach (int i in numbers)
        {
            Console.WriteLine(i);
        }
        numbers.Clear();
}
```

相同的代码使用了 System.Collections.Generic.List 对象

```
public static void UseGenericList()
    {
        // 创建一个 List.
```

```
        List<int> numbers = new List<int>();
        numbers.Add(1);
        numbers.Add(2);
        // 显示 List 中的所有整数.
        foreach (int i in numbers)
        {
            Console.WriteLine(i);
        }
        numbers.Clear();
}
```

讨论

因为所有的应用程序几乎都会使用 ArrayList，从提升您的应用程序的执行效率开始是一个不错的选择。对于应用程序中简单使用 ArrayList 的地方来说，这种替代是非常容易的。但有些地方需要注意，例如，泛型 List 类未实现 Icloneable 接口而 ArrayList 实现了它。表 4-1 显示了两个类中的等价成员。

ArrayList类成员	等价的泛型List类成员
Capacity 属性	Capacity属性
Count属性	Count属性
IsFixedSize属性	(IList)myList).IsFixedSize
IsReadOnly属性	(IList)myList).IsReadOnly
IsSynchronized属性	(IList)myList).IsSynchronized
Item属性	Item属性
SyncRoot属性	(IList)myList).SyncRoot
Adapter 静态方法	N/A
Add 方法	Add方法
AddRange方法	AddRange方法
N/A	AsReadOnly方法
BinarySearch方法	BinarySearch方法
Clear方法	Clear方法
Clone方法	Getrange(0, numbers.Count)
Contains方法	Contains方法
N/A	ConvertAll方法
CopyTo方法	CopyTo方法
N/A	Exists方法
N/A	Find方法

N/A	FindAll方法
N/A	FindIndex方法
N/A	FindLast方法
N/A	FindLastIndex方法
N/A	ForEach方法
FixedSize 静态方法	N/A
Getrange方法	Getrange方法
IndexOf方法	IndexOf方法
Insert方法	Insert方法
InsertRange方法	InsertRange方法
LastIndexOf方法	LastIndexOf方法
ReadOnly 静态方法	AsReadOnly方法
Remove方法	Remove方法
N/A	RemoveAll方法
RemoveAt方法	RemoveAt方法
RemoveRange方法	RemoveRange方法
Repeat 静态方法	使用for循环和Add方法
Reverse方法	Reverse方法
SetRange方法	InsertRange方法
Sort方法	Sort方法
Synchronized 静态方法	lock(myList.SyncRoot) {…}
ToArray方法	ToArray方法
N/A	trimExcess方法
TRimToSize方法	trimToSize方法
N/A	trueForAll方法

表 4-1 中的几个 ArrayList 的成员和泛型 List 的成员并非一一对应。从属性开始说，只有 Capacity，Count 和 Item 属性两个类中都存在。为了弥补 List 类中的几个缺失的属性，可以把它显式转换为 Ilist 接口。下面的代码演示了如何使用这些显式转换以获得缺失的属性。

```
List<int> numbers = new List<int>();
Console.WriteLine(((IList)numbers).IsReadOnly);
Console.WriteLine(((IList)numbers).IsFixedSize);
Console.WriteLine(((IList)numbers).IsSynchronized);
Console.WriteLine(((IList)numbers).SyncRoot);
```

注意，由于缺少返回同步版本的泛型 List 代码和缺少返回固定尺寸的泛型 List 代码,IsFixedSize 和 IsSynchronized 属性将总是返回 false。SyncRoot 属性被调用时将总是返回

相同的对象，本质上这个属性返回 this 指针。微软已经决定从所有泛型集合类中去除创建同步成员的功能。做为代替，他们推荐使用 lock 关键字去锁住整个集合或其他类型的同步对象来满足您的需要。

静态的 ArrayList.Repeat 在泛型 List 中没有对应的方法。做为代替，您可以使用下面的泛型方法：

```
public static void Repeat<T>(List<T> list, T obj, int count)
    {
        if (count < 0)
        {
            throw (new ArgumentException(
                    "参数 count  必须大于或等于零"));
        }
        for (int index = 0; index < count; index++)
        {
            list.Add(obj);
        }
}
```

这个泛型方法有三个参数：

list

泛型 List 对象

obj

将被以指定次数添加进泛型 List 中的对象

count

把 obj 添加进泛型类中的次数

因为 Clone 方法也没有出现在泛型 List 类中（因为这个类并没有实现 Icloneable 接口），您可以使用泛型 List 类的 GetRange 方法做为替代。

```
List<int> oldList = new List<int>();
    // 给 oldList 添加元素…
List<int> newList = oldList.GetRange(0, oldList.Count);
```

GetRange 方法对 List 对象中一个范围的元素执行浅拷贝（跟 ArrayList 中的 Clone 方法接近）。在此例中这个范围是所有元素。

提示：ArrayList 默认的初始容量是 16 个元素，而 List<T>的默认初始容量为 4 个元素。这意味着当添加第 17 个元素时，List<T>不得不改变尺寸（重新分配内存）3 次，而 ArrayList 只重新分配一次。这一点在评估应用程序性能时需要被考虑。

3.2.3、使用相应的泛型版本替换Stack和Queue

使用相应的泛型版本替换 Stack 和 Queue

问题

您希望通过将所有 Stack 和 Queue 对象替换为相应的泛型版本以提高应用程序的效率，并使得代码更易于使用。当结构体或其他值类型存储在这些数据结构中时，会导致装箱/拆箱操作，这时就需要这么做。

解决方案

使用 System.Collections.Generic.Stack 和 System.Collections.Generic.Queue 对象来替换现有的 System.Collections.Stack 和 System.Collections.Queue 对象。

这里有一个简单地使用 System.Collections.Queue 对象的简单例子：

```
public static void UseNonGenericQueue()
    {
        // 创建一个非泛型队列对象
        Queue numericQueue = new Queue();
        // 进队(导致装箱操作).
        numericQueue.Enqueue(1);
        numericQueue.Enqueue(2);
        numericQueue.Enqueue(3);
        //出队并显示项(导致拆箱操作)
        Console.WriteLine(numericQueue.Dequeue());
        Console.WriteLine(numericQueue.Dequeue());
        Console.WriteLine(numericQueue.Dequeue().ToString());
}
```

下面是相同的代码使用了 System.Collections.Generic.Queue 对象

```
public static void UseGenericQueue()
    {
        // 创建一个泛型队列对象.
        Queue<int> numericQueue = new Queue<int>();
        // 进队.
        numericQueue.Enqueue(1);
        numericQueue.Enqueue(2);
        numericQueue.Enqueue(3);
        // 出队并显示项目.
        Console.WriteLine(numericQueue.Dequeue());
        Console.WriteLine(numericQueue.Dequeue());
        Console.WriteLine(numericQueue.Dequeue());
}
```

下面是一个简单地使用 System.Collections.Stack 对象的例子

```
public static void UseNonGenericStack()
    {
```

```
            // 创建一个非泛型栈.
            Stack numericStack = new Stack();
            // 进栈(导致装箱操作).
            numericStack.Push(1);
            numericStack.Push(2);
            numericStack.Push(3);
            // 出栈并显示项目(导致拆箱操作).
            Console.WriteLine(numericStack.Pop().ToString());
            Console.WriteLine(numericStack.Pop().ToString());
            Console.WriteLine(numericStack.Pop().ToString());
}
```

下面是相同的代码使用了 System.Collections.Generic.Stack 对象

```
 public static void UseGenericStack()
    {
            // 创建一个泛型栈对象.
            Stack<int> numericStack = new Stack<int>();
            // 进栈.
            numericStack.Push(1);
            numericStack.Push(2);
            numericStack.Push(3);
            // 出栈并显示项目.
            Console.WriteLine(numericStack.Pop().ToString());
            Console.WriteLine(numericStack.Pop().ToString());
            Console.WriteLine(numericStack.Pop().ToString());
}
```

讨论

表面上，泛型和非泛型的 Queue 和 Stack 类非常相象。但在内部机制上却有极大地不同。除了实例化对象之外，泛型版的 Queue 和 Stack 和非泛型版的 Queue 和 Stack 的使用是基本相同的。泛型结构为了创建一个类型需要一个类型参数。在此例中，类型参数是 int。这个类型参数表明 Queue 和 Stack 对象将只能包含整数类型和能够被隐式转换为整数的类型，比如 short 类型：

```
    short s = 300;
    numericQueue.Enqueue(s); // 成功，因为进行了隐式转换
```

但如果一个类型不能被隐式转换为整数，如 double，将会导致一个编译期错误。

```
    double d = 300;
    numericQueue.Enqueue(d);            // 错误，不能进行隐式转换
```

```
numericQueue.Enqueue((int)d);     // 成功，因为进行了显式转换
```

非泛型结构不需要类型参数，因为非泛型版的 Queue 和 Stack 对象只能包含 Object 类型。（译者注：任何对象都可以隐式转换为 Object，并于这一点有不清楚的请参考：

http://cgbluesky.blog.163.com/blog/static/241235582007124934194 58/ ）

当需要在 Queue 或 Stack 的泛型版和非泛型版做出选择时，您需要决定使用强类型 Queue 或 Stack 对象（也就是泛型版 Queue 或 Stack 类）还是弱类型的 Queue 或 Stack 对象（也就是非泛型版 Queue 或 Stack 类）。选择泛型版 Queue 或 Stack 类相对于非泛型版来说会提供很多好处，包括：

类型安全

包含在数据结构中的每个元素都是指定的类型。这意味着当在数据结构中进行添加或删除操作时不再需要将它们转换成 object 类型。您不能在一个单一数据结构中存储多种不同的类型；您总是知道数据结构中存储的是什么类型。在编译期进行类型检查优于在运行期进行检查。可以归结为更简洁的代码，更好的性能，更少的错误。

缩短开发周期

创建一个类型安全的数据结构而不使用泛型意味着不得不从 System.Collections.Stack 或 System.Collections.Queue 继承创建自己的子类，这是一个耗时并容易发生错误的工作。而泛型只需您在编译期简单地告诉 Queue 或 Stack 对象控制什么类型。

性能

使用泛型 Queue 或 Stack 在添加和删除元素时避免了潜在的费时的类型转换的发生。另外在把值类型添加进 Queue 或 Stack 时不会发生装箱操作，从 Queue 或 Stack 删除值类型时也不会发生拆箱操作。

容易阅读的代码

您的基础代码将变得非常少，因为不再需要从非泛型版的 Queue 或 Stack 类继续创建您自己的强类型类。另外泛型代码的类型安全功能将使在代码中使用 Queue 或 Stack 类的目的变得更容易理解。

泛型版和非泛型版的 Queue 和 Stack 的不同之处在于两种类之间的成员实现。在非泛型版实现而在泛型版没有实现的成员列表如下：

```
Clone 方法
IsSynchronized 属性
SyncRoot 属性
Synchronized 方法
```

非泛型版 Queue 和 Stack 类中存在 Clone 方法是因为只有它实现了 Icolneable 接口。但非泛型版 Queue 和 Stack 类实现的其他接口是一样的。

一个弥补泛型版 Queue 和 Stack 类中不存在 Clone 方法的途径是接收一个 Ienumerable<T>类型。因为这是 Queue 和 Stack 类实现的接口之一，对于 Queue 对象来说，这非常容易实现，代码如下：

```
public static void CloneQueue()
    {
        // 创建一个 Queue 对象.
        Queue<int> numericQueue = new Queue<int>();
        // 进队
```

```
            numericQueue.Enqueue(1);
            numericQueue.Enqueue(2);
            numericQueue.Enqueue(3);
            // 创建一个克隆的 numericQueue.
            Queue<int> clonedNumericQueue = new Queue<int>(numericQueue);
            // 这只是简单地看一下里面的值，并非出队
            foreach (int i in clonedNumericQueue)
            {
                Console.WriteLine("foreach: " + i.ToString());
            }
            // 出队并显示项目
            Console.WriteLine(clonedNumericQueue.Dequeue().ToString());
            Console.WriteLine(clonedNumericQueue.Dequeue().ToString());
            Console.WriteLine(clonedNumericQueue.Dequeue().ToString());
}
```

方法的输出结果如下：

```
    foreach: 1
    foreach: 2
    foreach: 3
    1
    2
    3
```

对于 Stack 对象，其代码如下：

```
 public static void CloneStack()
    {
        // 创建一个泛型 Stack 对象
        Stack<int> numericStack = new Stack<int>();
        // 进栈
        numericStack.Push(1);
        numericStack.Push(2);
        numericStack.Push(3);
        // 克隆 numericStack  对象.
        Stack<int> clonedNumericStack = new Stack<int>(numericStack);
        // 这只是简单地看一下里面的值，并非出栈
        foreach (int i in clonedNumericStack)
        {
            Console.WriteLine("foreach: " + i.ToString());
        }
        // 出栈并显示项目
```

```
            Console.WriteLine(clonedNumericStack.Pop().ToString());
            Console.WriteLine(clonedNumericStack.Pop().ToString());
            Console.WriteLine(clonedNumericStack.Pop().ToString());
}
```

方法的输出结果如下：

```
foreach: 1
foreach: 2
foreach: 3
1
2
3
```

构造方法创建了一个新的 Queue 或 Stack 实例，并包含了 Ienumerable<T>类型中所有元素的一份拷贝。

3.2.4、链表的实现和可空类型

链表的实现

问题

您需要链表数据结构，这样就可以很容易地添加和删除元素。

解决方案

使用泛型 LinkedList<T>类。下面的方法创建了一个 LinkedList<T>类，并往链表对象中添加节点，然后使用了几种方法从链表节点中获得信息。

```
public static void UseLinkedList()
    {
        // 创建一个 LinkedList 对象.
        LinkedList<TodoItem> todoList = new LinkedList<TodoItem>();
        // 创建添加到链表内的 TodoItem 对象.
        TodoItem i1 = new TodoItem("paint door", "Should be done third");
        TodoItem i2 = new TodoItem("buy door", "Should be done first");
        TodoItem i3 = new TodoItem("assemble door", "Should be done second");
        TodoItem i4 = new TodoItem("hang door", "Should be done last");
        // 添加项目.
        todoList.AddFirst(i1);
        todoList.AddFirst(i2);
        todoList.AddBefore(todoList.Find(i1), i3);
        todoList.AddAfter(todoList.Find(i1), i4);
        // 显示所有项目.
        foreach (TodoItem tdi in todoList)
```

```
        {
            Console.WriteLine(tdi.Name + " : " + tdi.Comment);
        }
        // 显示链表内的第二个节点的信息
        Console.WriteLine("todoList.First.Next.Value.Name == " +
                          todoList.First.Next.Value.Name);
        // 显示链表内最后一个节点的前一节点信息.
        Console.WriteLine("todoList.Last.Previous.Value.Name == " +
                          todoList.Last.Previous.Value.Name);
}
```

这个方法的输出结果如下：

```
    buy door : Should be done first
    assemble door : Should be done second
    paint door : Should be done third
    hang door : Should be done last
    todoList.First.Value.Name == buy door
    todoList.First.Next.Value.Name == assemble door
todoList.Last.Previous.Value.Name == paint door
```

下面是 TodoItem 类，它只简单地包含了两个字符串_name 和_comment。

```
public class TodoItem
{
    public TodoItem(string name, string comment)
    {
        _name = name;
        _comment = comment;
    }
    private string _name = "";
    private string _comment = "";
    public string Name
    {
        get { return (_name); }
        set { _name = value; }
    }
    public string Comment
    {
        get { return (_comment); }
        set { _comment = value; }
    }
}
```

讨论

LinkedList<T>类在.NET framework 中是一个双向链表。这是因为链表中的每一个节点都包含了前一节点和后一节点的指针。图 4-1 演示了这种结构，图中的每个 node 代表了一个单独的 LinkedListNode<T>对象。

注意图中链表的每个节点（方块）包含了一个指向下一节点的指针（指向右边的箭头）和一个指向前一节点的指针（指向左边的箭头）。相反，单链表只包含指向下一节点的指针，它没有指向前一节点的指针。

在 LinkedList 类中，前一节点通过访问 Previous 属性获得，后一节点通过访问 Next 属性获得。链表中的第一个节点的 Previous 属性总是返回 null 值。两样，最后一个节点的 Next 属性也是返回 null 值。

链表中的每个节点（图 4-1 中用方块表示）实际上都是一个 LinkedListNode<T>对象。所以 LinkedList<T>对象实际上是由一组 LinkedListNode<T>对象组成，所有这些 LinkedListNode<T>对象都包含了访问下一个和前一个 LinkedListNode<T>对象的属性。LinkedListNode<T>中所包含的对象可以通过 Value 属性访问。除了这些属性外，LinkedListNode<T>对象还有一个属性叫 List，可以用它来访问所属的 LinkedList<T>对象。

性能是我们非常关心的一个问题，List<T>类的性能优越于 LinkedList<T>类。一般情况下在 List<T>内添加和删除节点要比在 LinkedList<T>内进行同样的操作快。对比 List<T>.Add 方法和 LinkedList<T>类的 Add*方法（译者注：之所以是 Add*方法是因为 LinkedList<T>中的添加方法有：AddAfter、AddBefore、AddFirst、AddLast），导到性能上的差异并非因为添加操作本身，而是 LinkedList<T>在垃圾回收时的压力。List<T>的数据本质上是存放在托管堆中的一个大容量数组之上，然而 LinkedList<T>有可能会把它的节点存放在托管堆的每个角落。这使得强制垃圾回收在处理托管堆中的 LinkedList<T>节点对象时需要花费更多的力气。需要注意，List<T>.Insert 方法可能会比 LinkedList<T>中的任一个 Add*方法要慢，但这取决于对象在 List<T>的哪个位置插入。当在某个点插入一人新元素时，Insert 方法必须移动它后面的所有元素一个位置。如果新元素插入到 List<T>的尾部，移动元素所花费的开销比起垃圾回收所花费的开销就可以忽略不计了。

List<T>另外一个胜于 LinkedList<T>的地方是可以使用索引访问。在 List<T>中，您可以使用索引器并通过索引值来访问指定位置的某个元素。但在 LinkedList<T>中就没有这么令人愉快了。在 LinkedList<T>类中，您必须使用每个 LinkedListNode<T>中的 Previous 和 Next 属性进行导航，并贯穿整个链表直到找到您所指定的位置。

在搜索一个元素或节点时，List<T>类也比 LinkedList<T>类有速度上的优势。使用 List<T>.BinarySearch 方法在 List<T>内查找元素比在 LinkedList<T>类中使用相应的 Contains、Find、FindLast 方法更快，这是因为 LinkedList<T>的方法执行线性搜索而 List<T>.BinarySearch 方法执行二分查找法。一般条件下，二分查找法利用元素在 List<T>内是按顺序排列的。这使得在进行二分查找之前必须调用 Sort 方法（注意：当添加新元素时，Sort 方法也会在 BinarySearch 方法之前被调用）。利用这些，二分查找将检查列表中的中间那个元素，并询问：你查找的对象是否大于列表中的当前对象？如果是这样，可知目标对象索引值将在当前对象之前。如果不是，则对象索引值在当前索引之后。二分查找算法保持询问，直到找到对象为止。恰恰相反，线性搜索从列表中的第一个元素开始查找指定元素，如果不是，则继续搜索下一个元素，直到在列表中找到相应的元素。

创建一个可以被初始化为空的值类型

问题

您有一个数字类型的变量，用于控制从数据库中获取的数值。数据库可能为这个值返回一个 null 值。您需要一个简洁的方法来存储这个数值，甚至它返回为 null。

解决方案

使用可空类型。有两个创建可空类型的方法。第一种方法是使用？类型修饰符：

```
int? myDBInt = null;
```

第二种方法是使用 Nullable<T>泛型类型：

```
Nullable<int> myDBInt = new Nullable<int>();
```

讨论

本质上，下面两个声明是等价的：

```
int? myDBInt = null;
Nullable<int> myDBInt = new Nullable<int>();
```

两个声明中，myDBInt 都被认为是可空类型并初始化为 null。一个可空类型实现了 InullableValue 接口，它只有两个属性成员：HasValue 和 Value。如果把可空类型设置为 null，HasValue 属性将返回 false，否则将返回 true。如果 HasValue 返回 true，就可以访问 Value 属性以获得可空数据类型里当前存放的值。如果引发了 InvalidOperationException 异常，这是因为此时 Value 属性还未被定义。

另外，测试可空类型可以有两种方法。第一，使用 HasValue 属性如下：

```
    if (myDBInt.HasValue)
        Console.WriteLine("Has a value: " + myDBInt.Value);
    else
Console.WriteLine("Does not have a value (NULL)");
```

第二种方法是跟 null 对比：

```
    if (myDBInt != null)
        Console.WriteLine("Has a value: " + myDBInt.Value);
    else
Console.WriteLine("Does not have a value (NULL)");
```

两种方法都可以让人接受。

当需要把可空类型转换为非可空类型时，转换操作将正常进行，如果可空类型被设置为 null 就会引发一个 InvalidOperationException 异常。当把一个非可空类型转换为可空类型时，转换操作将正常运行，不会引发 InvalidOperationException 异常，非可空类型永远不会为 null。

需要提防的是可空类型在进行比较运算的时候。例如执行下列代码时：

```
    if (myTempDBInt < 100)
        Console.WriteLine("myTempDBInt < 100");
    else
        Console.WriteLine("myTempDBInt >= 100");
```

“myTempDBInt < 100”这句代码明显有错。为了修正它，您不得不检查 myTempDBInt 是否为空。如果不是，才能执行 if 语句里的代码块：

```
if (myTempDBInt != null)
    {
        if (myTempDBInt < 100)
            Console.WriteLine("myTempDBInt < 100");
        else
            Console.WriteLine("myTempDBInt >= 100");
    }
    else
    {
        // 在这里处理空值
 }
```

另外一个有趣的事情是您可以象使用一般数字类型一样使用可空类型，如：

```
    int? DBInt = 10;
    int Value = 2;
    int? Result = DBInt + Value; // Result == 12
```

如果表达式中的可空类型是一个 null 值，则表达式的结果为 null，但如果没有可空类型的值为 null，运算符会把它当成一般类型。如上例中的 DBInt 为 null，则 Result 的结果也为 null

3.2.5、反转SortedList里的内容

反转 Sorted List 里的内容

问题

您希望在数组和列表类型中可以反转 sorted list 里的内容同时又维持 SortedList 和 SortedList<T>类原来的功能。无论是 SortedList 还是泛型 SortedList<T>类都直接提供了完成这个功能的方法而又不需要重填列表。

解决方案

ReversibleSortedList<TKey, TValue>类提供了这些功能，它基于 SortedList<TKey, TValue>类，所以拥有相同的功能，它提供了额外的功能是很容易反转已排序的列表。

在实例化 ReversibleSortedList<TKey, TValue>之后，键是整数类型，值是字符串类型，一连串无序的数字和它们的文本表达被插入到列表中。这些项目是这样显示的：

```
ReversibleSortedList<int, string> rsl = new ReversibleSortedList<int, string>();
rsl.Add(2, "2");
rsl.Add(5, "5");
rsl.Add(3, "3");
rsl.Add(1, "1");
foreach (KeyValuePair<int, string> kvp in rsl)
{
    Debug.WriteLine("\t" + kvp.Key + "\t" + kvp.Value);
}
```

列表输出显示为按升序排序（默认）：

```
    1    1
    2    2
    3    3
    5    5
```

现在排列顺序通过设置 ReversibleSortedList 的 SortDirection 属性被反转为降序。为了重新排序需要调用 Sort()方法。结果如下：

```
// 转换排序方向.
rsl.Comparer.SortDirection = ListSortDirection.Descending;
// 重排列表.
rsl.Sort();
foreach (KeyValuePair<int, string> kvp in rsl)
{
    Debug.WriteLine("\t" + kvp.Key + "\t" + kvp.Value);
}
```

这一次，输出为降序：

```
    5    5
    3    3
    2    2
    1    1
```

当把一个新项添加进列表，它将按当前的排列顺序被添加进去,但在添加完所有项后马上进行反转，就可以保持列表中元素的顺序。

```
        rsl.Add(4, "4");
        foreach (KeyValuePair<int, string> kvp in rsl)
        {
            Debug.WriteLine("\t" + kvp.Key + "\t" + kvp.Value);
        }
        // 转换排序方向.
        rsl.Comparer.SortDirection = ListSortDirection.Ascending;
        // 重排列表.
        rsl.Sort();
        foreach (KeyValuePair<int, string> kvp in rsl)
        {
            Debug.WriteLine("\t" + kvp.Key + "\t" + kvp.Value);
}
```

可以看到新项即按降序也按升序排列：

```
        5    5
        4    4
        3    3
        2    2
        1    1
        1    1
        2    2
        3    3
        4    4
        5    5
```

ReversibleSortedList<TKey, TValue>包含一个实现了 IComparer<T>接口的嵌套类 SortDirectionComparer<T>。这个类可以在“讨论”这一节中的 ReversibleSortedList<TKey, TValue>代码中看到。一个实现了 IComparer<T>接口的类可以做为 ReversibleSortedList<TKey, TValue>构造方法的参数来改变默认的排序。IComparer<T>接口实现了 Compare 方法：

```
class Program
    {
        public int Compare(T lhs, T rhs)
        {
            int compareResult =
                lhs.ToString().CompareTo(rhs.ToString());
            // 如果为降序，则反转
            if (SortDirection == ListSortDirection.Descending)
                compareResult *= -1;
            return compareResult;
        }
```

```
}
```

Compare 方法使用了 SortDirectionComparer<T>类的 SortDirection 属性来决定项的排序。这个属性在 ReversibleSortedList<TKey, TValue>的内部类 SortDirectionComparer<T>实例中被设置。SortDirection 属性是在构造方法中被设置的，代码如下：

```
 public ReversibleSortedList()
    {
        this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
        this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
        this._size = 0;
        this._sortDirectionComparer = new SortDirectionComparer<TKey>();
        this._currentSortDirection = this._sortDirectionComparer.SortDirection;
}
```

这允许它在指定时间内反转排列顺序，但并没有重排列表中已存在的项。为了实现这个功能，需要在 Reversible-SortedList<TKey, TValue>类中添加一个新的 Sort()方法以重排列表。代码如下：

```
public void Sort()
{
    //检查是否跟现有排序方向相同.
    if (this._currentSortDirection != this._sortDirectionComparer.SortDirection)
    {
        // 如果不同，则进行反转.
        Array.Reverse(this.keys, 0, this._size);
        Array.Reverse(this.values, 0, this._size);
        // 设置当前排序.
        this._currentSortDirection = this._sortDirectionComparer.SortDirection;
     }
}
```

讨论

例 4-3 是 ReversibleSortedList<TKey, TValue>类的所有代码：

(译者注：这个类的代码很恐怖，接近 1300 行，不过代码很规范，感觉应该是商业代码，非常值得借鉴。将来有时间我会专门写文章分析它。)

例 4-3 ReversibleSortedList 类

```
[Serializable, ComVisible(false), DebuggerDisplay("Count = {Count}")]
public class ReversibleSortedList<TKey, TValue> :
        IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>,
```

```
        IEnumerable<KeyValuePair<TKey, TValue>>,
        IDictionary, ICollection, IEnumerable
{
    SortDirectionComparer 类定义#region SortDirectionComparer 类定义
    public class SortDirectionComparer<T> : IComparer<T>
    {   //ListSortDirection 枚举，有两个值：
        //Ascending 按升序排列，Descending 按降序排列
        private System.ComponentModel.ListSortDirection _sortDir;
        //构造方法
        public SortDirectionComparer()
        {   //默认为升序
            _sortDir = ListSortDirection.Ascending;
        }
        //重载构造方法
        public SortDirectionComparer(ListSortDirection sortDir)
        {
            _sortDir = sortDir;
        }
        //排序方向属性
        public System.ComponentModel.ListSortDirection SortDirection
        {
            get { return _sortDir; }
            set { _sortDir = value; }
        }
        //实现 IComparer<T>接口的方法
        public int Compare(T lhs, T rhs)
        {
            int compareResult =
                lhs.ToString().CompareTo(rhs.ToString());

            // If order is DESC, reverse this comparison.
            if (SortDirection == ListSortDirection.Descending)
                compareResult *= -1;
            return compareResult;
        }
    }
    #endregion // SortDirectionComparer

    构造方法#region 构造方法
    //类型构造器
    static ReversibleSortedList()
    {
        ReversibleSortedList<TKey, TValue>.emptyKeys = new TKey[0];
        ReversibleSortedList<TKey, TValue>.emptyValues = new TValue[0];
```

```
}
//无参构造方法
public ReversibleSortedList()
{
    this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
    this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
    this._size = 0;
    this._sortDirectionComparer = new SortDirectionComparer<TKey>();
    this._currentSortDirection = this._sortDirectionComparer.SortDirection;
}
//用于指定排序方向的构造方法
public ReversibleSortedList(SortDirectionComparer<TKey> comparer)
    : this()
{
    if (comparer != null)
    {
        this._sortDirectionComparer = comparer;
        this._currentSortDirection = _sortDirectionComparer.SortDirection;
    }
}
//用于指定字典的构造方法
public ReversibleSortedList(IDictionary<TKey, TValue> dictionary)
    : this(dictionary, (SortDirectionComparer<TKey>)null)
{
}
//用于指定列表容量的构造方法
public ReversibleSortedList(int capacity)
{
    if (capacity < 0)
    {
        throw new ArgumentOutOfRangeException(
            "capacity", "Non-negative number required");
    }
    this.keys = new TKey[capacity];
    this.values = new TValue[capacity];
    this._sortDirectionComparer = new SortDirectionComparer<TKey>();
    this._currentSortDirection = _sortDirectionComparer.SortDirection;
}
//用于指定字典和排序方向的构造方法
public ReversibleSortedList(IDictionary<TKey, TValue> dictionary,
                            SortDirectionComparer<TKey> comparer)
    : this((dictionary != null) ? dictionary.Count : 0, comparer)
{
    if (dictionary == null)
```

```
        {
            throw new ArgumentNullException("dictionary");
        }
        dictionary.Keys.CopyTo(this.keys, 0);
        dictionary.Values.CopyTo(this.values, 0);
        Array.Sort<TKey, TValue>(this.keys, this.values,
                                                this._sortDirectionComparer);
        this._size = dictionary.Count;
    }
    //用于指定容量和排序方向的构造方法
    public ReversibleSortedList(int capacity, SortDirectionComparer<TKey> comparer)
        : this(comparer)
    {
        this.Capacity = capacity;
    }
    #endregion //CTORS

    公有方法#region  公有方法
    //添加元素
    public void Add(TKey key, TValue value)
    {
        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,
                                                        this._sortDirectionComparer);
        if (num1 >= 0)
        {
            throw new ArgumentException("Attempting to add duplicate");
        }
        this.Insert(~num1, key, value);
    }
    //ICollection<KeyValuePair<TKey, TValue>>接口方法实现
    public void Clear()
    {
        this.version++;
        Array.Clear(this.keys, 0, this._size);
        Array.Clear(this.values, 0, this._size);
        this._size = 0;
    }
    //判断是否包含指定键
    public bool ContainsKey(TKey key)
    {
```

```
        return (this.IndexOfKey(key) >= 0);
    }
    //判断是否包含指定值
    public bool ContainsValue(TValue value)
    {
        return (this.IndexOfValue(value) >= 0);
    }

    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
    {
        return new ReversibleSortedList<TKey, TValue>.Enumerator<TKey, TValue>(
                        this);
    }
    //查找指定键
    public int IndexOfKey(TKey key)
    {

        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,
                                                        this._sortDirectionComparer);
        if (num1 < 0)
        {
            return -1;
        }
        return num1;
    }
    //查找指定值
    public int IndexOfValue(TValue value)
    {
        return Array.IndexOf<TValue>(this.values, value, 0, this._size);
    }
    //IDictionary<TKey, TValue>接口方法实现
    public bool Remove(TKey key)
    {
        int num1 = this.IndexOfKey(key);
        if (num1 >= 0)
        {
            this.RemoveAt(num1);
        }
        return (num1 >= 0);
    }
```

```
//移除指定索引元素
public void RemoveAt(int index)
{
    if ((index < 0) || (index >= this._size))
    {
        throw new ArgumentOutOfRangeException("index", "Index out of range");
    }
    this._size--;
    if (index < this._size)
    {
        Array.Copy(this.keys, (int)(index + 1), this.keys, index,
                   (int)(this._size - index));
        Array.Copy(this.values, (int)(index + 1), this.values, index,
                   (int)(this._size - index));
    }
    this.keys[this._size] = default(TKey);
    this.values[this._size] = default(TValue);
    this.version++;
}
//排序
public void Sort()
{
    // 检查是否跟现有排序方向相同.
    if (this._currentSortDirection !=
        this._sortDirectionComparer.SortDirection)
    {
        // 如果不同，则进行反转.
        Array.Reverse(this.keys, 0, this._size);
        Array.Reverse(this.values, 0, this._size);
        // 设置当前排序.
        this._currentSortDirection = this._sortDirectionComparer.SortDirection;
    }
}
//剪除多余空间
public void TrimExcess()
{
    int num1 = (int)(this.keys.Length * 0.9);
    if (this._size < num1)
    {
        this.Capacity = this._size;
    }
}
//获取指定键的值
public bool TryGetValue(TKey key, out TValue value)
```

```
{
    int num1 = this.IndexOfKey(key);
    if (num1 >= 0)
    {
        value = this.values[num1];
        return true;
    }
    value = default(TValue);
    return false;
}

#endregion // Public Methods

私有方法#region  私有方法
private void EnsureCapacity(int min)
{
    int num1 = (this.keys.Length == 0) ? 4 : (this.keys.Length * 2);
    if (num1 < min)
    {
        num1 = min;
    }
    this.InternalSetCapacity(num1, false);
}
//返回指定索引的值
private TValue GetByIndex(int index)
{
    if ((index < 0) || (index >= this._size))
    {
        throw new ArgumentOutOfRangeException("index", "Index out of range");
    }
    return this.values[index];
}
//返回指定索引的键
private TKey GetKey(int index)
{
    if ((index < 0) || (index >= this._size))
    {
        throw new ArgumentOutOfRangeException("index", "Index out of range");
    }
    return this.keys[index];
}

private KeyList<TKey, TValue> GetKeyListHelper()
{
```

```
        if (this.keyList == null)
        {
            this.keyList = new KeyList<TKey, TValue>(this);
        }
        return this.keyList;
    }

    private ValueList<TKey, TValue> GetValueListHelper()
    {
        if (this.valueList == null)
        {
            this.valueList = new ValueList<TKey, TValue>(this);
        }
        return this.valueList;
    }
    //在指定位置插入元素
    private void Insert(int index, TKey key, TValue value)
    {
        if (this._size == this.keys.Length)
        {
            this.EnsureCapacity(this._size + 1);
        }
        if (index < this._size)
        {
            Array.Copy(this.keys, index, this.keys, (int)(index + 1),
                       (int)(this._size - index));
            Array.Copy(this.values, index, this.values, (int)(index + 1),
                       (int)(this._size - index));
        }
        this.keys[index] = key;
        this.values[index] = value;
        this._size++;
        this.version++;
    }

    private void InternalSetCapacity(int value, bool updateVersion)
    {
        if (value != this.keys.Length)
        {
            if (value < this._size)
            {
                throw new ArgumentOutOfRangeException(
                    "value", "Too small capacity");
            }
```

```
                if (value > 0)
                {
                    TKey[] localArray1 = new TKey[value];
                    TValue[] localArray2 = new TValue[value];
                    if (this._size > 0)
                    {
                        Array.Copy(this.keys, 0, localArray1, 0, this._size);
                        Array.Copy(this.values, 0, localArray2, 0, this._size);
                    }
                    this.keys = localArray1;
                    this.values = localArray2;
                }
                else
                {
                    this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
                    this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
                }
                if (updateVersion)
                {
                    this.version++;
                }
            }
        }

        private static bool IsCompatibleKey(object key)
        {
            if (key.Equals(null))
            {
                throw new ArgumentNullException("key");
            }
            return (key is TKey);
        }
        //显式接口成员实现
        void ICollection<KeyValuePair<TKey, TValue>>.Add(
                                                    KeyValuePair<TKey,  TValue>
keyValuePair)
        {
            this.Add(keyValuePair.Key, keyValuePair.Value);
        }
        //显式接口成员实现
        bool ICollection<KeyValuePair<TKey, TValue>>.Contains(
                                                        KeyValuePair<TKey,  TValue>
keyValuePair)
        {
```

```
            int num1 = this.IndexOfKey(keyValuePair.Key);
            if ((num1 >= 0) && EqualityComparer<TValue>.Default.Equals(this.values[num1],

keyValuePair.Value))
            {
                return true;
            }
            return false;
        }
        //显式接口成员实现
        void ICollection<KeyValuePair<TKey, TValue>>.CopyTo(
            KeyValuePair<TKey,TValue>[] array, int arrayIndex)
        {
            if (array == null)
            {
                throw new ArgumentNullException("array");
            }
            if ((arrayIndex < 0) || (arrayIndex > array.Length))
            {
                throw new ArgumentOutOfRangeException(
                        "arrayIndex", "Need a non-negative number");
            }
            if ((array.Length - arrayIndex) < this.Count)
            {
                throw new ArgumentException("ArrayPlusOffTooSmall");
            }
            for (int num1 = 0; num1 < this.Count; num1++)
            {
                KeyValuePair<TKey, TValue> pair1;
                pair1 = new KeyValuePair<TKey, TValue>(
                                this.keys[num1], this.values[num1]);
                array[arrayIndex + num1] = pair1;
            }
        }
        //显式接口成员实现
        bool ICollection<KeyValuePair<TKey, TValue>>.Remove(
            KeyValuePair<TKey, TValue> keyValuePair)
        {
            int num1 = this.IndexOfKey(keyValuePair.Key);
            if ((num1 >= 0) && EqualityComparer<TValue>.Default.Equals(
                this.values[num1], keyValuePair.Value))
            {
                this.RemoveAt(num1);
                return true;
```

```
        }
        return false;
    }

    IEnumerator<KeyValuePair<TKey, TValue>>
        IEnumerable<KeyValuePair<TKey, TValue>>.GetEnumerator()
    {
        return new ReversibleSortedList<TKey, TValue>.Enumerator<TKey, TValue>(
                    this);
    }
    //显式接口成员实现
    void ICollection.CopyTo(Array array, int arrayIndex)
    {
        if (array == null)
        {
            throw new ArgumentNullException("array");
        }
        if (array.Rank != 1)
        {
            throw new ArgumentException(
                "MultiDimensional array copies are not supported");
        }
        if (array.GetLowerBound(0) != 0)
        {
            throw new ArgumentException("A non-zero lower bound was provided");
        }
        if ((arrayIndex < 0) || (arrayIndex > array.Length))
        {
            throw new ArgumentOutOfRangeException(
                "arrayIndex", "Need non negative number");
        }
        if ((array.Length - arrayIndex) < this.Count)
        {
            throw new ArgumentException("Array plus the offset is too small");
        }
        KeyValuePair<TKey, TValue>[] pairArray1 =
            array as KeyValuePair<TKey, TValue>[];
        if (pairArray1 != null)
        {
            for (int num1 = 0; num1 < this.Count; num1++)
            {
                pairArray1[num1 + arrayIndex] =
                    new KeyValuePair<TKey, TValue>(this.keys[num1],
                    this.values[num1]);
```

```
                }
            }
            else
            {
                object[] objArray1 = array as object[];
                if (objArray1 == null)
                {
                    throw new ArgumentException("Invalid array type");
                }
                try
                {
                    for (int num2 = 0; num2 < this.Count; num2++)
                    {
                        objArray1[num2 + arrayIndex] =
                                new KeyValuePair<TKey, TValue>(this.keys[num2],

this.values[num2]);
                    }
                }
                catch (ArrayTypeMismatchException)
                {
                    throw new ArgumentException("Invalid array type");
                }
            }
        }
        //显式接口成员实现
        void IDictionary.Add(object key, object value)
        {
            ReversibleSortedList<TKey, TValue>.VerifyKey(key);
            ReversibleSortedList<TKey, TValue>.VerifyValueType(value);
            this.Add((TKey)key, (TValue)value);
        }
        //显式接口成员实现
        bool IDictionary.Contains(object key)
        {
            if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
            {
                return this.ContainsKey((TKey)key);
            }
            return false;
        }
        //显式接口成员实现
        IDictionaryEnumerator IDictionary.GetEnumerator()
        {
```

```
        return new ReversibleSortedList<TKey, TValue>.Enumerator<TKey, TValue>(
                this);
    }
    //显式接口成员实现
    void IDictionary.Remove(object key)
    {
        if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
        {
            this.Remove((TKey)key);
        }
    }
    //显式接口成员实现
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new ReversibleSortedList<TKey, TValue>.Enumerator<TKey, TValue>(
                this);
    }


    private static void VerifyKey(object key)
    {
        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        if (!(key is TKey))
        {
            throw new ArgumentException(
                "Argument passed is of wrong type", "key");
        }
    }

    private static void VerifyValueType(object value)
    {
        if (!(value is TValue) && ((value != null) || typeof(TValue).IsValueType))
        {
            throw new ArgumentException(
                "Argument passed is of wrong type", "value");
        }
    }
    #endregion // Private methods

    Public Properties#region Public Properties
    public int Capacity
```

```
{
    get
    {
        return this.keys.Length;
    }
    set
    {
        this.InternalSetCapacity(value, true);
    }
}

public SortDirectionComparer<TKey> Comparer
{
    get
    {
        return this._sortDirectionComparer;
    }
}

public int Count
{
    get
    {
        return this._size;
    }
}

public TValue this[TKey key]
{
    get
    {
        TValue local1;
        int num1 = this.IndexOfKey(key);
        if (num1 >= 0)
        {
            return this.values[num1];
        }
        else
        {
            //throw new KeyNotFoundException();
            local1 = default(TValue);
            return local1;
        }
    }
```

```
            set
            {
                if (key == null)
                {
                    throw new ArgumentNullException("key");
                }
                int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,

this._sortDirectionComparer);
                if (num1 >= 0)
                {
                    this.values[num1] = value;
                    this.version++;
                }
                else
                {
                    this.Insert(~num1, key, value);
                }
            }
        }

        public IList<TKey> Keys
        {
            get
            {
                return this.GetKeyListHelper();
            }
        }

        public IList<TValue> Values
        {
            get
            {
                return this.GetValueListHelper();
            }
        }
        #endregion // Public Properties

        Private Properties#region Private Properties
        bool ICollection<KeyValuePair<TKey, TValue>>.IsReadOnly
        {
            get
            {
                return false;
```

```
        }
    }

    ICollection<TKey> IDictionary<TKey, TValue>.Keys
    {
        get
        {
            return this.GetKeyListHelper();
        }
    }

    ICollection<TValue> IDictionary<TKey, TValue>.Values
    {
        get
        {
            return this.GetValueListHelper();
        }
    }

    bool ICollection.IsSynchronized
    {
        get
        {
            return false;
        }
    }

    object ICollection.SyncRoot
    {
        get
        {
            return this;
        }
    }

    bool IDictionary.IsFixedSize
    {
        get
        {
            return false;
        }
    }

    bool IDictionary.IsReadOnly
```

```
{
    get
    {
        return false;
    }
}

object IDictionary.this[object key]
{
    get
    {
        if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
        {
            int num1 = this.IndexOfKey((TKey)key);
            if (num1 >= 0)
            {
                return this.values[num1];
            }
        }
        return null;
    }
    set
    {
        ReversibleSortedList<TKey, TValue>.VerifyKey(key);
        ReversibleSortedList<TKey, TValue>.VerifyValueType(value);
        this[(TKey)key] = (TValue)value;
    }
}

ICollection IDictionary.Keys
{
    get
    {
        return this.GetKeyListHelper();
    }
}

ICollection IDictionary.Values
{
    get
    {
        return this.GetValueListHelper();
    }
}
```

```
#endregion // Private properties

Fields#region Fields
private const int _defaultCapacity = 4;
private int _size;
//private IComparer<TKey> comparer;
private static TKey[] emptyKeys;
private static TValue[] emptyValues;
private KeyList<TKey, TValue> keyList;
private TKey[] keys;
private ValueList<TKey, TValue> valueList;
private TValue[] values;
private int version;
// Declare comparison object.
private SortDirectionComparer<TKey> _sortDirectionComparer = null;
// Default to ascending.
private ListSortDirection _currentSortDirection = ListSortDirection.Descending;
#endregion

Nested Types#region Nested Types

Enumerator K, V#region Enumerator <K, V>
[Serializable, StructLayout(LayoutKind.Sequential)]
private struct Enumerator<K, V> : IEnumerator<KeyValuePair<K, V>>, IDisposable,
    IDictionaryEnumerator, IEnumerator
{
    private ReversibleSortedList<K, V> _ReversibleSortedList;
    private K key;
    private V value;
    private int index;
    private int version;
    internal Enumerator(ReversibleSortedList<K, V> ReversibleSortedList)
    {
        this._ReversibleSortedList = ReversibleSortedList;
        this.index = 0;
        this.version = this._ReversibleSortedList.version;
        this.key = default(K);
        this.value = default(V);
    }
    public void Dispose()
    {
        this.index = 0;
        this.key = default(K);
        this.value = default(V);
```

```
        }
        object IDictionaryEnumerator.Key
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "Enumeration operation cannot occur.");
                }
                return this.key;
            }
        }
        public bool MoveNext()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                        "Enumeration failed version check");
            }
            if (this.index < this._ReversibleSortedList.Count)
            {
                this.key = this._ReversibleSortedList.keys[this.index];
                this.value = this._ReversibleSortedList.values[this.index];
                this.index++;
                return true;
            }
            this.index = this._ReversibleSortedList.Count + 1;
            this.key = default(K);
            this.value = default(V);
            return false;
        }
        DictionaryEntry IDictionaryEnumerator.Entry
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "Enumeration operation cannot happen.");
                }
                return new DictionaryEntry(this.key, this.value);
```

```
        }
    }
    public KeyValuePair<K, V> Current
    {
        get
        {
            return new KeyValuePair<K, V>(this.key, this.value);
        }
    }
    object IEnumerator.Current
    {
        get
        {
            if ((this.index == 0) ||
                (this.index == (this._ReversibleSortedList.Count + 1)))
            {
                throw new InvalidOperationException(
                        "Enumeration operation cannot occur");
            }
            return new DictionaryEntry(this.key, this.value);
        }
    }
    object IDictionaryEnumerator.Value
    {
        get
        {
            if ((this.index == 0) ||
                (this.index == (this._ReversibleSortedList.Count + 1)))
            {
                throw new InvalidOperationException(
                        "Enumeration operation cannot occur");
            }
            return this.value;
        }
    }
    void IEnumerator.Reset()
    {
        if (this.version != this._ReversibleSortedList.version)
        {
            throw new InvalidOperationException(
                    "Enumeration version check failed");
        }
        this.index = 0;
        this.key = default(K);
```

```
                this.value = default(V);
            }
        }
        #endregion // Enumerator <K, V>

        KeyListK,V#region KeyList<K,V>
        [Serializable]
        private sealed class KeyList<K, V> : IList<K>, ICollection<K>,
                                                IEnumerable<K>,        ICollection,
IEnumerable
        {
            // Methods
            internal KeyList(ReversibleSortedList<K, V> dictionary)
            {
                this._dict = dictionary;
            }

            public void Add(K key)
            {
                throw new NotSupportedException("Add is unsupported");
            }

            public void Clear()
            {
                throw new NotSupportedException("Clear is unsupported");
            }

            public bool Contains(K key)
            {
                return this._dict.ContainsKey(key);
            }

            public void CopyTo(K[] array, int arrayIndex)
            {
                Array.Copy(this._dict.keys, 0, array, arrayIndex, this._dict.Count);
            }

            public IEnumerator<K> GetEnumerator()
            {
                return new
                        ReversibleSortedList<K, V>.ReversibleSortedListKeyEnumerator(
                                                        this._dict);
            }
```

```
        public int IndexOf(K key)
        {
            if (key == null)
            {
                throw new ArgumentNullException("key");
            }
            int num1 = Array.BinarySearch<K>(this._dict.keys, 0,
                                                        this._dict.Count, key,

this._dict._sortDirectionComparer);
            if (num1 >= 0)
            {
                return num1;
            }
            return -1;
        }

        public void Insert(int index, K value)
        {
            throw new NotSupportedException("Insert is unsupported");
        }

        public bool Remove(K key)
        {
            //throw new NotSupportedException("Remove is unsupported");
            return false;
        }

        public void RemoveAt(int index)
        {
            throw new NotSupportedException("RemoveAt is unsupported");
        }

        void ICollection.CopyTo(Array array, int arrayIndex)
        {
            if ((array != null) && (array.Rank != 1))
            {
                throw new ArgumentException(
                    "MultiDimensional arrays are not unsupported");
            }
            try
            {
                Array.Copy(this._dict.keys, 0, array, arrayIndex,
                        this._dict.Count);
```

```
                }
                catch (ArrayTypeMismatchException atme)
                {
                        throw new ArgumentException("InvalidArrayType", atme);
                }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
                return new
                        ReversibleSortedList<K, V>.ReversibleSortedListKeyEnumerator(

this._dict);
        }

        // Properties
        public int Count
        {
                get
                {
                        return this._dict._size;
                }
        }

        public bool IsReadOnly
        {
                get
                {
                        return true;
                }
        }

        public K this[int index]
        {
                get
                {
                        return this._dict.GetKey(index);
                }
                set
                {
                        throw new NotSupportedException("Set is an unsupported operation");
                }
        }
```

```
        bool ICollection.IsSynchronized
        {
            get
            {
                return false;
            }
        }

        object ICollection.SyncRoot
        {
            get
            {
                return this._dict;
            }
        }


        // Fields
        private ReversibleSortedList<K, V> _dict;
    }
    #endregion // KeyList<K,V>

    ReversibleSortedListKeyEnumerator definition#region ReversibleSortedListKeyEnumerator definition
    [Serializable]
    private sealed class ReversibleSortedListKeyEnumerator : IEnumerator<TKey>,
                                                             IDisposable,
                                                             IEnumerator
    {
        // Methods
        internal ReversibleSortedListKeyEnumerator(
                ReversibleSortedList<TKey, TValue> ReversibleSortedList)
        {
            this._ReversibleSortedList = ReversibleSortedList;
            this.version = ReversibleSortedList.version;
        }

        public void Dispose()
        {
            this.index = 0;
            this.currentKey = default(TKey);
        }

        public bool MoveNext()
```

```
{
    if (this.version != this._ReversibleSortedList.version)
    {
        throw new InvalidOperationException(
            "Enumeration failed version check");
    }
    if (this.index < this._ReversibleSortedList.Count)
    {
        this.currentKey = this._ReversibleSortedList.keys[this.index];
        this.index++;
        return true;
    }
    this.index = this._ReversibleSortedList.Count + 1;
    this.currentKey = default(TKey);
    return false;
}

void IEnumerator.Reset()
{
    if (this.version != this._ReversibleSortedList.version)
    {
        throw new InvalidOperationException(
            "Enumeration failed version check");
    }
    this.index = 0;
    this.currentKey = default(TKey);
}


// Properties
public TKey Current
{
    get
    {
        return this.currentKey;
    }
}

object IEnumerator.Current
{
    get
    {
        if ((this.index == 0) || (this.index ==
                (this._ReversibleSortedList.Count + 1)))
```

```
                {
                    throw new InvalidOperationException(
                            "Enumeration operation could not occur");
                }
                return this.currentKey;
            }
        }


        // Fields
        private ReversibleSortedList<TKey, TValue> _ReversibleSortedList;
        private TKey currentKey;
        private int index;
        private int version;
    }
    #endregion //ReversibleSortedListKeyEnumerator definition

ReversibleSortedListValueEnumerator definition#region ReversibleSortedListValueEnumerator
definition
    [Serializable]
    private sealed class ReversibleSortedListValueEnumerator : IEnumerator<TValue>,
                                                               IDisposable,
                                                               IEnumerator
    {
        // Methods
        internal ReversibleSortedListValueEnumerator(
                        ReversibleSortedList<TKey, TValue> ReversibleSortedList)
        {
            this._ReversibleSortedList = ReversibleSortedList;
            this.version = ReversibleSortedList.version;
        }

        public void Dispose()
        {
            this.index = 0;
            this.currentValue = default(TValue);
        }

        public bool MoveNext()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "Enumeration failed version check");
```

```
            }
            if (this.index < this._ReversibleSortedList.Count)
            {
                this.currentValue = this._ReversibleSortedList.values[this.index];
                this.index++;
                return true;
            }
            this.index = this._ReversibleSortedList.Count + 1;
            this.currentValue = default(TValue);
            return false;
        }

        void IEnumerator.Reset()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "Enumeration failed version check");
            }
            this.index = 0;
            this.currentValue = default(TValue);
        }


        // Properties
        public TValue Current
        {
            get
            {
                return this.currentValue;
            }
        }

        object IEnumerator.Current
        {
            get
            {
                if ((this.index == 0) || (this.index ==
                        (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "Enumeration operation could not occur");
                }
                return this.currentValue;
```

```
                }
            }


            // Fields
            private ReversibleSortedList<TKey, TValue> _ReversibleSortedList;
            private TValue currentValue;
            private int index;
            private int version;
        }
        #endregion //ReversibleSortedListValueEnumerator

        ValueList K, V definition#region ValueList <K, V> definition
        [Serializable]
        private sealed class ValueList<K, V> : IList<V>, ICollection<V>,
                                                IEnumerable<V>,      ICollection,
IEnumerable
        {
            // Methods
            internal ValueList(ReversibleSortedList<K, V> dictionary)
            {
                this._dict = dictionary;
            }

            public void Add(V key)
            {
                throw new NotSupportedException("Add is not supported");
            }

            public void Clear()
            {
                throw new NotSupportedException("Clear is not supported");
            }

            public bool Contains(V value)
            {
                return this._dict.ContainsValue(value);
            }

            public void CopyTo(V[] array, int arrayIndex)
            {
                Array.Copy(this._dict.values, 0, array, arrayIndex, this._dict.Count);
            }
```

```
        public IEnumerator<V> GetEnumerator()
        {
            return new
                    ReversibleSortedList<K, V>.ReversibleSortedListValueEnumerator(

this._dict);
        }

        public int IndexOf(V value)
        {
            return Array.IndexOf<V>(this._dict.values, value, 0, this._dict.Count);
        }

        public void Insert(int index, V value)
        {
            throw new NotSupportedException("Insert is not supported");
        }

        public bool Remove(V value)
        {
            //throw new NotSupportedException("Remove is not supported");
            return false;
        }

        public void RemoveAt(int index)
        {
            throw new NotSupportedException("RemoveAt is not supported");
        }

        void ICollection.CopyTo(Array array, int arrayIndex)
        {
            if ((array != null) && (array.Rank != 1))
            {
                throw new ArgumentException(
                    "MultiDimensional arrays not supported");
            }
            try
            {
                Array.Copy(this._dict.values, 0, array, arrayIndex,
                            this._dict.Count);
            }
            catch (ArrayTypeMismatchException atme)
            {
                throw new ArgumentException("Invalid array type", atme);
```

```
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return new
                ReversibleSortedList<K, V>.ReversibleSortedListValueEnumerator(
                                                                    this._dict);
    }


    // Properties
    public int Count
    {
        get
        {
            return this._dict._size;
        }
    }

    public bool IsReadOnly
    {
        get
        {
            return true;
        }
    }

    public V this[int index]
    {
        get
        {
            return this._dict.GetByIndex(index);
        }
        set
        {
            throw new NotSupportedException("Set by indexer is not supported");
        }
    }

    bool ICollection.IsSynchronized
    {
        get
        {
```

```
                return false;
            }
        }

        object ICollection.SyncRoot
        {
            get
            {
                return this._dict;
            }
        }

        // Fields
        private ReversibleSortedList<K, V> _dict;
    }
    #endregion // ValueList <TKey, TValue> definition
    #endregion // Nested types
}
```

在 SortedList 混合使用了数组和列表语法，这使得以任一方式访问数据变得非常容易。ReversibleSortedList<T>中数据也可以使用键/值对或索引来访问。和 SortedList 一样，它不允许重复键。另外不管值是引用类型还是值类型都可以为 null，但键不行。ReversibleSortedList<T>的默认容量是 16，这和 SortedList 是一样的。里面的项可以使用 foreach 循环进行迭代，它返回 KeyValuePair，但这是只读的，迭代语法禁止在读取列表时进行元素的更新或删除，否则就是无效的迭代器。

3.2.6、创建只读集合及使用相应的泛型版本替换Hashtable

使用泛型创建只读集合

问题

您希望类中的一个集合里的信息可以被外界访问，但不希望用户改变这个集合。

解决方案

使用 ReadOnlyCollection<T>包装就很容易实现只读的集合类。例子如，Lottery 类包含了中奖号码，它可以被访问，但不允许被改变：

```
public class Lottery
    {
        // 创建一个列表.
        List<int> _numbers = null;
        public Lottery()
        {
            // 初始化内部列表
```

```
            _numbers = new List<int>(5);
            // 添加值
            _numbers.Add(17);
            _numbers.Add(21);
            _numbers.Add(32);
            _numbers.Add(44);
            _numbers.Add(58);
        }
        public ReadOnlyCollection<int> Results
        {
            // 返回一份包装后的结果
            get { return new ReadOnlyCollection<int>(_numbers); }
        }
}
```

Lottery 有一个内部的 List<int>，它包含了在构造方法中被填的中奖号码。有趣的部分是它有一个公有属性叫 Results，通过返回的 ReadOnlyCollection<int>类型可以看到其中的中奖号码，从而使用户通过返回的实例来使用它。

如果用户试图设置集合中的一个值，将引发一个编译错误：

```
    Lottery tryYourLuck = new Lottery();
    // 打印结果.
    for (int i = 0; i < tryYourLuck.Results.Count; i++)
    {
        Console.WriteLine("Lottery Number " + i + " is " + tryYourLuck.Results[i]);
    }
    // 改变中奖号码!
    tryYourLuck.Results[0]=29;
    //最后一行引发错误：// Error 26 // Property or indexer
    // 'System.Collections.ObjectModel.ReadOnlyCollection<int>.this[int]'
    // cannot be assigned to -- it is read only
```

讨论

ReadOnlyCollection 的主要优势是使用上的灵活性，可以在任何支持 IList 或 IList<T>的集合中把它做为接口使用。ReadOnlyCollection 还可以象这样包装一般数组：

```
int [] items = new int[3];
    items[0]=0;
    items[1]=1;
    items[2]=2;
new ReadOnlyCollection<int>(items);
```

这为类的只读属性的标准化提供了一种方法，并使得类库使用人员习惯于这种简单的只读属性返回类型。

使用相应的泛型版本替换 Hashtable

问题

您希望通过使用相应的泛型版本替换所有 Hashtable 来增强应用程序性能并使得代码更为易读。当您发现这些数据结构中存放结构体和值类型会导致装箱/拆箱操作，这就变得非常有必要了。

解决方案

替换所有已存在的 System.Collections.Hashtable 类为速度更快的 System.Collections.Generic.Dictionary 泛型类。

这有一个使用 System.Collections.Hashtable 对象的简单例子：

```
public static void UseNonGenericHashtable()
    {
        // 创建并填充一个 Hashtable.
        Hashtable numbers = new Hashtable();
        numbers.Add(1, "one");      // 键会导致装箱操作
        numbers.Add(2, "two");      // 键会导致装箱操作

        // 在 Hashtable 显示所有的键/值对.
        // 在每次迭代中都会因为键导致一个拆箱操作
        foreach (DictionaryEntry de in numbers)
        {
            Console.WriteLine("Key: " + de.Key + "\tValue: " + de.Value);
        }
        numbers.Clear();
}
```

下面是相同的代码使用了 System.Collections.Generic.Dictionary<T,U>对象：

```
public static void UseGenericDictionary()
    {
        // 创建并填充字典.
        Dictionary<int, string> numbers = new Dictionary<int, string>();
        numbers.Add(1, "one");
        numbers.Add(2, "two");
        // 显示字典中的所有键值对.
        foreach (KeyValuePair<int, string> kvp in numbers)
        {
            Console.WriteLine("Key: " + kvp.Key + "\tValue: " + kvp.Value);
        }
```

```
        numbers.Clear();
}
```

讨论

对于应用程序中简单的 Hashtable 实现，这种替换将十分容易。但有些地方需要注意，如泛型 Dictionary 类没有实现 ICloneable 接口，而 Hashtable 类实现了。

表 4-2 显示了两个类中的等价成员：

表 4-2 Hashtable 和泛型 Dictionary 类的等价成员

Hashtable 类的成员	泛型 Dictionary 类的相应成员
N/A	Comparer 属性
Count属性	Count属性
IsFixedSize属性	((IDictionary)myDict).IsFixedSize
IsReadOnly属性	((IDictionary)myDict).IsReadOnly
IsSynchronized属性	((IDictionary)myDict).IsSynchronized
Item属性	Item属性
Keys属性	Keys属性
SyncRoot属性	((IDictionary)myDict).SyncRoot
Values属性	Values属性
Add 方法	Add方法
Clear方法	Clear方法
Clone方法	在重载构造方法中接收一个 IDictionary<T,U> 类型
Contains方法	ContainsKey方法
ContainsKey方法	ContainsKey方法
ContainsValue方法	ContainsValue方法
CopyTo方法	((ICollection)myDict).CopyTo(arr,0)
Remove方法	Remove方法
Synchronized static方法	lock(myDictionary.SyncRoot) {…}
N/A	TRyGetValue方法

表 4-2 中，并非所有的 Hashtable 和 Dictionary 的成员都一一对应。我们从属性开始，注意，只有 Count，Keys，Values 和 Item 属性在两个类中都存在。为了弥补 Dictionary 中缺少的属性，需要把它转化为 IDictionary 类型。下面的代码演示了如果进行这些转换以获得缺少的属性：

```
Dictionary<int, string> numbers = new Dictionary<int, string>();
    Console.WriteLine(((IDictionary)numbers).IsReadOnly);
    Console.WriteLine(((IDictionary)numbers).IsFixedSize);
```

```
    Console.WriteLine(((IDictionary)numbers).IsSynchronized);
Console.WriteLine(((IDictionary)numbers).SyncRoot);
```

注意，由于缺少返回一个泛型字典同步版本的代码，IsSynchronized 属性将总是返回 false。SyncRoot 属性在被调用时总是返回相同的对象。实际上这个属性返回的是 this 指针。微软已经决定移除泛型集合类的创建同步包装的功能。

做为替代，他们推荐使用 lock 关键字锁住整个集合或其他同步对象类型以满足您的需求。

因为在泛型字典类中也缺少了克隆方法(实际是是因为这个类没有实现 ICloneable 接口)，您可以转而使用重载的构造方法来接收一个 IDictionary<T,U>类型：

```
 // 创建并填充字典.
    Dictionary<int, string> numbers = new Dictionary<int, string>();
    numbers.Add(1, "one");
    numbers.Add(2, "two");
    // 显示原字典的键/值对.
    foreach (KeyValuePair<int, string> kvp in numbers)
    {
        Console.WriteLine("Original Key: " + kvp.Key + "\tValue: " + kvp.Value);
    }
    // 克隆字典对象.
    Dictionary<int, string> clonedNumbers = new Dictionary<int, string>(numbers);
    // 显示克隆字典中的键/值对.
    foreach (KeyValuePair<int, string> kvp in numbers)
    {
        Console.WriteLine("Cloned Key: " + kvp.Key + "\tValue: " + kvp.Value);
}
```

还有两个 Dictionary 类中缺少的方法：Contains 和 CopyTo 方法。Contains 方法的功能在 Dictionary 类中很容易被实现。在 Hashtable 类中，Cintains 方法和 ContainsKey 方法有相同的行为，因此您可以在 Dictionary 类中简单地使用 ContainsKey 方法来模拟 Hashtable 类中的 Contains 方法：

```
 // 创建和填充字典.
    Dictionary<int, string> numbers = new Dictionary<int, string>();
    numbers.Add(1, "one");
    numbers.Add(2, "two");
    Console.WriteLine("numbers.ContainsKey(1) == " + numbers.ContainsKey(1));
Console.WriteLine("numbers.ContainsKey(3) == " + numbers.ContainsKey(3));
CopyTo 方法也很容易在 Dictionary 类中被模拟，但需要做一些额外的工作：
    // 创建和填充字典.
    Dictionary<int, string> numbers = new Dictionary<int, string>();
```

```
        numbers.Add(1, "one");
        numbers.Add(2, "two");
        // 显示字典中的所有键/值对.
        foreach (KeyValuePair<int, string> kvp in numbers)
        {
            Console.WriteLine("Key: " + kvp.Key + "\tValue: " + kvp.Value);
        }
        // 创建对象数组来拷贝字典对象中的信息.
        KeyValuePair<int, string>[] objs = new KeyValuePair<int,
string>[numbers.Count];
        // 调用字典中的 CopyTo 方法
        // 把字典中的所有键/值对对象拷贝到 objs 中
        ((IDictionary)numbers).CopyTo(objs, 0);
        // 显示 objs[]中的所有键/值对.
        foreach (KeyValuePair<int, string> kvp in objs)
        {
            Console.WriteLine("Key: " + kvp.Key + "\tValue: " + kvp.Value);
}
```

调用 Dictionary 对象中的 CopyTo 方法需要创建一个 KeyValuePair<T,U>对象数组，它用于在 CopyTo 方法被调用之后，控制字典对象中的所有 KeyValuePair<T,U>对象。接下来 numbers 字典对象被转换为 IDictionary 类型以调用 CopyTo 方法。一旦 CopyTo 方法被调用，objs 数组将包含原 numbers 对象中的所有 KeyValuePair<T,U>对象。注意 objs 数组迭代时使用了 foreach 循环，这点和 numbers 对象是相同的。

3.2.7、在泛型字典类中使用foreach及泛型约束

在泛型字典类中使用 foreach

问题

您希望在实现了 System. Collections.Generic.IDictionary 接口的类型枚举元素，如 System.Collections.Generic.Dictionary 或 System.Collections.Generic.SortedList。

解决方案

最简单的方法是在 foreach 循环中使用 KeyValuePair 结构体：

```
 // 创建字典对象并填充.
        Dictionary<int, string> myStringDict = new Dictionary<int, string>();
        myStringDict.Add(1, "Foo");
        myStringDict.Add(2, "Bar");
        myStringDict.Add(3, "Baz");
        // 枚举并显示所有的键/值对.
        foreach (KeyValuePair<int, string> kvp in myStringDict)
        {
```

```
        Console.WriteLine("key     " + kvp.Key);
        Console.WriteLine("Value " + kvp.Value);
}
```

讨论

非泛型类 System.Collections.Hashtable (对应的泛型版本为 System.Collections.Generic.Dictionary class), System.Collections.CollectionBase 和 System.Collections.SortedList 类支持在 foreach 使用 DictionaryEntry 类型:

```
foreach (DictionaryEntry de in myDict)
    {
        Console.WriteLine("key " + de.Key);
        Console.WriteLine("Value " + de.Value);
}
```

但是 Dictionary 对象支持在 foreach 循环中使用 KeyValuePair<T,U>类型。这是因为 GetEnumerator 方法返回一个 Ienumerator，而它依次返回 KeyValuePair<T,U>类型，而不是 DictionaryEntry 类型。

KeyValuePair<T,U>类型非常合适在 foreach 循环中枚举泛型 Dictionary 类。DictionaryEntry 类型包含的是键和值的 object 对象，而 KeyValuePair<T,U>类型包含的是键和值在创建一个 Dictionary 对象是被定义的原本类型。这提高了性能并减少了代码量，因为您不再需要把键和值转化为它们原来的类型。

类型参数的约束

问题

您希望创建泛型类型时，它的类型参数支持指定接口，如 IDisposable。

解决方案

使用约束强制泛型的类型参数实现一个或多个指定接口：

```
public class DisposableList<T> : IList<T>
        where T : IDisposable
    {
        private List<T> _items = new List<T>();
        // 用于释放列表中的项目的私有方法
        private void Delete(T item)
        {
            item.Dispose();
        }
}
```

DisposableList 只接收实现了 IDisposable 接口的对象做为它的类型实参。这样无论什么

时候，从 DisposableList 对象中移除一个对象时，那个对象的 Dispose 方法总是被调用。这使得您可以很容易的处理存储在 DisposableList 对象中的所有对象。

下面代码演示了 DisposableList 对象的使用：

```
public static void TestDisposableListCls()
    {
        DisposableList<StreamReader> dl = new DisposableList<StreamReader>();
        // 创建一些测试对象.
        StreamReader tr1 = new StreamReader("c:\\boot.ini");
        StreamReader tr2 = new StreamReader("c:\\autoexec.bat");
        StreamReader tr3 = new StreamReader("c:\\config.sys");
        // 在 DisposableList 内添加一些测试对象.
        dl.Add(tr1);
        dl.Insert(0, tr2);
        dl.Add(tr3);
        foreach(StreamReader sr in dl)
        {
            Console.WriteLine("sr.ReadLine() == " + sr.ReadLine());
        }
        // 在元素从 DisposableList 被移除之前将调用它们的 Dispose 方法
        dl.RemoveAt(0);
        dl.Remove(tr1);
        dl.Clear();
}
```

讨论

where 关键字用来约束一个类型参数只能接收满足给定约束的实参。例如，DisposableList 约束所有类型实参 T 必须实现 IDisposable 接口：

```
public class DisposableList<T> : IList<T>

        where T : IDisposable
```

这意味着下面的代码将成功编译：

```
DisposableList<StreamReader> dl = new DisposableList<StreamReader>();
```

但下面的代码不行：

```
DisposableList<string> dl = new DisposableList<string>();
```

这是因为 string 类型没有实现 IDisposable 接口，而 StreamReader 类型实现了。

除了一个或多个指定接口需要被实现外，类型实参还允许其他约束。您可以强制类型实参继承自一个指定类，如 Textreader 类：

```
public class DisposableList<T> : IList<T>

        where T : System.IO.TextReader, IDisposable
```

您也可以决定是否类型实参仅为值类型或引用类型。下面的类声明被约束为只使用值类型：

```
public class DisposableList<T> : IList<T>

            where T : struct
```

这个类型声明为只能使用引用类型：

```
public class DisposableList<T> : IList<T>

            where T : class
```

另外，您也可能会需要一些类型实参实现了公有的默认构造方法：

```
public class DisposableList<T> : IList<T>

            where T : IDisposable, new()
```

使用约束允许您编写只接收部分类型实参的泛型类型。如果本节中的解决方案忽略了 IDisposable 约束，有可能会引发一个编译错误。这是因为并非所有 DisaposableList 类的类型实参都实现了 IDisposable 接口。如果您跳过这个编译期检查，DisaposableList 对象就可能会包含一个没有公有无参的 Dispose 方法的对象。些例中将会引发一个运行期异常。

给泛型指定约束强制类的类型实参进行严格的类型检查，并使得您在编译期发现问题而不是运行期。

3.2.8、初始化泛型类型变量为它们的默认值

初始化泛型变量为它们的默认值

问题

您的泛型类包含一个变量，它的类型和类中定义的类型参数一样。在构造泛型类时，您希望这个变量被初始化为它的默认值。

解决方案

简单地使用 default 关键字把变量初始化为它的默认值：

```
public class DefaultValueExample<T>
    {
        T data = default(T);
        public bool IsDefaultData()
        {
            T temp = default(T);
            if (temp.Equals(data))
            {
                return (true);
            }
            else
            {
                return (false);
            }
        }
        public void SetData(T val)
        {
            data = val;
        }
}
```

下面的代码使用了这个类：

```
public static void ShowSettingFieldsToDefaults()
    {
        DefaultValueExample<int> dv = new DefaultValueExample<int>();
        // 检查成员 data 是否已经被设置为它的默认值；返回 true.
        bool isDefault = dv.IsDefaultData();
        Console.WriteLine("Initial data: " + isDefault);
        // 设置成员 data.
        dv.SetData(100);
        // 再次检查，这次返回 false.
        isDefault = dv.IsDefaultData();
        Console.WriteLine("Set data: " + isDefault);
```

第一次调用 IsDefaultData 返回 true，而第二次调用返回 false，输出如下：

```
Initial data: True

Set data: False
```

讨论

当对一个和泛型的类型参数一样类型的变量进行初始化时，您不能仅仅设置它为 null。当类型参数是一个值类型如 int 或 char 会怎么样呢？这将无法工作，因为值类型不能为 null。您可能会想到可空类型，如 long？或 Nullable<long>可以被设置为 null（参考秘诀 4.7 了解更多关于可空类型的内容）。但编译器无从得知将使用何种类型实参来构造类型。（译者注：这里的意思是编译器并不知道用户使用的是值类型还是引用类型，因为可空类型仅仅是对值类型而言）

default 关键字允许您告诉编译器在编译期将会使用这个变量的默认值。如果类型实参提供了一个数字值（如 int，long，decimal），那么默认值为 0。如果类型实参提供的是引用类型，那么默认值为 null。如果类型实参提供了一个结构体，那么结构体的默认值会把它的每个成员进行初始化：数字类型为 0，引用类型为 null。

4、C# 泛型集合

4.1、版权声明

文章出处：http://www.cnblogs.com/abatei/archive/2008/02/20/1075760.html
文章作者：abatei

4.2、内容详情

4.2.1、泛型的集合接口

集合接口

.NET Framework 为集合的枚举和对比提供了两组标准接口：传统的（非类型安全）和新的泛型类型安全集合。本书只聚焦于新的，类型安全的集合接口，因为它更优越。

您可以声明一些指定了类型的 ICollection 来取代实际类型（如 int 或 string），以用于接口声明中的泛型类型（<T>）。

C++程序员需要注意：C#泛型和 C++中的模板在语法和用法上都很相似。但是，因为泛型在运行期为其指定类型进行扩展，JIT 编译器可以为它们的不同的实例共享一段代码，从而使得在使用 C++模板时有可能出现的代码膨胀问题在这里显著地减少。

关键的泛型集合接口列于表 9-2：

For backward compatibility, C# also provides nongeneric interfaces (e.g., ICollection, IEnumerator), but they aren't considered here because they are obsolescent.
为了向后兼容，C#也提供了非泛型接口（如 IConnection，IEnumerator），但由于慢慢被弃用，这里并不列出他们。

接口	用途
ICollection<T>	泛型集合的基接口
IEnumerator<T> IEnumerable<T>	使用foreach语句枚举整个集合
ICollection<T>	被所有集合实现以提供CopyTo()方法，Count，IsSynchronized和SyncRoot属性
IComparer<T> IComparable<T>	对比集合中的两个对象，使得集合可以被排序
IList<T>	象数组一样使用索引访问集合
IDictionary<K,V>	在基于键/值的集合中使用，如Dictionary

IEnumerable<T>接口

您可以通过实现 IEnumerable<T>接口来使得 ListBoxTest 支持 foreach 语句（见例 9-11）。IEnumerable 只有一个方法：GetEnumerator()，它的工作是返回一个实现了 IEnumerator<T>接口的类。C#语言使用一个新的关键字 yield 来为创建枚举器（enumerator）提供特殊的帮助。

例 9-11 创建 ListBox 的可枚举类

```
using System;
using System.Collections; //译者注：这句原文没有，必须添加
using System.Collections.Generic;
using System.Text;

namespace Enumerable
{
    public class ListBoxTest : IEnumerable<string>
    {
        private string[] strings;
        private int ctr = 0;
        //Enumerable 类可以返回一个枚举器
        public IEnumerator<string> GetEnumerator()
        {
            foreach (string s in strings)
                yield return s;
        }
        /*译者注：泛型 IEnumerable 的定义为
         *public interface IEnumerable<T> : IEnumerable
         * 也就是说，在实现泛型版 IEnumerable 的同时还必须同时实现
         * 非泛型版的 IEnumerable 接口，原文代码并没有这个内容，下面的三行
         * 代码是我添加进去的以使得代码可以直接拷贝并运行*/
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
        //使用字符串数组来初始化 ListBox
        public ListBoxTest(params string[] initialStrings)

        {
            //为 strings 分配空间
            strings = new string[8];
            //拷贝从构造方法传递进来的字符串数组
            foreach (string s in initialStrings)
            {
```

```
                strings[ctr++] = s;
            }
        }
        //在 ListBox 末尾添加一个字符串
        public void Add(string theString)
        {
            strings[ctr] = theString;
            ctr++;
        }
        //允许象数组一样访问，其实就是索引器，如果对索引器有不明白的请访问：
        //http://www.enet.com.cn/eschool/video/c/20.shtml
        public string this[int index]
        {
            get
            {
                if (index < 0 || index >= strings.Length)
                {
                    //处理错误的索引
                    //译者注：原文这里没加代码，我加了一个异常下去
                    throw new ArgumentOutOfRangeException("索引", "索引超出范围");
                }
                return strings[index];
            }
            set
            {
                strings[index] = value;
            }
        }
        //获得拥有字符串的数量
        public int GetNumEneries()
        {
            return ctr;
        }
    }
    public class Tester
    {
        static void Main()
        {
            //创建一个新的 ListBox 并初始化
            ListBoxTest lbt = new ListBoxTest("Hello", "World");
            //添加一些字符串
            lbt.Add("Who");
            lbt.Add("Is");
            lbt.Add("John");
```

```
                lbt.Add("Galt");
                //访问测试
                string subst = "Universe";
                lbt[1] = subst;
                //列出所有字符串
                foreach (string s in lbt)
                {
                    Console.WriteLine("Value: {0}", s);
                }
            }
        }
}
```

输出结果:

```
Value: Hello

Value: Universe

Value: Who

Value: Is

Value: John

Value: Galt

Value:

Value:
```

程序从 Main()开始执行，创建一个新的 ListBoxTest 对象并给构造方法传递了两个字符串。当对象被创建后，将创建一个容纳 8 个元素的数组。如上例所示，之后使用 Add 方法添加了 4 个字符串，并更改了第二个字符串。

这个版本的程序的一个重大变化是调用了 foreach 循环来获得 listbox 中的每个字符串。Foreach 循环自动使用 IEnumerable<T>接口并调用 GetEnumerator()。

GetEnumerator 方法声明为返回一个字符串类型的 IEnumerator：

```
public IEnumerator<string> GetEnumerator()
```

迭代的实现是遍历整个字符串并依次访问每个元素：

```
foreach (string s in strings)
{
    yield return s;
}
```

4.2.2、泛型约束

约束

有的时候，您必须确保添加进泛型列表中的元素拥有某些约束（例如，它们从一个给定的基类继承或它们实现了指定的接口）。在下面的例子中，我们实现了一个简单的可排序的单链表。链表由多个 Node 组成，每个 Node 必须保证添加进去的元素实现了 IComparer 接口。您可以这样声明：

```
public class Node<T> : IComparable<Node<T>> where T : IComparable<T>
```

这句代码定义了一个泛型 Node，它操作类型 T。Node 中的 T 实现了 IComparable<T> 接口，这意味着两个 Node 的 T 可以进行对比。Node 类通过约束（where T : IComparable<T>）来操作那些实现了 IComparable 接口的类型。因此您可以使用任何类型来替代 T，只要那种类型实现了 IComparable 接口

例 9-12 举例说明了完整的接口实现，并进行分析如下。

例 9-12 使用约束

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

namespace UsingConstraints
{
    public class Employee : IComparable<Employee>
    {
        private string name;
        public Employee(string name)
        {
            this.name = name;
        }
        public override string ToString()
        {
            return this.name;
        }
        //实现接口
```

```
        public int CompareTo(Employee rhs)
        {
            return this.name.CompareTo(rhs.name);
        }
        public bool Equals(Employee rhs)
        {
            return this.name == rhs.name;
        }
    }
    //节点必须实现 Node<T>的 IComparable 接口。
    //通过 where 关键字约束 Node 只接收实现了 IComparable 接口的项
    public class Node<T> : IComparable<Node<T>> where T : IComparable<T>
    {
        //成员变量
        private T data;
        private Node<T> next = null; //下一个节点
        private Node<T> prev = null; //前一个节点
        //构造方法
        public Node(T data)
        {
            this.data = data;
        }
        //属性
        public T Data
        {
            get { return this.data; }
        }
        public Node<T> Next
        {
            get { return this.next; }
        }
        public int CompareTo(Node<T> rhs)
        {   //这样使用是因为约束
            return data.CompareTo(rhs.data);
        }
        public bool Equals(Node<T> rhs)
        {
            return this.data.Equals(rhs.data);
        }
        //方法
        public Node<T> Add(Node<T> newNode)
        {   //下面的“我”代表类的当前实例
            if (this.CompareTo(newNode) > 0) //小于我则放在我前面
            {
```

```
                newNode.next = this; //新节点的下一节点指向我
                //如果我有前一个节点，则新节点的前一个节点指向它
                //新节点做为它的下一个节点
                if (this.prev != null)
                {
                    this.prev.next = newNode;
                    newNode.prev = this.prev;
                }
                //设置我的前一个节点为新节点
                this.prev = newNode;
                //从下面的 LinkedList<T>代码可以得知，添加都是从
                //头节点开始判断，只有新节点为头节点时才返回它
                return newNode;
            }
            else //大于等于我则放在我后面
            {
                //如果我有下一个，则跟下一个进行对比
                //这里使用了递归，直到新节点找到比它大的节点为止
                if (this.next != null)
                {
                    this.next.Add(newNode);
                }
                //如果我没有下一个节点，则设置新节点为我的下一个
                //节点，并把它的上一个节点指向我
                else
                {
                    this.next = newNode;
                    newNode.prev = this;
                }
                return this;
            }
        }
        public override string ToString()
        {
            string output = data.ToString();
            if (next != null)
            {    //这里也使用了递归打印链表上的所有元素
                output += ", " + next.ToString();
            }
            return output;
        }
    }
    public class LinkedList<T> where T : IComparable<T>
    {
```

```
//成员变量
private Node<T> headNode = null;
//索引器
public T this[int index]
{
    get
    {   //由于是链表，这里需要从头遍历
        int ctr = 0;
        Node<T> node = headNode;
        while (node != null && ctr <= index)
        {
            if (ctr == index)
            {
                return node.Data;
            }
            else
            {
                node = node.Next;
            }
            ++ctr;
        }
        throw new ArgumentOutOfRangeException();
    }
}
//默认构造方法
public LinkedList()
{
}
//方法
public void Add(T data)
{
    if (headNode == null)
    {
        headNode = new Node<T>(data);
    }
    else
    {
        headNode = headNode.Add(new Node<T>(data));
    }
}
public override string ToString()
{
    if (this.headNode != null)
    {
```

```
                return this.headNode.ToString();
            }
            else
            {
                return string.Empty;
            }
        }
    }
    //测试类
    class Test
    {
        static void Main()
        {   //创建一个实例来进行方法
            Test t = new Test();
            t.Run();
        }
        public void Run()
        {
            LinkedList<int> myLinkedList = new LinkedList<int>();
            Random rand = new Random();
            Console.Write("Adding: ");
            for (int i = 0; i < 10; i++)
            {
                int nextInt = rand.Next(10);
                Console.Write("{0} ", nextInt);
                myLinkedList.Add(nextInt);
            }

            LinkedList<Employee> employees = new LinkedList<Employee>();
            employees.Add(new Employee("John"));
            employees.Add(new Employee("Paul"));
            employees.Add(new Employee("George"));
            employees.Add(new Employee("Ringo"));

            Console.WriteLine("\nRetrieving collections");
            Console.WriteLine("Integers: " + myLinkedList);
            Console.WriteLine("Employees: " + employees);
        }
    }
}
```

运行结果：

```
Adding：2 1 2 6 1 5 9 0 5
```

```
Retrieving collections…

Integers：0,1,1,1,2,2,5,5,6,9

Employees：George, John, Paul, Ringo
```

本例的开头声明了一个可以放到链表中的类：

```
public class Employee : IComparable<Employee>
```

这个声明指出 Employee 对象是可以进行对比的，可以看到 Employee 类实现了 CompareTo 和 Equals 方法。注意：这些方法是类型安全的（从参数传递进去的类是 Employee 类型）。LinkedList 本身声明为只操作实现了 IComparable 接口的类型：

```
public class LinkedList<T> where T : IComparable<T>
```

这样就可以保证对列表进行排序。LinkedList 操作 Node 类型的对象。Node 也实现了 IComparable 接口，并要求它本身所操作的数据也实现了 IComparable 接口：

```
public class Node<T> : IComparable<Node<T>> where T : IComparable<T>
```

这些约束使得 Node 实现 CompareTo 方法变得安全而简单，因为 Node 知道它将和其它 Node 的数据进行对比：

```
public int CompareTo(Node<T> rhs)
{
    // 这样使用是因为约束
    return data.CompareTo(rhs.data);
}
```

注意，我们不需要测试 rhs 从而得知它是否实现了 IComparable 接口；我们已经约束了 Node 只能操作实现了 IComparable 接口的数据。

4.2.3、泛型List

数组类型的一个典型问题是固定容量。如果您预先不知道数组将容纳多少对象，就会冒着给数组声明太小（溢出）或太大（浪费空间）的空间的风险。

您的程序可能让用户输入数据或从 Web 站点收集数据。当它发现对象（字符串，书，值等等），将把它们添加进数组，但您并不知道在这段时间内会收集多少数据。固定尺寸的数组并不是一个很好的选择，因为您并不知道需要多大的数组。

List 类是一个根据需要动态增加尺寸的数组。它提供了一组有用的方法和属性用于操作。它们中最重要的显示在表 9-3 中。

方法或属性	作用
Capacity	用于获取或设置List可容纳元素的数量。当数量超过容量时，这个值会自动增长。您可以设置这个值以减少容量，也可以调用trin()方法来减少容量以适合实际的元素数目。
Count	属性，用于获取数组中当前元素数量
Item( )	通过指定索引获取或设置元素。对于List类来说，它是一个索引器。
Add( )	在List中添加一个对象的公有方法
AddRange( )	公有方法，在List尾部添加实现了ICollection接口的多个元素
BinarySearch( )	重载的公有方法，用于在排序的List内使用二分查找来定位指定元素.
Clear( )	在List内移除所有元素
Contains( )	测试一个元素是否在List内
CopyTo( )	重载的公有方法，把一个List拷贝到一维数组内
Exists( )	测试一个元素是否在List内
Find( )	查找并返回List内的出现的第一个匹配元素
FindAll( )	查找并返回List内的所有匹配元素
GetEnumerator ( )	重载的公有方法，返回一个用于迭代List的枚举器
Getrange( )	拷贝指定范围的元素到新的List内
IndexOf( )	重载的公有方法，查找并返回每一个匹配元素的索引
Insert( )	在List内插入一个元素
InsertRange( )	在List内插入一组元素
LastIndexOf( )	重载的公有方法，，查找并返回最后一个匹配元素的索引
Remove( )	移除与指定元素匹配的第一个元素
RemoveAt( )	移除指定索引的元素
RemoveRange( )	移除指定范围的元素
Reverse( )	反转List内元素的顺序
Sort( )	对List内的元素进行排序
ToArray( )	把List内的元素拷贝到一个新的数组内
trimToSize( )	将容量设置为List中元素的实际数目

FCL 的习惯是给集合类提供一个 Item 元素，它在 C#中被实现为一个索引器

当您创建了一个 List，并没有定义它可以容纳多少对象。在 List 内添加元素使用 Add() 方法，列表会自己处理它内部的帐目，如例 9-13 所示。

例 9-13 List 的使用

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

```
namespace ListCollection
{
    //存储于 List 内的一个简单类
    public class Employee
    {
        private int empID;
        public Employee(int empID)
        {
            this.empID = empID;
        }
        public override string ToString()
        {
            return empID.ToString();
        }
        public int EmpID
        {
            get
            {
                return empID;
            }
            set
            {
                empID = value;
            }
        }
    }
    public class Tester
    {
        static void Main()
        {
            List<Employee> empList = new List<Employee>();
            List<int> intList = new List<int>();
            //填充 List
            for (int i = 0; i < 5; i++)
            {
                empList.Add(new Employee(i + 100));
                intList.Add(i * 5);
            }
            //打印整数列表所有内容
            for (int i = 0; i < intList.Count; i++)
            {
                Console.Write("{0} ", intList[i].ToString());
            }
```

```
                Console.WriteLine("\n");
                //打印员工列表的所有内容
                for (int i = 0; i < empList.Count; i++)
                {
                    Console.Write("{0} ", empList[i].ToString());
                }
                Console.WriteLine("\n");
                Console.WriteLine("empList.Capacity: {0}", empList.Capacity);
            }
        }
}
```

输出结果：

```
0 5 10 15 20

100 101 102 103 104

empArray.Capacity: 16
```

(译者注：很有意思，在我的电脑上 empArray.Capacity 输出的是 8，看样子老外的电脑和操作系统跟我们的是有些不同)

在 Array 中，您定义了 Array 操作对象的数目，如果尝试添加多于这个数目的元素，Array 将引发一个异常。在 List 中，您不需要声明 List 操作对象的数目。List 有一个 Capacity 属性，表示 List 能够存储的元素的数目：

```
public int Capacity { get; set; }
```

默认容量是 16，当您添加第 17 个元素时，容量会自动翻倍为 32。如果您改变 for 循环为：

```
for (int i = 0;i<17;i++)
```

输出结果会变成这样：

```
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

empArray.Capacity: 32
```

您可以手动设置容量为任何等于或大于这个数目的数字。如果您把它设置为小于这个数目的数字，程序将引发一个 ArgumentOutOfRangeException 异常。

4.2.4、实现IComparable<T>接口

实现 IComparable 接口

像所有集合类一样，List 实现了 Sort()方法，它允许您对所有实现了 IComparable 接口的对象进行排序。在下一个例子中，您将修改 Employee 类以实现 IComparable：

```
public class Employee : IComparable<Employee>
```

实现 IComparable<Employee>接口，Employee 对象必须提供 CompareTo()方法：

```
public int CompareTo(Employee rhs)
{
    return this.empID.CompareTo(rhs.empID);
}
```

CompareTo()方法把 Employee 做为参数。我们知道使用 Employee 是因为这是一个类型安全的集合。当前 Employee 对象必须把它自己跟做为参数传递进来的 Employee 进行对比，如果返回-1，表示它小于参数，如果大于参数则返回 1，如果两者相等则返回 0。这意味着决定大于、小于、等于的是 Employee。在本例中，您委托成员 empId 进行比较。empId 成员是一个 int 类型，并使用了整数类型默认的 CompareTo()方法，以对比两个值之间的大小。

System.Int32 类实现了 Icomparable<Int32>接口，所以您可以把比较的职责委托给整数类型。

您现在准备对员工数组列表（empList）进行排序，为了查看排序是否正常工作，您需要随机地添加整数和 Employee 实例到它们各自的数组中。创建随机数，需要实例化 Random 类；调用 Random 对象的 Next()方法产生随机数。Next()方法是一个重载方法；一种版本允许您传递一个整数值，表示您想要的最大随机数。在本例中，您将传递 10 来产生一个 0 到 10 之间的随机数：（译者注：参数为 10，最大的随机数只能为 9）

```
Random r = new Random();
r.Next(10);
```

例 9-14 创建了一个整型数组和一个 Employee 数组，并给两者填充随机数，并打印它们的值。随后排序数组并打印新值。

例 9-14 排序整数和 employee 数组

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

```
namespace IComparable
{
    // 一个简单的用于存储在数组中的类
    public class Employee : IComparable<Employee>
    {
        private int empID;
        public Employee( int empID )
        {
            this.empID = empID;
        }
        public override string ToString( )
        {
            return empID.ToString( );
        }
        public bool Equals( Employee other )
        {
            if ( this.empID == other.empID )
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        // Employee 使用整数默认的 CompareTo 方法
        public int CompareTo( Employee rhs )
        {
            return this.empID.CompareTo( rhs.empID );
        }
    }
    public class Tester
    {
        static void Main( )
        {
            List<Employee> empArray = new List<Employee>( );
            List<Int32> intArray = new List<Int32>( );
            // 产生整数和 employee 的 ID 的随机数
            Random r = new Random( );
            // 填充数组
            for ( int i = 0; i < 5; i++ )
            {
                // 添加随机的 employee 的 id
```

```
            empArray.Add( new Employee( r.Next( 10 ) + 100 ) );
            // 添加随机的整数
            intArray.Add( r.Next( 10 ) );
        }
        // 显示整型数组中的所有内容
        for ( int i = 0; i < intArray.Count; i++ )
        {
            Console.Write( "{0} ", intArray[i].ToString( ) );
        }
        Console.WriteLine( "\n" );
        // 显示 Employee 数组中的所有内容
        for ( int i = 0; i < empArray.Count; i++ )
        {
            Console.Write( "{0} ", empArray[i].ToString( ) );
        }
        Console.WriteLine( "\n" );
        // 整型数组排序
        intArray.Sort( );
        for ( int i = 0; i < intArray.Count; i++ )
        {
            Console.Write( "{0} ", intArray[i].ToString( ) );
        }
        Console.WriteLine( "\n" );
        // employee 数组排序并显示
        //原文的下面这两句应该注释掉，现在还没用到
        //Employee.EmployeeComparer c = Employee.GetComparer( );
        //empArray.Sort(c);
        empArray.Sort( );
        // 显示 Employee 数组中的所有内容
        for ( int i = 0; i < empArray.Count; i++ )
        {
            Console.Write( "{0} ", empArray[i].ToString( ) );
        }
        Console.WriteLine( "\n" );
    }
  }
}
```

输出结果：

```
4 5 6 5 7
108 100 101 103 103
4 5 5 6 7
100 101 103 103 108
```

输出显示整型数组和 Employee 数组产生的是随机数。排序后，显示的是已经进行排序后的值。

实现 IComparer 接口

当您在 List 中调用 Sort()，默认的 IComparer（译者注：这里可能是错误，应该为 IComparable）实现被调用，它调用 IComparable 所实现的 CompareTo()方法对 List 内的每个元素进行快速排序。

当您想控制排列方式时，可以自由地创建自己的 IComparer 实现。在下一个例子中，将在 Employee 中添加第二个字段：yearsOfSvc。您希望在 List 中按两种字段 empID 或 yearsOfSvc 来对 Employee 对象进行排序。

为了达到这个目的，需要创建 IComparer 的实现，用于传递给 List 中 Sort()方法的参数。这个 IComparer 类是 EmployeeComparer，它让 Employee 对象知道如何进行排序。

EmployeeComparer 类有一个 WhichComparison 属性，它是 Employee.EmployeeComparer.ComparisonType 类型：

```
public Employee.EmployeeComparer.ComparisonType WhichComparison
{
    get{return whichComparison;}
    set{whichComparison = value;}
}
```

ComparisonType 是一个枚举类型，它有两个值：empID 或 yearsOfSvc（指示您希望按员工 ID 还是工龄进行排序）：

```
public enum ComparisonType
{
    EmpID,
    YearsOfService
};
```

在调用 Sort()方法之前，创建 EmployeeComparer 的实例并设置它的 ComparisonType 属性：

```
Employee.EmployeeComparer c = Employee.GetComparer();
c.WhichComparison=Employee.EmployeeComparer.ComparisonType.EmpID;
empArray.Sort(c);
```

调用 Sort()方法之时，List 会调用 EmployeeComparer 中的 Compare 方法，并把当前要

对比的字段通过 WhickComparison 属性传递给 Employee.CompareTo()方法：

```
public int Compare( Employee lhs, Employee rhs )
{
    return lhs.CompareTo( rhs, WhichComparison );
}
```

Employee 对象必须实现一个自定义的 CompareTo()方法，用于获得比较方式并按照要求进行对比：

```
public int CompareTo(Employee rhs, Employee.EmployeeComparer.ComparisonType which)
{
    switch (which)
    {

        case Employee.EmployeeComparer.ComparisonType.EmpID:

            return this.empID.CompareTo(rhs.empID);

        case Employee.EmployeeComparer.ComparisonType.Yrs:

            return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);

    }
    return 0;
}
```

例 9-15 是这个例子的完整代码。为了简化例子，整型数组已经被移除，用于输出的 Employee 的 ToString()方法也增加了代码以让您看得到排序的效果。

例 9-15 按员工的 ID 和工龄进行排序

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

namespace IComparer
{
    public class Employee : IComparable<Employee>
    {
        private int empID;
        private int yearsOfSvc = 1;
        public Employee(int empID)
```

```
{
    this.empID = empID;
}
public Employee(int empID, int yearsOfSvc)
{
    this.empID = empID;
    this.yearsOfSvc = yearsOfSvc;
}
public override string ToString()
{
    return "ID：" + empID.ToString() +
        ". Years of Svc：" + yearsOfSvc.ToString();
}
public bool Equals(Employee other)
{
    if (this.empID == other.empID)
    {
        return true;
    }
    else
    {
        return false;
    }
}
//静态方法，用于获取一个 Comparer 对象
public static EmployeeComparer GetComparer()
{
    return new Employee.EmployeeComparer();
}
public int CompareTo(Employee rhs)
{
    return this.empID.CompareTo(rhs.empID);
}
//通过自定义 comparer 来调用指定的实现
public int CompareTo(Employee rhs,
    Employee.EmployeeComparer.ComparisonType which)
{
    switch (which)
    {
        case Employee.EmployeeComparer.ComparisonType.EmpID:
            return this.empID.CompareTo(rhs.empID);
        case Employee.EmployeeComparer.ComparisonType.Yrs:
            return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
    }
```

```
            return 0;
        }
        //实现 IComparer 接口的嵌套类
        public class EmployeeComparer : IComparer<Employee>
        {
            private Employee.EmployeeComparer.ComparisonType
                whichComparison;
            //比较方式枚举
            public enum ComparisonType
            {
                EmpID,
                Yrs
            };
            public bool Equals(Employee lhs, Employee rhs)
            {
                return this.Compare(lhs, rhs) == 0;
            }
            public int GetHashCode(Employee e)
            {
                return e.GetHashCode();
            }
            public int Compare(Employee lhs, Employee rhs)
            {
                return lhs.CompareTo(rhs, WhichComparison);
            }
            public Employee.EmployeeComparer.ComparisonType
                WhichComparison
            {
                get { return whichComparison; }
                set { whichComparison = value; }
            }
        }
    }
    public class Tester
    {
        static void Main()
        {
            List<Employee> empArray = new List<Employee>();
            Random r = new Random();
            for (int i = 0; i < 5; i++)
            {
                //添加一个随机的员工 ID
                empArray.Add(new Employee(
                    r.Next(10) + 100, r.Next(20)));
```

```
            }
            //显示 Employee 数组的所有内容
            for (int i = 0; i < empArray.Count; i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
            //排序并显示 Employee 数组
            Employee.EmployeeComparer c = Employee.GetComparer();
            c.WhichComparison =
                Employee.EmployeeComparer.ComparisonType.EmpID;
            empArray.Sort(c);
            //显示 Employee 数组的所有内容
            for (int i = 0; i < empArray.Count; i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");

            c.WhichComparison = Employee.EmployeeComparer.ComparisonType.Yrs;
            empArray.Sort(c);
            for (int i = 0; i < empArray.Count; i++)
            {
                Console.Write("\n{0} ", empArray[i].ToString());
            }
            Console.WriteLine("\n");
        }
    }
}
```

输出结果：

```
ID: 103. Years of Svc: 11
ID: 108. Years of Svc: 15
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 5
ID: 102. Years of Svc: 0
ID: 102. Years of Svc: 0
ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
ID: 108. Years of Svc: 5
ID: 102. Years of Svc: 0
```

```
ID: 108. Years of Svc: 5

ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
```

第一块输出显示的是 Employee 对象被加进 List 时的情形。员工 ID 值和工龄是随机顺序的。第二块显示的是按员工 ID 排序后的结果，第三块显示的是按工龄排序后的结果。

如果您如例 9-11 那样创建自己的集合，并希望实现 IComparer，可能需要确保所有放在列表中的类型都实现了 IComparer 接口（这样他们才有可能被排序），这可以通过前面讲述的约束来实现。

5、构建可反转排序的泛型字典类

5.1、版权声明

文章出处：http://www.cnblogs.com/abatei/archive/2008/02/20/1075760.html
文章作者：abatei

5.2、内容详情

5.2.1、雏形

你想构建一个集合类用于存储数据，它里面的值是成对出现的，每一对值都包含“键”和“值”两个部分。键和值里存放的数据类型是不确定的，最好什么类型放到里面都适用。想起了什么？就是它！泛型！真是太伟大了！

接下来要考虑的问题是采用什么样的方式来存储这些值，你觉得自己的数据结构学得还比较好，决定要控制一切。在所有集合类型中，数组的速度是最快的，而且它使用方便并且是类型安全的，唯一的缺点就是容量固定。好！不管那么多了，用的就是它。先把代码写出来再说。

```
public class ReversibleSortedList<TKey, TValue>
{
    private TKey[] keys; //键数组
    private TValue[] values; //值数组
}
```

总算迈出了第一步，确定了大的方向。但是数组是容量固定的，如何能让它的容量可以随着元素的增长而自动增长呢？即然要控制容量，那就要有一个容量属性，用于读取和设置容量。先从读取开始，容量值就是数组 keys 或 values 的长度。好，继续添加代码：

```
public class ReversibleSortedList<TKey, TValue>
{
    private TKey[] keys; //键数组
    private TValue[] values; //值数组
    public int Capacity //容量属性
    {
        get
        {
            return this.keys.Length;
```

```
        }
    }
}
```

现在问题来了，当 ReversibleSortedList 被实例化后直接读取 Capacity 属性，将会调用 keys.Length 属性来返回。但此时 keys 并没有被初始化，肯定不能访问其 Length 属性。可不可以在声明的同时把它初始化为元素个数为 0 的数组呢？

```
private TKey[] keys = new TKey[0]; //键数组
```

这样做是没有什么问题，但它刚声明就成为了垃圾，什么都放不了，总让人感觉不舒服。那可不可以让数组初始化为一个固定容量呢？

```
private TKey[] keys = new TKey[4]; //键数组
```

但你是一个完美主义者，不希望这样就使用了 4 个位置的空间，你更希望在添加元素时才会有空间的拓展。可不可以声明两个静态的，初始长度为 0 的数组做为其初始状态呢？这样不管类有多少个实例，进行初始化时使用的都是它们。这样即避免了空间的浪费，也不再需要多次初始化 0 长度数组。

```
    private static TKey[] emptyKeys; //用于键数组的初始化
    private static TValue[] emptyValues; //用于值数组的初始化
```

可以在静态构造器（又称类型构造器）里把它们初始化为长度为 0 的数组，并在无参实例构造器中把这两个初始值赋给 keys 和 values 数组。

为了测试自己的想法，需要添加 Main()方法进行测试，下列代码可以直接拷贝并运行，如果不知道如何运行，请参考：

http://www.enet.com.cn/eschool/video/c/1.shtml

ReversibleSortedList 0.1 版本：对 ReversibleSortedList 类的容量进行初始化

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

public class ReversibleSortedList<TKey, TValue>
{
    #region  成员变量
    private TKey[] keys; //键数组
    private TValue[] values; //值数组
    private static TKey[] emptyKeys; //用于键数组的初始化
```

```
    private static TValue[] emptyValues; //用于值数组的初始化
    #endregion
    #region 构造方法
    //类型构造器
    static ReversibleSortedList()
    {
        ReversibleSortedList<TKey, TValue>.emptyKeys = new TKey[0];
        ReversibleSortedList<TKey, TValue>.emptyValues = new TValue[0];
    }
    public ReversibleSortedList()
    {
        this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
        this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
    }
    #endregion
    #region 公有属性
    public int Capacity //容量属性
    {
        get
        {
            return this.keys.Length;
        }
    }
    #endregion
}
public class Test
{
    static void Main()
    {
        ReversibleSortedList<int, string> rs=new ReversibleSortedList<int, string>();
        Console.WriteLine(rs.Capacity);
    }
}
```

运行结果：

```
0
```

5.2.2、排序方向

你希望 ReversibleSortedList 类中的元素是以 TKey（键）的顺序进行存储的，并且它即

可以从小排到大，也可以从大排到小。当然，最佳方式就是在添加元素时找到合适的位置插入，插入后元素就已经按顺序排好。在一个有序数组中查找合适的插入点这样的算法并不困难，但 FCL 已经帮我们实现了，而且是采用速度最快的二分查找法（在 MSDN 中被称为“二进制搜索法”）。太棒了！它就是：静态方法 Array.BinarySearch。下面我们来看 MSDN 中对它的介绍。

Array.BinarySearch 一共有 8 个重载版本，最后一个是我们需要的：

```
public static int BinarySearch<T> (
    T[] array, //要搜索的从零开始的一维排序 Array
    int index, //要搜索的范围的起始索引。
    int length, //要搜索的范围的长度。
    T value, //要搜索的对象。
    IComparer<T> comparer //比较元素时要使用的 IComparer 实现。
)
```

其中，T 表示数组元素的类型。对返回值的介绍是：如果找到 value，则为指定 array 中的指定 value 的索引。如果找不到 value 且 value 小于 array 中的一个或多个元素，则为一个负数，该负数是大于 value 的第一个元素的索引的按位求补。如果找不到 value 且 value 大于 array 中的任何元素，则为一个负数，该负数是最后一个元素的索引加 1 的按位求补。

我们的 ReversibleSortedList 不能插入重复的键值。当返回值大于或等于 0 时，表明不能插入。当返回值小于零时，表明没有找到重复键，而且这时返回值还带有插入位置的信息。考虑得可真周到啊，赞一个！

求补是什么呢？就是把二进制数的 0 变成 1，1 变成 0。对于 int 来说，由于它是有符号整数，求补会把正数变为负数，把负数变为正数。对一个数进行两次求补运算就会得到原来的数。哈哈，如果反回值小于 0，对它求补就可以得到插入位置信息了。真是得来全不费工夫！

现在的问题是需要一个实现了 IComparer<T>接口的类。可以在 ReversibleSortedList 声明一个嵌套类以解决这个问题。当然，在实现 IComparer<T>接口的 Compare 方法时在里面做些手脚就可以实现正向和反向排序了。这时需要一个能表示正向和反向排序的东西，FCL 里有现成的，它就是 System.ComponentModel 命名空间下的 ListSortDirection 枚举。它有两个值：Ascending 表示升序，Descending 表示降序。下面的代码在 1.0 版本的基础上添加了实现 IComparer<T>接口的内部类：SortDirectionComparer<T>。代码可直接拷贝运行，运行它纯粹是为了检查是否有错误，没有什么看得见的效果。

ReversibleSortedList 0.2 版本：添加了实现 IComparer<T>接口的内部类

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

```
using System.ComponentModel;

public class ReversibleSortedList<TKey, TValue>
{
    #region 成员变量
    private TKey[] keys=new TKey[0]; //键数组
    private TValue[] values; //值数组
    private static TKey[] emptyKeys;
    private static TValue[] emptyValues;
    #endregion
    #region 构造方法
    //类型构造器
    static ReversibleSortedList()
    {
        ReversibleSortedList<TKey, TValue>.emptyKeys = new TKey[0];
        ReversibleSortedList<TKey, TValue>.emptyValues = new TValue[0];
    }
    public ReversibleSortedList()
    {
        this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
        this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
    }
    #endregion
    #region 公有属性
    public int Capacity //容量属性
    {
        get
        {
            return this.keys.Length;
        }
    }
    #endregion
    #region SortDirectionComparer 类定义
    public class SortDirectionComparer<T> : IComparer<T>
    {   //ListSortDirection 枚举，有两个值：
        //Ascending 按升序排列，Descending 按降序排列
        private System.ComponentModel.ListSortDirection _sortDir;
        //构造方法
        public SortDirectionComparer()
        {   //默认为升序
            _sortDir = ListSortDirection.Ascending;
        }
        //可指定排序方向的构造方法
        public SortDirectionComparer(ListSortDirection sortDir)
```

```
            {
                _sortDir = sortDir;
            }
            //排序方向属性
            public System.ComponentModel.ListSortDirection SortDirection
            {
                get { return _sortDir; }
                set { _sortDir = value; }
            }
            //实现 IComparer<T>接口的方法
            public int Compare(T lhs, T rhs)
            {
                int compareResult =
                    lhs.ToString().CompareTo(rhs.ToString());
                // 如果是降序，则反转.
                if (SortDirection == ListSortDirection.Descending)
                    compareResult *= -1;
                return compareResult;
            }
        }
        #endregion
}
public class Test
{
        static void Main()
        {
            ReversibleSortedList<int, string> rs=new ReversibleSortedList<int, string>();
            Console.WriteLine(rs.Capacity);
        }
}
```

5.2.3、实现元素添加及自动扩展

您是一单位 CEO，单位占地 50 亩，这几年在你的带领下，公司不断发展壮大，原来 50 亩地已经不够用。公司急需扩大地盘，这个现实问题摆在你面前，该怎么办？到旁边单位抢地？不行，现在是法制社会。有两个解决方案，第一是买一块 50 亩的地，这样你的公司就有两个办公地点，缺点是不能统一管理，两个地点的员工交流不顺畅。第二是买一块 100 亩的地，把原来的地卖掉，公司全部搬到新地点。这样做的缺点是重建费用太大。

我们要构建的 ReversibleSortedList 集合也面临着这样的问题，由于使用数组存放数据，数组的空间一旦确定就不能再改变。这一次我们选择了第二种方案，原因很简单，内存间成片数据的拷贝速度非常快，不耗费什么成本。既然搬家费用不高，有什么理由把公司一分为二呢？

ReversibleSortedList 中的方案是，初始空间为 0，当有元素添加时，空间增长为 4，每当添加新元素时，如果现有空间已满，则另开辟一块大小为原来 2 倍的空间，把原来的数据拷贝到新空间后再添加新元素。当然，原来存放数据的空间这时就变成了待回收的垃圾。

由于数组的长度只能代表 ReversibleSortedList 的存储空间，并不能表示当前元素个数，所以需要使用一个成员变量来表示当前元素个数：

```
private int _size; //表示元素个数
public int Count //属性，表示当前元素个数
{
    get
    {
        return this._size;
    }
}
```

前面声明的 SortDirectionComparer<T>内部类也需要进行初始化：

```
private SortDirectionComparer<TKey> _sortDirectionComparer = null;
```

注意，这里把 TKey 做为类型参数传递给 SortDirectionComparer<T>，TKey 本身也是一个类型参数。

在无参实例构造方法中对它们进行初始化：

```
this._size = 0;
this._sortDirectionComparer = new SortDirectionComparer<TKey>();
```

剩下的就是插入数据的方法，共有 1 个公方法：Add，3 个私有方法：Insert、EnsureCapacity、InternalSetCapacity。具体的方法代码请参考稍后的 ReversibleSortedList 0.3 版本。关于以上几个方法的算法及作用，请参考代码中的注释。这里讲一下添加数据的过程，如图 1 所示，首先利用 Array.BinarySearch 查找到插入点，然后把插入点及其后元素向后移动一个位置，最后在插入点插入数据。这里有一点需要明确，任何时候，数据在内存中都是以有序的方法排列的。

为了对程序进行测试，添加了一个 Print 方法用于打印数组中的元素。代码测试成功后可以把它删除。并且在 Main()方法中依次添加 9 个元素并在期间打印数组元素进行观察。

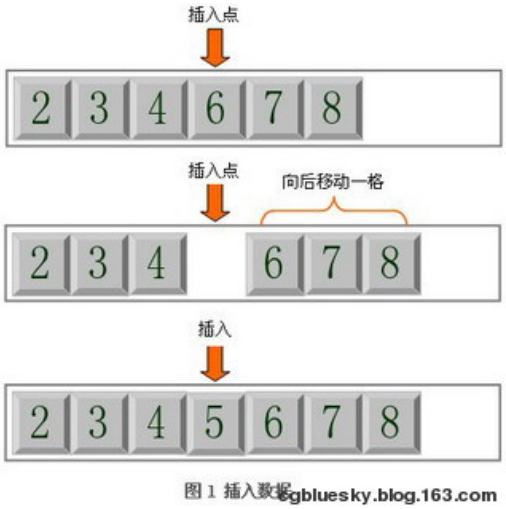

图 1 插入数据 gbluesky.blog.163.com

ReversibleSortedList 0.3 版本：添加数据（以下代码可直接拷贝并运行）

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

public class ReversibleSortedList<TKey, TValue>
{
    #region  成员变量
    private TKey[] keys=new TKey[0]; //键数组
    private TValue[] values; //值数组
    private static TKey[] emptyKeys;
    private static TValue[] emptyValues;
    private SortDirectionComparer<TKey> _sortDirectionComparer = null;
    private int _size; //表示元素个数
    #endregion
    #region  构造方法
    //类型构造器
    static ReversibleSortedList()
    {    //设置数组初始状态值
        ReversibleSortedList<TKey, TValue>.emptyKeys = new TKey[0];
        ReversibleSortedList<TKey, TValue>.emptyValues = new TValue[0];
    }
    public ReversibleSortedList()
    {
        this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
        this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
        this._size = 0;
        this._sortDirectionComparer = new SortDirectionComparer<TKey>();
    }
```

```
#endregion
#region 公有属性
public int Capacity //容量属性
{
    get
    {
        return this.keys.Length;
    }
}
public int Count //当前元素个数
{
    get
    {
        return this._size;
    }
}
#endregion
#region 公有方法
//添加元素
public void Add(TKey key, TValue value)
{
    if (key.Equals(null))
    {
        throw new ArgumentNullException("key");
    }
    //使用二分查找法搜索将插入的键
    int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,
                                                    this._sortDirectionComparer);
    if (num1 >= 0) //如果数组中已存在将插入的键
    {
        throw new ArgumentException("尝试添加重复值！ ");
    }
    this.Insert(~num1, key, value); //在插入点插入键和值
}
public void Print() //只用于测试
{
    for(int i=0;i<_size;i++)
    {
        Console.WriteLine("key:{0}    value:{1}",keys[i],values[i]);
    }
}
#endregion
#region 私有方法
private void Insert(int index, TKey key, TValue value)
```

```
{    //在指定索引入插入数据
     if (this._size == this.keys.Length)
     {
          this.EnsureCapacity(this._size + 1);
     }
     if (index < this._size)
     {    //当插入元素不是添加在未尾时，移动插入点后面的元素
          Array.Copy(this.keys, index, this.keys, (int)(index + 1),
                    (int)(this._size - index));
          Array.Copy(this.values, index, this.values, (int)(index + 1),
                    (int)(this._size - index));
     }
     this.keys[index] = key; //在插入点插入键
     this.values[index] = value; //在插入点插入值
     this._size++;
}
private void EnsureCapacity(int min) //确保当前容量
{    //如果当前容量为，则增长为，否则翻倍
     int num1 = (this.keys.Length == 0) ? 4 : (this.keys.Length * 2);
     if (num1 < min)
     {
          num1 = min;
     }
     this.InternalSetCapacity(num1);
}
private void InternalSetCapacity(int value)
{    //调整容量
     if (value != this.keys.Length)
     {
          if (value < this._size)
          {
               throw new ArgumentOutOfRangeException(
                    "value", "要调整的容量值太小");
          }
          if (value > 0)
          {    //重新开辟一块内存空间用来存放集合中的值
               TKey[] localArray1 = new TKey[value];
               TValue[] localArray2 = new TValue[value];
               if (this._size > 0)
               {    //数据搬家
                    Array.Copy(this.keys, 0, localArray1, 0, this._size);
                    Array.Copy(this.values, 0, localArray2, 0, this._size);
               }
               this.keys = localArray1;
```

```
                    this.values = localArray2;
                }
                else
                {   //设置容量为
                    this.keys = ReversibleSortedList<TKey, TValue>.emptyKeys;
                    this.values = ReversibleSortedList<TKey, TValue>.emptyValues;
                }
            }
        }
        #endregion
        #region SortDirectionComparer 类定义
        public class SortDirectionComparer<T> : IComparer<T>
        {   //ListSortDirection 枚举，有两个值：
            //Ascending 按升序排列，Descending 按降序排列
            private System.ComponentModel.ListSortDirection _sortDir;
            //构造方法
            public SortDirectionComparer()
            {   //默认为升序
                _sortDir = ListSortDirection.Ascending;
            }
            //指定排序方向的构造方法
            public SortDirectionComparer(ListSortDirection sortDir)
            {
                _sortDir = sortDir;
            }
            //排序方向属性
            public System.ComponentModel.ListSortDirection SortDirection
            {
                get { return _sortDir; }
                set { _sortDir = value; }
            }
            //实现 IComparer<T>接口的方法
            public int Compare(T lhs, T rhs)
            {
                int compareResult =
                    lhs.ToString().CompareTo(rhs.ToString());
                // 如果是降序，则反转.
                if (SortDirection == ListSortDirection.Descending)
                    compareResult *= -1;
                return compareResult;
            }
        }
        #endregion // SortDirectionComparer
}
```

```
public class Test
{
    static void Main()
    {
        ReversibleSortedList<int, string> rs=new ReversibleSortedList<int, string>();
        rs.Add(3,"a");
        rs.Add(1,"b");
        rs.Add(2,"c");
        rs.Add(6,"d");
        rs.Print();
        Console.WriteLine("当前容量为：  "+rs.Capacity+"元素个数为：  "+rs.Count);
        rs.Add(5,"e");
        rs.Add(4,"f");
        rs.Print();
        Console.WriteLine("当前容量为：  "+rs.Capacity+"元素个数为：  "+rs.Count);
        rs.Add(8,"g");
        rs.Add(7,"h");
        rs.Add(9,"i");
        rs.Print();
        Console.WriteLine("当前容量为：  "+rs.Capacity+"元素个数为：  "+rs.Count);
    }
}
```

运行结果：

```
key：1   value：b
key：2   value：c
key：3   value：a
key：4   value：d

当前容量为：4  元素个数为：4

key：1   value：b
key：2   value：c
key：3   value：a
key：4   value：f
key：5   value：e
key：6   value：d

当前容量为：8  元素个数为：6

key：1   value：b
key：2   value：c
key：3   value：a
```

```
key：4   value：f
key：5   value：e
key：6   value：d
key：7   value：h
key：8   value：g
key：9   value：i

当前容量为：16  元素个数为：9
```

从运行结果可以得知：刚开始插入了 4 个元素后，容量为 4，接下来再插 2 个元素，容量自动扩展为 8。最后再插入 3 个元素，容量自动扩展为 16。并且，元素是按 key 的顺序进行排列的，完全符合我们之前的预想。终于可以点鞭炮庆祝一下了，但不要高兴得太早，百里长征大概才走了几里地。随着代码越来越复杂，要走的路会变得更艰难。

5.2.4、IDictionary接口

C#对集合类型有统一的规范。它的好处不言而喻，所有集合类都有一些统一的调用方法和属性，这使得学习成本大大降低。统一的规范就是通过接口来实现的，另一方面一些类也会直接调用这些标准接口，使得我们写出来的类有更好的兼容性。最典型的例子莫过于 IEnumerable 接口，只要实现了它就可以使用 foreach 语句进行调用。

我们将要给 ReversibleSortedList 实现的是 IDictionary 接口，先来看看它的定义：

```
public interface IDictionary : ICollection, IEnumerable
```

ICollection 接口是所有集合类的基接口，FCL 中所有集合，不管是哪种方式的集合都实现它。

IEnumerable 接口则是枚举器接口，实现了它就可以使用 foreach 语句对它进行访问。

IDictionary 接口则继承自这两个接口，它表示键/值对的非通用集合。

ICollection 接口的定义为：

```
public interface ICollection : IEnumerable
```

从这个定义可以看出，所有集合类都应该支持 foreach 语句进行访问。

表 1 列出了各个接口的成员

接口	成员	说明
ICollection	Count属性	获取 ICollection 中包含的元素数
	IsSynchronized属性	获取一个值，该值指示是否同步对 ICollection 的访问（线程安全）
	SyncRoot属性	获取可用于同步 ICollection 访问的对象
	CopyTo方法	从特定的 Array 索引处开始，将 ICollection 的元素复制到一个 Array 中
IEnumerable	GetEnumerator方法	返回一个循环访问集合的枚举器
IDictionary	IsFixedSize属性	获取一个值，该值指示 IDictionary 对象是否具有固定大小
	IsReadOnly属性	获取一个值，该值指示 IDictionary 对象是否为只读
	Item属性	获取或设置具有指定键的元素
	Keys属性	获取 ICollection 对象，它包含 IDictionary 对象的键
	Values属性	获取 ICollection 对象，它包含 IDictionary 对象中的值
	Add方法	在 IDictionary 对象中添加一个带有所提供的键和值的元素
	Clear方法	从 IDictionary 对象中移除所有元素
	Contains方法	确定 IDictionary 对象是否包含具有指定键的元素
	GetEnumerator方法	返回一个用于 IDictionary 对象的 IDictionaryEnumerator 对象
	Remove方法	从 IDictionary 对象中移除带有指定键的元素

从上表可以看出，实现 IDictionary 接口并不简单。我个人喜欢先把复杂的问题简单化，但所要实现的东西实在太多，只能尽力而为。如果在我们之前所构建的 ReversibleSortedList 类中加上这么代码将不利于理解如何实现 IDictionary 接口。所以我从 MSDN 上 copy 了一段 IDictionary 接口实现的代码，并把所有属于出错判断的部分咔嚓掉，先让大家对 IDictionary 接口有个了解后再给 ReversibleSortedList 类实现它。

实现这个接口需要注意以下几点：

1）IEnumerable 接口的 GetEnumerator 方法的实现，这个方法的原型为：

```
IEnumerator GetEnumerator()
```

也就是说这个方法返回的是一个实现了 IEnumerator 的类，IEnumerator 接口的成员如下：

Current 属性：获取集合中的当前元素

MoveNext 方法：将枚举数推进到集合的下一个元素

Reset 方法：将枚举数设置为其初始位置，该位置位于集合中第一个元素之前

实现了 IEnumerator 接口的类可以通过一个内部类来实现，又要多实现三个成员，事态变得进一步复杂。关于 IEnumerable 接口实现，如果不懂可以上网搜搜，能搜一大箩筐出来，也可以参考设计模式中的“Iterator 迭代器模式”进行学习。

IDictionary 接口中有自己的 GetEnumerator 方法，它返回的 IEnumerator 接口的子接口：IDictionaryEnumerator 接口。这个接口的定义为：

public interface IDictionaryEnumerator : IEnumerator

它的成员为：

Entry：同时获取当前字典项的键和值

Key：获取当前字典项的键

Value：获取当前字典项的值

这意味着除了要实现 IEnumerable 接口的三个成员外，还要实现它自己的三个成员，变得更复杂了，晕！值得庆幸的是由于 IDictionaryEnumerator 接口继承自 IEnumerator 接口，可以把 IDictionaryEnumerator 接口强制转换为 IEnumerator 接口用于实现 IEnumerable 接口的 GetEnumerator 方法。当然，由于两个方法同名，只能给 IEnumerable 接口用显示接口成员实现了。

2） Item 属性，这个属性并不是叫你实现了个名字叫“Item”的属性，而是一个索引器，通过实例名加方括号中的索引来访问集合里的元素。关于索引器，如果不熟，请参考：http://www.enet.com.cn/eschool/video/c/20.shtml 。

3）Keys 属性，这个属性的原型为：

```
ICollection Keys { get; }
```

也就是说，它返回一个实现了 ICollection 接口的类。ICollection 接口前面已经讲过，C#中的所有集合类都实现了它。本例中这个属性返回的是一个数组，因为数组也实现了 ICollection 接口。另外 Values 属性也是同样的情况。

4）本例中，字典集合里的一个元素是由键和值组成的，这一个元素使用了 FCL 中现成的 DictionaryEntry 来实现，它定义了可设置或检索的字典键/值对。

下面列出了代码，请大家参照注释进行理解。

IDictionary 接口的实现（以下代码可直接拷贝并运行）

```
using System;
using System.Collections;

public class SimpleDictionary : IDictionary
{
    private DictionaryEntry[] items; //用数组存放元素
    private Int32 ItemsInUse = 0; //元素个数
    //指定存储空间的构造方法
```

```
public SimpleDictionary(Int32 numItems)
{
    items = new DictionaryEntry[numItems];
}
#region IDictionary 成员
public bool IsReadOnly { get { return false; } }
public bool Contains(object key)
{   //检测是否包含指定的 key 键
    Int32 index;
    return TryGetIndexOfKey(key, out index);
}
public bool IsFixedSize { get { return false; } }
public void Remove(object key)
{   //移除指定键的元素
    Int32 index;
    if (TryGetIndexOfKey(key, out index))
    {   //把移除元素后面的所有元素向前移动一个位置
        Array.Copy(items, index + 1, items, index, ItemsInUse - index - 1);
        ItemsInUse--;
    }
}
public void Clear() { ItemsInUse = 0; } //清除所有元素
public void Add(object key, object value)
{   //添加一个元素
    items[ItemsInUse++] = new DictionaryEntry(key, value);
}
public ICollection Keys
{   //返回所有键的集合
    get
    {   //把所有键的集合拷贝到新数组中并返回
        Object[] keys = new Object[ItemsInUse];
        for (Int32 n = 0; n < ItemsInUse; n++)
            keys[n] = items[n].Key;
        return keys;
    }
}
public ICollection Values
{   //返回所有值的集合
    get
    {   //把所有值的集合拷贝到新数组中并返回
        Object[] values = new Object[ItemsInUse];
        for (Int32 n = 0; n < ItemsInUse; n++)
            values[n] = items[n].Value;
        return values;
```

```
        }
    }
    public object this[object key]
    {    //item 属性的实现，也就是索引器
        get
        {
            Int32 index;
            if (TryGetIndexOfKey(key, out index))
            {
                return items[index].Value;
            }
            return null;
        }
        set
        {
            Int32 index;
            if (TryGetIndexOfKey(key, out index))
            {
                items[index].Value = value;
            }
            else
            {
                Add(key, value);
            }
        }
    }
    //这个方法有两个作用，第一是查找指定键是否存在
    //第二是返回指定键的索引
    private Boolean TryGetIndexOfKey(Object key, out Int32 index)
    {
        for (index = 0; index < ItemsInUse; index++)
        {
            if (items[index].Key.Equals(key)) return true;
        }
        return false;
    }
    //用于迭代的嵌套类，它同时实现了 IEnumerator 和 IDictionaryEnumerator
    private class SimpleDictionaryEnumerator : IDictionaryEnumerator
    {
        DictionaryEntry[] items;
        Int32 index = -1;
        //构造方法，用于获得 SimpleDictionary 类的所有元素
        public SimpleDictionaryEnumerator(SimpleDictionary sd)
        {
```

```
            items = new DictionaryEntry[sd.Count];
            Array.Copy(sd.items, 0, items, 0, sd.Count);
        }
        public Object Current
        {
            get { return items[index]; }
        }
        public DictionaryEntry Entry
        {
            get { return (DictionaryEntry) Current; }
        }
        public Object Key { get { return items[index].Key; } }
        public Object Value { get { return items[index].Value; } }
        public Boolean MoveNext()
        {
            if (index < items.Length - 1) { index++; return true; }
            return false;
        }
        public void Reset()
        {
            index = -1;
        }
    }
    //实现 IDictionary 接口中的 GetEnumerator 方法
    public IDictionaryEnumerator GetEnumerator()
    {
        return new SimpleDictionaryEnumerator(this);
    }
    #endregion

    #region ICollection 成员
    public bool IsSynchronized { get { return false; } }
    //这句够简化，直接弹出异常不给使用
    public object SyncRoot { get { throw new NotImplementedException(); } }
    public int Count { get { return ItemsInUse; } }
    public void CopyTo(Array array, int index) { throw new NotImplementedException(); }
    #endregion

    #region IEnumerable 成员
    //实现 IEnumerable 接口的 GetEnumerator 方法
    IEnumerator IEnumerable.GetEnumerator()
    {
        // 这里使用了强制类型转换.
        return ((IDictionary)this).GetEnumerator();
```

```
    }
    #endregion
}
public sealed class App
{
    static void Main()
    {
        // 创建一个只能包含三个元素的字典类
        IDictionary d = new SimpleDictionary(3);
        // 添加三个人名和它们的年龄到字典内
        d.Add("Jeff", 40);
        d.Add("Kristin", 34);
        d.Add("Aidan", 1);
        Console.WriteLine("字典元素个数= {0}", d.Count);
        Console.WriteLine("字典中是否包含'Jeff'? {0}", d.Contains("Jeff"));
        Console.WriteLine("Jeff 的年龄是：{0}", d["Jeff"]);
        // 显示字典中的所有键和值，由于实现了 IDictionaryEnumerator 接口
        //所以可以使用 foreach 进行调用
        foreach (DictionaryEntry de in d)
        {
            Console.WriteLine("{0} is {1} years old.", de.Key, de.Value);
        }
        // 移除“Jeff”
        d.Remove("Jeff");
        // 移除不存在的元素不会引发异常
        d.Remove("Max");
        // 显示字典中的所有人名（key）
        foreach (String s in d.Keys)
            Console.WriteLine(s);
        // 显示字典中的所有年龄（value)
        foreach (Int32 age in d.Values)
            Console.WriteLine(age);
    }
}
```

5.2.5、实现IEnumerable<T>接口

实现 IEnumerable<KeyValuePair<TKey, TValue>>接口
我们先来看看 ReversibleSortedList 类的定义：

```
public class ReversibleSortedList<TKey, TValue> :
IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>,
IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable
```

它一共实现了 6 个接口。但从本质上来说，实现 IDictionary<TKey, TValue>接口和 IDictionary 接口就等同于实现了以上 6 个接口。因为 IDictionary<TKey, TValue>继承自 ICollection<KeyValuePair<TKey, TValue>>和 IEnumerable<KeyValuePair<TKey, TValue>>接口，而 IDictionary 接口继承自 ICollection 和 IEnumerable 接口。下面我们来看一下这些接口的关系图：

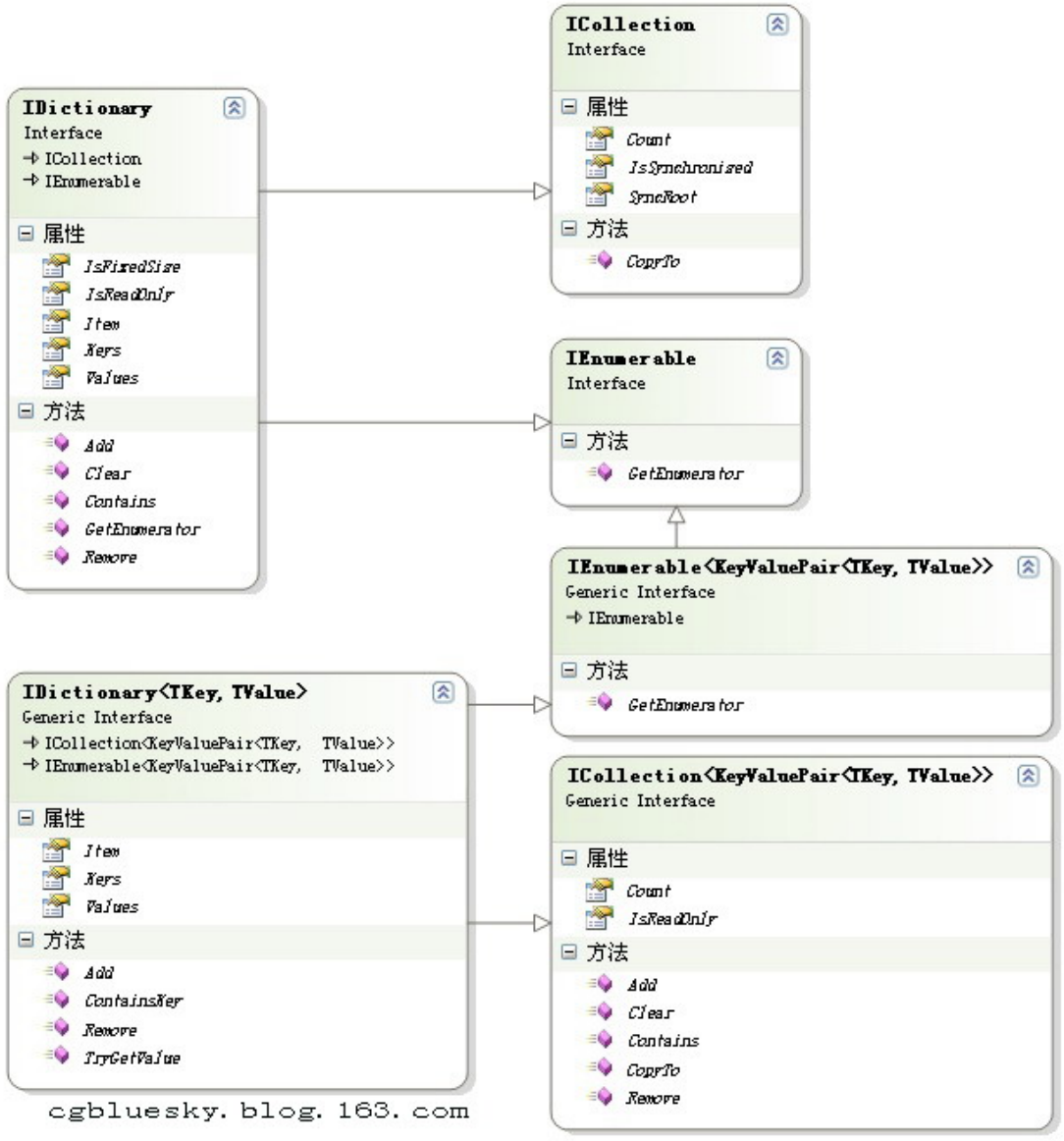

图 2　ReversibleSortedList 类接口关系图

其实 ReversibleSortedList 类的大部份代码都用于实现左边这两个接口，看到这张图你应该理解了为什么会需要 1300 行的代码。为了一步一步地实现这个类，我们需要首先实现底层接口，由于 ICollection 接口本身就继承自 IEnumerable 接口，所以首先应该实现的是 IEnumerable 接口。从前面一节可以知道 IEnumerable 接口的实现需要一个嵌套类，它返回一个枚举器。我们可以为 IDictionary<TKey, TValue>接口和 IDictionary 接口实现同一个枚举器，只要这个枚举器实现了 IEnumerator<KeyValuePair<K, V>>接口和 IDictionaryEnumerator 接口就可以了。前者是 IDictionary<TKey, TValue>接口所需要的枚举器，后者是 IDictionary 接口

所需要的枚举器（见前一节）。下面是我们的这个枚举器所需实现的接口的关系图：

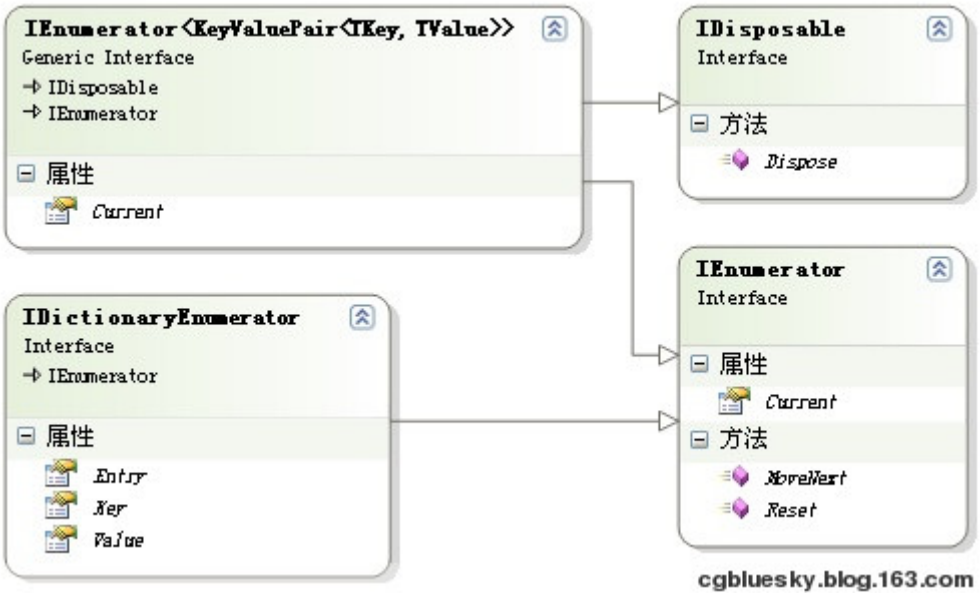

图 3　Enumerator<K, V>类接口关系图

实现这个类需要注意一个问题，在 ReversibleSortedList 类中添加了一个新的成员变量：version。在插入元素时（insert 方法）会更改这个 version 值，表示类中的元素已经进行了改动。为什么要使用它呢？在此我做些推测。从 MSDN 中或许可以找到线索。我们在 MSDN 找到关于 Dictionary<TKey, TValue>类的介绍，这个类跟 ReversibleSortedList 类非常相似。在关于它的线程安全中有这么一段话：

此类型的公共静态成员是线程安全的。但不能保证任何实例成员是线程安全的。

只要不修改该集合，Dictionary<(Of <(TKey, TValue>)>) 就可以同时支持多个阅读器。即便如此，从头到尾对一个集合进行枚举本质上并不是一个线程安全的过程。当出现枚举与写访问互相争用这种极少发生的情况时，必须在整个枚举过程中锁定集合。若要允许多个线程访问集合以进行读写操作，则必须实现自己的同步。

可以推测，version 用于在枚举过程中判断是否有其他线程更改了 ReversibleSortedList 实例中存储的元素，以便弹出异常。这一点可以在稍后的代码中看见。

这时可能有人会问了，前面一节中关没有这种判断啊？好，让我们看看上一节关于 IDictionary 接口的代码。在它的实现枚举器的嵌套类 SimpleDictionaryEnumerator 中，看看它的构造方法：

```
public SimpleDictionaryEnumerator(SimpleDictionary sd)
        {
            items = new DictionaryEntry[sd.Count];
            Array.Copy(sd.items, 0, items, 0, sd.Count);
        }
```

从代码中可以看出，它把外部类中的所有元素拷贝到另一块内存中进行枚举，这样在多线程访问集合时自然不会出错，但如果集合中的元素很多就会带来性能上的损失。而我们实现的 ReversibleSortedList 类将直接使用集合中的元素进行枚举，所以需要使用 version 来保证在出错时可以弹出异常。

下面我们在“ReversibleSortedList 0.3 版本”的基础上继续构建。关于 version 的代码这里不再讲解，请大家查看稍后完整的 0.4 版本的代码。首先添加一个实现枚举器的嵌套类：

```
private struct Enumerator<K, V> : IEnumerator<KeyValuePair<K, V>>, IDisposable,
        IDictionaryEnumerator, IEnumerator
    {
        private ReversibleSortedList<K, V> _ReversibleSortedList;
        private K key;
        private V value;
        private int index;
        private int version;
        internal Enumerator(ReversibleSortedList<K, V> ReversibleSortedList)
        {    //获取外部类中的元素引用，以对它进行枚举
            this._ReversibleSortedList = ReversibleSortedList;
            this.index = 0;
            this.version = this._ReversibleSortedList.version;//记录外部类版本号
            //设置键和值为其默认类型，请参考：
            //http://cgbluesky.blog.163.com/blog/static/241235582008169534082 2/
            this.key = default(K);
            this.value = default(V);
        }
        public void Dispose()
        {
            this.index = 0;
            this.key = default(K);
            this.value = default(V);
        }
        object IDictionaryEnumerator.Key
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "不能进行枚举操作.");
                }
                return this.key;
            }
```

```
        }
        public bool MoveNext()
        {
            if (this.version != this._ReversibleSortedList.version)
            {     //版本检查
                throw new InvalidOperationException("枚举版本检查失败！");
            }
            if (this.index < this._ReversibleSortedList.Count)
            {
                this.key = this._ReversibleSortedList.keys[this.index];
                this.value = this._ReversibleSortedList.values[this.index];
                this.index++;
                return true;
            }
            this.index = this._ReversibleSortedList.Count + 1;
            this.key = default(K);
            this.value = default(V);
            return false;
        }
        DictionaryEntry IDictionaryEnumerator.Entry
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException("不能进行枚举操作.");
                }
                return new DictionaryEntry(this.key, this.value);
            }
        }
        public KeyValuePair<K, V> Current
        {
            get
            {
                return new KeyValuePair<K, V>(this.key, this.value);
            }
        }
        object IEnumerator.Current
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
```

```
                {
                    throw new InvalidOperationException("不能进行枚举操作");
                }
                return new DictionaryEntry(this.key, this.value);
            }
        }
        object IDictionaryEnumerator.Value
        {
            get
            {
                if ((this.index == 0) ||
                    (this.index == (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException("不能进行枚举操作");
                }
                return this.value;
            }
        }
        void IEnumerator.Reset()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException("枚举版本检查失败！");
            }
            this.index = 0;
            this.key = default(K);
            this.value = default(V);
        }
    }
```

这个嵌套类的代码对照图 3 很容易看懂，每个方法的功能在 MSDN 中也有详细的介绍，这里不再对它进行讲解。接下来要给外部类实现 IEnumerable<KeyValuePair<TKey, TValue>> 和 IEnumerable 接口。更改类声明代码如下：

```
public class ReversibleSortedList<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
```

当然，这两个接口分别只有一个成员：GetEnumerator()方法，刚才所创建的嵌套类就是为这个方法所创建的。接下来在 ReversibleSortedList 类中使用显式接口成员实现来实现这两个接口：

```
IEnumerator<KeyValuePair<TKey, TValue>>
        IEnumerable<KeyValuePair<TKey, TValue>>.GetEnumerator()
```

```
    {
        return new Enumerator<TKey, TValue>(this);
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new Enumerator<TKey, TValue>(this);
    }
```

好，现在更改 Main()方法中的代码看看是否可以使用 foreach 循环来访问 ReversibleSortedList 中的元素，当然，前面所写的 Print()成员方法可以退休了。

```
static void Main()
    {
        ReversibleSortedList<int, string> rs=new ReversibleSortedList<int, string>();
        rs.Add(3,"a");
        rs.Add(1,"b");
        rs.Add(2,"c");
        rs.Add(6,"d");
        rs.Add(5,"e");
        rs.Add(4,"f");
        foreach (KeyValuePair<int, string> d in rs)
        {
            Console.WriteLine(d.Key + "       " + d.Value);
        }
    }
```

由于代码已经达到 300 行，贴到博客上会导致运行缓慢，后面所有的可运行代码将以文件的形式给出，大家可以直接下载运行。

ReversibleSortedList 0.4 版本：实现迭代

单击下载完整代码

运行结果：

```
1       b
2       c
3       a
4       f
5       e
6       d
```

真棒，现在已经取得了阶段性的成果。但还有一些遗憾，虽然在 Enumerator 类中实现了 IDictionaryEnumerator 接口，但还不能在 foreach 中使用 DictionaryEntry 访问元素。这是

因为 IDictionary 接口重载了 GetEnumerator()接口，而它返回的是一个 IDictionaryEnumerator 接口，也就是说，只有实现了 IDictionary 接口才能使用 DictionaryEntry 访问其中元素。实现 IDictionary 接口之前我们需要建立一些辅助的内部类，这将在下一节进行讲解。

5.2.6、实现IDictionary接口中的Keys和Values属性

现在我们可以着眼于 IDictionary 接口的实现。第 4 节中，专门针对这个接口做了一个最简化的例子，我们来回顾一下，它是怎么实现 IDictionary 接口中的 Keys 和 Values 属性的。

```
public ICollection Keys
    {    //返回所有键的集合
        get
        {    //把所有键的集合拷贝到新数组中并返回
            Object[] keys = new Object[ItemsInUse];
            for (Int32 n = 0; n < ItemsInUse; n++)
                keys[n] = items[n].Key;
            return keys;
        }
    }
    public ICollection Values
    {    //返回所有值的集合
        get
        {    //把所有值的集合拷贝到新数组中并返回
            Object[] values = new Object[ItemsInUse];
            for (Int32 n = 0; n < ItemsInUse; n++)
                values[n] = items[n].Value;
            return values;
        }
    }
```

可以很清楚地看到，它把数组里的所有元素拷贝到另一块内存空间中并返回，这再一次带来了性能问题，如果频繁地访问 Keys 和 Values 属性还会给垃圾回收带来压力。最好的解决办法当然是直接引用而不是拷贝数组里的元素，你还希望增加一些功能，可以使用索引访问 Keys 属性或 Values 属性所返回的 ICollection。但从第 5 节中的图 2（最好直接下载下来以方便观看）中可以看到 ICollection 接口只有寥寥几个成员，并没有 Item 属性，怎么办呢？当然是从 ICollection 的子接口中寻找合适的接口了。我们知道，ICollection 接口是集合接口的基接口，而它的子接口则是更专用的集合接口，如 IDictionary 表示带有键\值对的集合，IList 表示值的集合，它们都可以按索引访问。

所以这一次你决定另外实现公有的 Keys 和 Values 属性，并返回一个 ILst<T>接口，并手动实现它，一方面满足所有的功能，另一方面也可以实现 IDictionary 和 IDictionary<TKey, TValue>接口的 Keys 和 Values 属性。好，先来看看 ILst<T>接口的关系图：

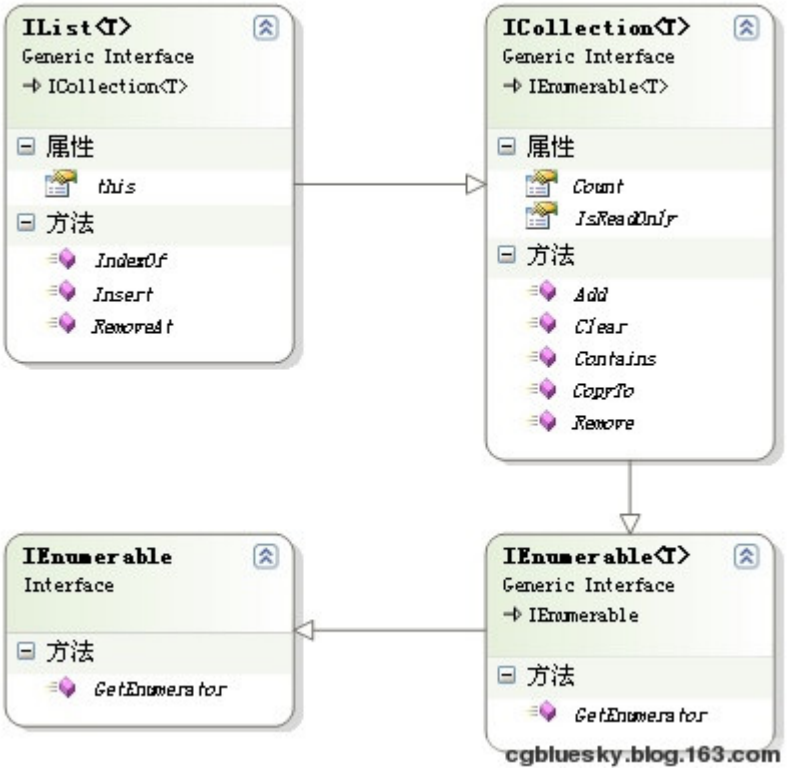

图 4　IList<T>接口关系图

可以看到，实现它并不简单，好在我们将屏蔽所有对元素进行改动的功能。当然，首先还是要要实现一个枚举器，不同的地方在于，这次枚举器跟调用类的关系不再是嵌套关系，它们处于同一层次，都是 ReversibleSortedList 的嵌套类。另外由于引用外部类集合，这里也不需要考虑反向排序的问题。

下面是 Keys 属性枚举时所需要的枚举器：

```
ReversibleSortedListKeyEnumerator#region ReversibleSortedListKeyEnumerator
    private sealed class ReversibleSortedListKeyEnumerator :
        IEnumerator<TKey>, IDisposable, IEnumerator
    {
        // 成员变量
        private ReversibleSortedList<TKey, TValue> _ReversibleSortedList;
        private TKey currentKey; //记录当前值
        private int index; //记录当前索引
        private int version; //记录此类创建时，外部类的版本号
        //构造方法
        internal ReversibleSortedListKeyEnumerator(
                ReversibleSortedList<TKey, TValue> ReversibleSortedList)
        {    //传递外部类元素所在集合的引用
            this._ReversibleSortedList = ReversibleSortedList;
            this.version = ReversibleSortedList.version;
        }
```

```
        public void Dispose()
        {
            this.index = 0;
            this.currentKey = default(TKey);
        }
        public bool MoveNext()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "枚举版本检查错误");
            }
            if (this.index < this._ReversibleSortedList.Count)
            {
                this.currentKey = this._ReversibleSortedList.keys[this.index];
                this.index++;
                return true;
            }
            this.index = this._ReversibleSortedList.Count + 1;
            this.currentKey = default(TKey);
            return false;
        }
        void IEnumerator.Reset()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "枚举版本检查错误");
            }
            this.index = 0;
            this.currentKey = default(TKey);
        }
        // 属性
        public TKey Current
        {
            get
            {
                return this.currentKey;
            }
        }
        object IEnumerator.Current
        {
            get
            {
```

```
                if ((this.index == 0) || (this.index ==
                        (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "不能进行枚举操作");
                }
                return this.currentKey;
            }
        }
    }
    #endregion
```

同理，Values 属性枚举时所需要的枚举器代码类似：

```
ReversibleSortedListValueEnumerator#region ReversibleSortedListValueEnumerator
    private sealed class ReversibleSortedListValueEnumerator :
        IEnumerator<TValue>, IDisposable, IEnumerator
    {
        // 成员变量
        private ReversibleSortedList<TKey, TValue> _ReversibleSortedList;
        private TValue currentValue;
        private int index;
        private int version;
        internal ReversibleSortedListValueEnumerator(
                        ReversibleSortedList<TKey, TValue> ReversibleSortedList)
        {
            this._ReversibleSortedList = ReversibleSortedList;
            this.version = ReversibleSortedList.version;
        }
        public void Dispose()
        {
            this.index = 0;
            this.currentValue = default(TValue);
        }
        public bool MoveNext()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "Enumeration failed version check");
            }
            if (this.index < this._ReversibleSortedList.Count)
            {
                this.currentValue = this._ReversibleSortedList.values[this.index];
```

```
                this.index++;
                return true;
            }
            this.index = this._ReversibleSortedList.Count + 1;
            this.currentValue = default(TValue);
            return false;
        }
        void IEnumerator.Reset()
        {
            if (this.version != this._ReversibleSortedList.version)
            {
                throw new InvalidOperationException(
                    "Enumeration failed version check");
            }
            this.index = 0;
            this.currentValue = default(TValue);
        }
        // 属性
        public TValue Current
        {
            get
            {
                return this.currentValue;
            }
        }
        object IEnumerator.Current
        {
            get
            {
                if ((this.index == 0) || (this.index ==
                        (this._ReversibleSortedList.Count + 1)))
                {
                    throw new InvalidOperationException(
                            "Enumeration operation could not occur");
                }
                return this.currentValue;
            }
        }
    }
    #endregion
```

枚举器实现了，接下来实现 Keys 所需要的 IList<T>。注意，这里通过弹出异常屏蔽了所有企图对元素进行修改的操作。

```
KeyListK,V#region KeyList<K,V>
private sealed class KeyList<K, V> : IList<K>, ICollection<K>,IEnumerable<K>,
ICollection, IEnumerable
    {
        // 成员变量
        private ReversibleSortedList<K, V> _dict;
        //成员方法
        internal KeyList(ReversibleSortedList<K, V> dictionary)
        {
            this._dict = dictionary;
        }
        public void Add(K key)
        {
            throw new NotSupportedException("不支持 Add 操作");
        }
        public void Clear()
        {
            throw new NotSupportedException("不支持 Clear 操作");
        }
        public bool Contains(K key)
        {
            return this._dict.ContainsKey(key);
        }
        public void CopyTo(K[] array, int arrayIndex)
        {
            Array.Copy(this._dict.keys, 0, array, arrayIndex, this._dict.Count);
        }
        public IEnumerator<K> GetEnumerator()
        {
            return new
                    ReversibleSortedList<K, V>.ReversibleSortedListKeyEnumerator(
                                                                this._dict);
        }
        public int IndexOf(K key)
        {
            if (key == null)
            {
                throw new ArgumentNullException("key");
            }
            int num1 = Array.BinarySearch<K>(this._dict.keys, 0,this._dict.Count, key,
                                                        this._dict._sortDirectionComparer);
            if (num1 >= 0)
            {
                return num1;
```

```
        }
        return -1;
    }
    public void Insert(int index, K value)
    {
        throw new NotSupportedException("不支持 Insert 操作");
    }
    public bool Remove(K key)
    {
        return false;
    }
    public void RemoveAt(int index)
    {
        throw new NotSupportedException("不支持 RemoveAt 操作");
    }
    void ICollection.CopyTo(Array array, int arrayIndex)
    {
        if ((array != null) && (array.Rank != 1))
        {
            throw new ArgumentException(
                    "不支持多维数组");
        }
        try
        {
            Array.Copy(this._dict.keys, 0, array, arrayIndex,
                        this._dict.Count);
        }
        catch (ArrayTypeMismatchException atme)
        {
            throw new ArgumentException("错误的数组类型", atme);
        }
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new
                ReversibleSortedList<K, V>.ReversibleSortedListKeyEnumerator(
                    this._dict);
    }
    // 属性
    public int Count
    {
        get
        {
            return this._dict._size;
```

```
            }
        }
        public bool IsReadOnly
        {
            get
            {
                return true;
            }
        }
        public K this[int index]
        {
            get
            {
                return this._dict.GetKey(index);
            }
            set
            {
                throw new NotSupportedException("不支持通过索引进行更改的操作");
            }
        }
        bool ICollection.IsSynchronized
        {
            get
            {
                return false;
            }
        }
        object ICollection.SyncRoot
        {
            get
            {
                return this._dict;
            }
        }
    }
    #endregion
```

同理，Values 所需要的 Llist<T>代码相似：

```
ValueList K, V definition#region ValueList <K, V> definition
    private sealed class ValueList<K, V> : IList<V>, ICollection<V>,
        IEnumerable<V>, ICollection, IEnumerable
    {
        // 成员变量
```

```
private ReversibleSortedList<K, V> _dict;
// 成员方法
internal ValueList(ReversibleSortedList<K, V> dictionary)
{
    this._dict = dictionary;
}
public void Add(V key)
{
    throw new NotSupportedException("不支持 Add 操作");
}
public void Clear()
{
    throw new NotSupportedException("不支持 Clear 操作");
}
public bool Contains(V value)
{
    return this._dict.ContainsValue(value);
}
public void CopyTo(V[] array, int arrayIndex)
{
    Array.Copy(this._dict.values, 0, array, arrayIndex, this._dict.Count);
}
public IEnumerator<V> GetEnumerator()
{
    return new
        ReversibleSortedList<K, V>.ReversibleSortedListValueEnumerator(
        this._dict);
}
public int IndexOf(V value)
{
    return Array.IndexOf<V>(this._dict.values, value, 0, this._dict.Count);
}
public void Insert(int index, V value)
{
    throw new NotSupportedException("不支持 Insert 操作");
}
public bool Remove(V value)
{
    return false;
}
public void RemoveAt(int index)
{
    throw new NotSupportedException("不支持 RemoveAt 操作");
}
```

```
        void ICollection.CopyTo(Array array, int arrayIndex)
        {
            if ((array != null) && (array.Rank != 1))
            {
                throw new ArgumentException(
                    "不支持多维数组");
            }
            try
            {
                Array.Copy(this._dict.values, 0, array, arrayIndex,
                           this._dict.Count);
            }
            catch (ArrayTypeMismatchException atme)
            {
                throw new ArgumentException("错误的数组类型", atme);
            }
        }
        IEnumerator IEnumerable.GetEnumerator()
        {
            return new
                    ReversibleSortedList<K, V>.ReversibleSortedListValueEnumerator(
                    this._dict);
        }
        // 属性
        public int Count
        {
            get
            {
                return this._dict._size;
            }
        }
        public bool IsReadOnly
        {
            get
            {
                return true;
            }
        }
        public V this[int index]
        {
            get
            {
                return this._dict.GetByIndex(index);
            }
```

```
            set
            {
                throw new NotSupportedException("不支持通过索引改变元素值");
            }
        }
        bool ICollection.IsSynchronized
        {
            get
            {
                return false;
            }
        }
        object ICollection.SyncRoot
        {
            get
            {
                return this._dict;
            }
        }
    }
    #endregion
```

上面两个 IList<T>接口代码调用了外部类的一些方法，需要把它们添加到 ReversibleSortedList 类中：

公有方法如下：

```
 //查找指定键索引
    public int IndexOfKey(TKey key)
    {

        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,
                                                        this._sortDirectionComparer);
        if (num1 < 0)
        {
            return -1;
        }
        return num1;
    }
    //查找指定值索引
```

```
    public int IndexOfValue(TValue value)
    {
        return Array.IndexOf<TValue>(this.values, value, 0, this._size);
    }
    //判断是否包含指定键
    public bool ContainsKey(TKey key)
    {
        return (this.IndexOfKey(key) >= 0);
    }
    //判断是否包含指定值
    public bool ContainsValue(TValue value)
    {
        return (this.IndexOfValue(value) >= 0);
    }
```

私有方法如下：

```
    private TValue GetByIndex(int index)
    {
        if ((index < 0) || (index >= this._size))
        {
            throw new ArgumentOutOfRangeException("index", "Index out of range");
        }
        return this.values[index];
    }
    //返回指定索引的键
    private TKey GetKey(int index)
    {
        if ((index < 0) || (index >= this._size))
        {
            throw new ArgumentOutOfRangeException("index", "Index out of range");
        }
        return this.keys[index];
    }
    private void Insert(int index, TKey key, TValue value)
    {    //在指定索引入插入数据
        if (this._size == this.keys.Length)
        {
            this.EnsureCapacity(this._size + 1);
        }
        if (index < this._size)
        {    //当插入元素不是添加在未尾时，移动插入点后面的元素
            Array.Copy(this.keys, index, this.keys, (int)(index + 1),
                        (int)(this._size - index));
            Array.Copy(this.values, index, this.values, (int)(index + 1),
                        (int)(this._size - index));
```

```
        }
        this.keys[index] = key; //在插入点插入键
        this.values[index] = value; //在插入点插入值
        this._size++;
        this.version++;
    }
    private KeyList<TKey, TValue> GetKeyListHelper()
    {   //获取键的 IList<T>集合
        if (this.keyList == null)
        {
            this.keyList = new KeyList<TKey, TValue>(this);
        }
        return this.keyList;
    }
    private ValueList<TKey, TValue> GetValueListHelper()
    {   //获取值的 IList<T>集合
        if (this.valueList == null)
        {
            this.valueList = new ValueList<TKey, TValue>(this);
        }
        return this.valueList;
    }
```

写了这么多代码，终于可以实现 Keys 和 Values 属性了。

首先添加这两个属性所需的成员变量：

```
    private KeyList<TKey, TValue> keyList;
    private ValueList<TKey, TValue> valueList;
```

然后实现 Keys 和 Values 属性，以上这么多代码，就是为了这两个属性：

```
    public IList<TKey> Keys
    {
        get
        {
            return this.GetKeyListHelper();
        }
    }

    public IList<TValue> Values
    {
        get
```

```
        {
            return this.GetValueListHelper();
        }
    }
```

看到这，你是不是有些不耐烦了，反正我是有这样的感觉了。但有一点你必须明白，FCL 里的代码就是这么实现的。

好，做了这么多工作，终于可以看看成果，测试一下是否可用了。在 Main 方法添加如下代码：

```
    static void Main()
    {
        ReversibleSortedList<int, string> rs = new ReversibleSortedList<int, string>();
        //添加元素
        rs.Add(3, "a");
        rs.Add(1, "b");
        rs.Add(2, "c");
        rs.Add(6, "d");
        rs.Add(5, "e");
        rs.Add(4, "f");
        //打印键/值
        foreach (KeyValuePair<int, string> d in rs)
        {
            Console.WriteLine(d.Key + "       " + d.Value);
        }
        //打印键，这里访问了 Keys 属性
        foreach (int i in rs.Keys)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();//换行
        //打印值，这里访问了 Values 属性
        foreach (string s in rs.Values)
        {
            Console.Write(s + " ");
        }
        Console.WriteLine();//换行
        Console.WriteLine(rs.Keys[3]); //通过索引访问键
        Console.WriteLine(rs.Values[4]); //通过索引访问值
    }
```

上帝啊！太痛苦了，如果你也有这样的感觉，请下载现成的代码，更改 Main()方法里的

东西，尝试是否能通过 Keys 和 Values 属性更改元素。

ReversibleSortedList 0.5 版本：实现 Keys 和 Values 属性

完整代码下载

运行结果：

```
1    b
2    c
3    a
4    f
5    e
6    d
1 2 3 4 5 6
b c a f e d
4
e
```

5.2.7、实现IDictionary接口

前面做了很多努力，现在终于可以实现 IDictionary 接口了。当然，之所以要先实现它，目的之一还是为了之前留下的一点遗憾：在 foreach 中使用 DictionaryEntry 访问集合中的元素。

需要注意，由于 ReversibleSortedList 类最主要的接口是泛型 IDictionary 接口，实现非泛型 IDictionary 接口主要是考虑到兼容性，试想，你的项目是用.NET 1.0 实现的，但现在你需要使用.NET 2.0 继续完善程序并使用到了一些.NET 2.0 所独有的功能。但你并不想更改之前曾写好并稳定运行了很久的程序，此时，兼容是非常重要的。，IDictionary 接口成员大部份是显式接口成员实现。这一节请对照第 5 节的图 2 观看，最好下载用图片浏览器打开，我们先改变 ReversibleSortedList 类的声明，加上 IDictionary：

```
public class ReversibleSortedList<TKey, TValue> :
IDictionary, IEnumerable<KeyValuePair<TKey, TValue>>, ICollection, IEnumerable
```

从 ICollection 接口开始实现：

由于它的 Count 属性只是指示一个整数，可以和 ICollection<KeyValuePain<TKey, TVlaue>>共用，并且前面已经实现了它，所以无需再另外实现。剩下的就只是 IsSynchnonised 、 SyncRoot 属性和 CopyTo 方法了，但 CopyTo 方法 ICollection<KeyValuePain<TKey, TVlaue>>接口中也有，在 MSDN 中查看两个方法的定义：

ICollection 中的是：

```
void CopyTo(       Array array,       int index)
```

而 ICollection<KeyValuePain<TKey, TVlaue>>中的是

```
void CopyTo(T[] array, int arrayIndex)
```

可以看到 array 参数并不一样，一个是泛型，一个是数组，所以需要为这个方法实现两个版本，当然，我们先实现的是 ICollection 接口中的 CopyTo 方法：

```
void ICollection.CopyTo(Array array, int arrayIndex)
    {
        if (array == null)
        {
            throw new ArgumentNullException("array");
        }
        if (array.Rank != 1)
        {
            throw new ArgumentException(
                "不支持多维数组拷贝");
        }
        if (array.GetLowerBound(0) != 0)
        {   //检查参数数组是否下限为
            throw new ArgumentException("A non-zero lower bound was provided");
        }
        if ((arrayIndex < 0) || (arrayIndex > array.Length))
        {   //检查参数数组容量
            throw new ArgumentOutOfRangeException(
                "arrayIndex", "Need non negative number");
        }
        if ((array.Length - arrayIndex) < this.Count)
        {   //检查参数数组容量
            throw new ArgumentException("Array plus the offset is too small");
        }
        //把参数 array 强制转换为 KeyValuePair<TKey, TValue>数组类型
        KeyValuePair<TKey, TValue>[] pairArray1 =
            array as KeyValuePair<TKey, TValue>[];
        if (pairArray1 != null)
        {   //如果转换成功，则拷贝数据
            for (int num1 = 0; num1 < this.Count; num1++)
            {
                pairArray1[num1 + arrayIndex] =
                    new KeyValuePair<TKey, TValue>(this.keys[num1],
                    this.values[num1]);
```

```
                }
            }
            else
            {    //如果转换不成功，则把参数 array 强制转化为 object 数组
                object[] objArray1 = array as object[];
                if (objArray1 == null)
                {
                    throw new ArgumentException("错误的数组类型");
                }
                try
                {
                    for (int num2 = 0; num2 < this.Count; num2++)
                    {
                        objArray1[num2 + arrayIndex] =
                                new KeyValuePair<TKey, TValue>(this.keys[num2],

this.values[num2]);
                    }
                }
                catch (ArrayTypeMismatchException)
                {
                    throw new ArgumentException("错误的数组类型");
                }
            }
        }
```

下面实现了 ICollection 接口的两个私有属性：

```
bool ICollection.IsSynchronized
        {
            get
            {
                return false;
            }
        }
        object ICollection.SyncRoot
        {
            get
            {
                return this;
            }
        }
```

好，现在开始正式实现 IDictionary 接口。首先是属性：

```
bool IDictionary.IsFixedSize
{
    get
    {
        return false;
    }
}
bool IDictionary.IsReadOnly
{
    get
    {
        return false;
    }
}
//item 属性，本质上就是索引器
object IDictionary.this[object key]
{
    get
    {
        if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
        {
            int num1 = this.IndexOfKey((TKey)key);
            if (num1 >= 0)
            {
                return this.values[num1];
            }
        }
        return null;
    }
    set
    {
        ReversibleSortedList<TKey, TValue>.VerifyKey(key);
        ReversibleSortedList<TKey, TValue>.VerifyValueType(value);
        //这里调用了 IDictionary<TKey, TValue>接口的 Item 属性，由于
        //还没有实现它，所以先把这句屏敝掉
        //this[(TKey)key] = (TValue)value;
    }
}
ICollection IDictionary.Keys
{
    get
    {
        return this.GetKeyListHelper();
```

```
        }
    }
    ICollection IDictionary.Values
    {
        get
        {
            return this.GetValueListHelper();
        }
    }
```

首先注意，全部使用了显式接口成员实现，也就是说，只能通过接口调用这些方法。另外它的 Item 属性调用了还未实现的 IDictionary<TKey, TValue>接口的 Item 属性：

```
this[(TKey)key] = (TValue)value;
```

所以需要先把这句屏蔽掉，等最后再释放出来。另外调用了两个新的私有方法，也需要加进去：

```
private static void VerifyKey(object key)
    {   //检查 key 类型是否兼容 TKey
        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        if (!(key is TKey))
        {   //检查 key 是否和 TKey 兼容。注意，此时 TKey 已经被替换为实际类型
            throw new ArgumentException(
                "参数类型错误", "key");
        }
    }
    private static void VerifyValueType(object value)
    {   //检查 value 类型
        if (!(value is TValue) && ((value != null) || typeof(TValue).IsValueType))
        {
            throw new ArgumentException(
                "参数类型错误", "value");
        }
    }
```

VerifyValueType 方法进行类型检查时有一个奇怪的判断，这一点主要是针对 C#2.0 的可空类型而设的。也就是说，在这里，空的值类型是合法的。关于可空类型，可以参考：

http://cgbluesky.blog.163.com/blog/static/241235582008111129264 23/

下面，就剩下 IDictionary 接口成员方法没有实现了。其中，由于 Clear()方法同时为 IDictionary 和 ICollection <KeyValuePair<TKey, TValue>>的成员方法，但两者的返回值和参数完全相同，所以可以共用一个拷贝：

```
public void Clear()
    {
        this.version++;
        Array.Clear(this.keys, 0, this._size);
        Array.Clear(this.values, 0, this._size);
        this._size = 0;
    }
```

剩下的全部为显式接口成员实现：

```
void IDictionary.Add(object key, object value)
    {    //做类型检查再添加
        ReversibleSortedList<TKey, TValue>.VerifyKey(key);
        ReversibleSortedList<TKey, TValue>.VerifyValueType(value);
        this.Add((TKey)key, (TValue)value);
    }
    bool IDictionary.Contains(object key)
    {
        if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
        {
            return this.ContainsKey((TKey)key);
        }
        return false;
    }
    IDictionaryEnumerator IDictionary.GetEnumerator()
    {
        return new ReversibleSortedList<TKey, TValue>.Enumerator<TKey, TValue>(
                this);
    }
    void IDictionary.Remove(object key)
    {
        if (ReversibleSortedList<TKey, TValue>.IsCompatibleKey(key))
        {   //这里调用了 IDictionary<TKey, TValue>接口的 Remove 方法，由于
            //还没有实现它，所以先把这句屏敝掉
            // this.Remove((TKey)key);
        }
    }
```

上面代码调用了 IsCompatibleKey(key)方法，需要把它加上去：

```
private static bool IsCompatibleKey(object key)
    {    //用于检查键的类型并返回一个 bool 值
        if (key.Equals(null))
        {
            throw new ArgumentNullException("key");
        }
        return (key is TKey);
    }
```

其中，IDictionaryEnumerator IDictionary.GetEnumerator()方法是我们期待已久的，实现了它就可以在 foreach 中使用 DictionaryEntry 访问集合中的元素。另外 Remove 方法调用了 IDictionary<TKey, TValue>接口的 Remove 方法。当然，我们还未实现它，这是第二次这样了，看样子还是首先应该实现泛型接口再实现非泛型接口。教训惨痛啊！写了这么多，懒得改了，大家知道就行了。

好！终于完成了，现在终于可以看看成果了。改变 Main 方法如下：

```
    static void Main()
    {
        ReversibleSortedList<int, string> rs = new ReversibleSortedList<int, string>();
        //添加元素
        rs.Add(3, "a");
        rs.Add(1, "b");
        rs.Add(2, "c");
        rs.Add(6, "d");
        rs.Add(5, "e");
        rs.Add(4, "f");
        //使用 DictionaryEntry 打印键/值
        foreach (DictionaryEntry d in (IDictionary)rs)
        {
            Console.WriteLine(d.Key + "       " + d.Value);
        }
}
```

ReversibleSortedList 0.6 版本：实现 IDictionary 接口

完整代码下载

运行结果：

```
        1    b
        2    c
```

```
3    a
4    f
5    e
6    d
```

很遗憾，由于 IDictionary.GetEnumerator()是显式接口成员实现，只能使用接口调用。接下来，终于可以实现最重要的一个接口 IDictionary<TKey, TValue>了。

5.2.8、实现IDictionary<TKey, TValue>接口

由于前面实现了 IDictionary 接口，现在实现 IDictionary<TKey, TValue>也就没什么困难的了，照葫芦画瓢。

首先改变类声明：

```
public class ReversibleSortedList<TKey, TValue> :IDictionary<TKey, TValue>
    IDictionary, IEnumerable<KeyValuePair<TKey, TValue>>, ICollection,
IEnumerable, ICollection<KeyValuePair<TKey, TValue>>
```

然后实现 ICollection<KeyValuePair<TKey, TValue>>接口成员：

```
bool ICollection<KeyValuePair<TKey, TValue>>.IsReadOnly
    {
        get
        {
            return false;
        }
    }
    void ICollection<KeyValuePair<TKey, TValue>>.Add(
        KeyValuePair<TKey, TValue> keyValuePair)
    {
        this.Add(keyValuePair.Key, keyValuePair.Value);
    }
    bool ICollection<KeyValuePair<TKey, TValue>>.Contains(
        KeyValuePair<TKey, TValue> keyValuePair)
    {
        int num1 = this.IndexOfKey(keyValuePair.Key);
        if ((num1 >= 0) && EqualityComparer<TValue>.Default.Equals(this.values[num1],
            keyValuePair.Value))
        {
            return true;
        }
        return false;
```

```
    }
    void ICollection<KeyValuePair<TKey, TValue>>.CopyTo(
        KeyValuePair<TKey, TValue>[] array, int arrayIndex)
    {
        if (array == null)
        {
            throw new ArgumentNullException("array");
        }
        if ((arrayIndex < 0) || (arrayIndex > array.Length))
        {
            throw new ArgumentOutOfRangeException(
                    "arrayIndex", "Need a non-negative number");
        }
        if ((array.Length - arrayIndex) < this.Count)
        {
            throw new ArgumentException("ArrayPlusOffTooSmall");
        }
        for (int num1 = 0; num1 < this.Count; num1++)
        {
            KeyValuePair<TKey, TValue> pair1;
            pair1 = new KeyValuePair<TKey, TValue>(
                        this.keys[num1], this.values[num1]);
            array[arrayIndex + num1] = pair1;
        }
    }
    bool ICollection<KeyValuePair<TKey, TValue>>.Remove(
        KeyValuePair<TKey, TValue> keyValuePair)
    {
        int num1 = this.IndexOfKey(keyValuePair.Key);
        if ((num1 >= 0) && EqualityComparer<TValue>.Default.Equals(
            this.values[num1], keyValuePair.Value))
        {
            this.RemoveAt(num1);
            return true;
        }
        return false;
}
```

需要注意，Count 属性和 Clear 方法都是共用方法，不需要再次实现。

接下来实现 IDictionary<TKey, TValue>接口，它是最主要的接口，所有成员均为公有的，前面我们已经实现了 Add、ContainsKey 方法，这里不再列出：

```
    ICollection<TKey> IDictionary<TKey, TValue>.Keys
```

```
{
    get
    {
        return this.GetKeyListHelper();
    }
}
ICollection<TValue> IDictionary<TKey, TValue>.Values
{
    get
    {
        return this.GetValueListHelper();
    }
}
public TValue this[TKey key] //Item 属性，就是索引器
{
    get
    {
        TValue local1;
        int num1 = this.IndexOfKey(key);
        if (num1 >= 0)
        {
            return this.values[num1];
        }
        else
        {
            local1 = default(TValue);
            return local1;
        }
    }
    set
    {
        if (key == null)
        {
            throw new ArgumentNullException("key");
        }
        int num1 = Array.BinarySearch<TKey>(this.keys, 0, this._size, key,
            this._sortDirectionComparer);
        if (num1 >= 0)
        {
            this.values[num1] = value;
            this.version++;
        }
        else
        {
```

```
                    this.Insert(~num1, key, value);
                }
            }
        }
        public bool Remove(TKey key)
        {
            int num1 = this.IndexOfKey(key);
            if (num1 >= 0)
            {
                this.RemoveAt(num1);
            }
            return (num1 >= 0);
        }
        public bool TryGetValue(TKey key, out TValue value)
        {
            int num1 = this.IndexOfKey(key);
            if (num1 >= 0)
            {
                value = this.values[num1];
                return true;
            }
            value = default(TValue);
            return false;
        }
```

另外需要添加一个 RemoveAt 方法：

```
        //移除指定索引元素
        public void RemoveAt(int index)
        {
            if ((index < 0) || (index >= this._size))
            {
                throw new ArgumentOutOfRangeException("index", "Index out of range");
            }
            this._size--;
            if (index < this._size)
            {
                Array.Copy(this.keys, (int)(index + 1), this.keys, index,
                                (int)(this._size - index));
                Array.Copy(this.values, (int)(index + 1), this.values, index,
                                (int)(this._size - index));
            }
            this.keys[this._size] = default(TKey);
            this.values[this._size] = default(TValue);
```

```
            this.version++;
        }
```

最后，进行测试，这一次不再使用 int 做为键，而改用 string：

```
static void Main()
    {
        ReversibleSortedList<string, string> rs = new ReversibleSortedList<string, string>();
        //添加元素
        rs.Add("3", "a");
        rs.Add("1", "b");
        rs.Add("2", "c");
        rs.Add("6", "d");
        rs.Add("5", "e");
        rs.Add("4", "f");
        //使用 DictionaryEntry 打印键/值
        foreach (KeyValuePair<string, string> d in rs)
        {
            Console.WriteLine(d.Key + "       " + d.Value);
        }
        Console.WriteLine("删除索引为“2”的“5”的元素");
        rs.Remove("2");
        rs.Remove("5");
        foreach (KeyValuePair<string, string> d in rs)
        {
            Console.WriteLine(d.Key + "       " + d.Value);
        }
}
```

ReversibleSortedList 0.7 版本：实现 IDictionary<TKey, TValue>接口

完整代码下载

运行结果：

```
1    b
2    c
3
4    f
5    e
6    d
```

删除索引为“2”的“5”的元素

```
1    b
3    a
4    f
6    d
```

5.2.9、完善

大楼已经盖好，剩下的工作就是装修，装修好就可以入住了。从本文的题目得知，这是一个可反转排序的集合类，但我们只实现了降序插入功能，如果希望把升序转换为降序该怎么办呢？此例的解决方法是声明一个代表排序方向的属性 Comparer，并加入一个 sort 方法，调用 sort 方法时根据 Comparer 属性进行排序：

```
private ListSortDirection _currentSortDirection = ListSortDirection.Descending;
public SortDirectionComparer<TKey> Comparer
    {
        get
        {
            return this._sortDirectionComparer;
        }
    }
    public void Sort()
    {
        // 检查是否跟现有排序方向相同.
        if (this._currentSortDirection != this._sortDirectionComparer.SortDirection)
        {
            // 如果不同，则进行反转.
            Array.Reverse(this.keys, 0, this._size);
            Array.Reverse(this.values, 0, this._size);
            // 设置当前排序.
            this._currentSortDirection = this._sortDirectionComparer.SortDirection;
        }
    }
```

其中 SortDirectionComparer 类是第二节所声明的类，请参考：

http://cgbluesky.blog.163.com/blog/static/241235582008113103320661/

接下来再增加一个剪除多余空间的功能：

```
//剪除多余空间
    public void TrimExcess()
    {
```

```
            int num1 = (int)(this.keys.Length * 0.9);
            if (this._size < num1)
            {
                this.Capacity = this._size;
            }
}
```

当然，需要给 Capacity 属性添加 set 方法

```
public int Capacity //容量属性
    {
        get
        {
            return this.keys.Length;
        }
        set
        {
            this.InternalSetCapacity(value, true);
        }
    }
```

注意：这样的调整空间会导致另外开辟内存以重新存放元素。

好，最后的工作就是增加一些构造方法，比如指定排序方向，指定容量，以使得这个集合类的使用更为灵活：

```
 //用于指定排序方向的构造方法
    public ReversibleSortedList(SortDirectionComparer<TKey> comparer)
        : this()
    {
        if (comparer != null)
        {
            this._sortDirectionComparer = comparer;
            this._currentSortDirection = _sortDirectionComparer.SortDirection;
        }
    }
    //用于指定字典的构造方法
    public ReversibleSortedList(IDictionary<TKey, TValue> dictionary)
        : this(dictionary, (SortDirectionComparer<TKey>)null)
    {
    }
```

```
    //用于指定初始容量的构造方法
    public ReversibleSortedList(int capacity)
    {
        if (capacity < 0)
        {
            throw new ArgumentOutOfRangeException(
                "capacity", "Non-negative number required");
        }
        this.keys = new TKey[capacity];
        this.values = new TValue[capacity];
        this._sortDirectionComparer = new SortDirectionComparer<TKey>();
        this._currentSortDirection = _sortDirectionComparer.SortDirection;
    }
    //用于指定字典和排序方向的构造方法
    //这个构造方法用于在指定集合中创建新的字典类
    public ReversibleSortedList(IDictionary<TKey, TValue> dictionary,
                                        SortDirectionComparer<TKey> comparer)
        : this((dictionary != null) ? dictionary.Count : 0, comparer)
    {
        if (dictionary == null)
        {
            throw new ArgumentNullException("dictionary");
        }
        dictionary.Keys.CopyTo(this.keys, 0);
        dictionary.Values.CopyTo(this.values, 0);
        Array.Sort<TKey, TValue>(this.keys, this.values,
                                                this._sortDirectionComparer);
        this._size = dictionary.Count;
    }
    //用于指定容量和排序方向的构造方法
    public ReversibleSortedList(int capacity, SortDirectionComparer<TKey> comparer)
        : this(comparer)
    {
        this.Capacity = capacity;
    }
```

好！添加测试代码：

```
static void Main()
    {
        ReversibleSortedList<string, string> rs = new ReversibleSortedList<string, string>();
        //添加元素
        rs.Add("3", "a");
        rs.Add("1", "b");
```

```
            rs.Add("2", "c");
            rs.Add("6", "d");
            rs.Add("5", "e");
            rs.Add("4", "f");
            //使用 DictionaryEntry 打印键/值
            foreach (KeyValuePair<string, string> d in rs)
            {
                Console.WriteLine(d.Key + "       " + d.Value);
            }
            Console.WriteLine("重新排序");
            rs.Comparer.SortDirection = ListSortDirection.Ascending;
            rs.Sort();
            foreach (KeyValuePair<string, string> d in rs)
            {
                Console.WriteLine(d.Key + "       " + d.Value);
            }
        }
```

下载完整代码：

运行结果：

ReversibleSortedList 1.0 版本：完成

```
1    b
2    c
3    a
4    f
5    e
6    d
```

重新排序

```
6    a
5    e
4    f
3    a
2    c
1    b
```

10. 结束语

从翻译《C# Cookbook》中的泛型内容到翻译《Programming C#》中的泛型内容，再到

写完这篇文章，一共写了 129 页的 Word 文档。当然，这里有很大一部份是代码。写到后面我自己都有些不耐烦了。呵呵，不管怎么说，能坚持做完一件事并不容易，现在坚持下来了总算是对自己有个交待，也值得庆贺。

读完这一系列文章，您应该对 FCL 中几个重要的集合接口已经非常熟悉了吧。现在去看 FCL 中几个集合类的源代码将不再困难，还犹豫什么，阅读源代码将会给您带来极大的提高！