集合是OOP中的一个重要概念，C#中对集合的全面支持更是该语言的精华之一。

    为什么要用泛型集合？

    在C# 2.0之前，主要可以通过两种方式实现集合：

    a.使用ArrayList

    直接将对象放入ArrayList，操作直观，但由于集合中的项是Object类型，因此每次使用都必须进行繁琐的类型转换。

    b.使用自定义集合类

    比较常见的做法是从CollectionBase抽象类继承一个自定义类，通过对IList对象进行封装实现强类型集合。这种方式要求为每种集合类型写一个相应的自定义类，工作量较大。泛型集合的出现较好的解决了上述问题，只需一行代码便能创建指定类型的集合。

    什么是泛型？

    泛型是C# 2.0中的新增元素(C++中称为模板)，主要用于解决一系列类似的问题。这种机制允许将类名作为参数传递给泛型类型，并生成相应的对象。将泛型(包括类、接口、方法、委托等)看作模板可能更好理解，模板中的变体部分将被作为参数传进来的类名称所代替，从而得到一个新的类型定义。泛型是一个比较大的话题，在此不作详细解析，有兴趣者可以查阅相关资料。

    怎样创建泛型集合？

    主要利用System.Collections.Generic命名空间下面的List<T>泛型类创建集合，语法如下：

定义Person类如下：

   可以看到，泛型集合大大简化了集合的实现代码，通过它，可以轻松创建指定类型的集合。非但如此，泛型集合还提供了更加强大的功能，下面看看其中的排序及搜索。

List<T> ListOfT = new List<T>();

其中的"T"就是所要使用的类型，既可以是简单类型，如string、int，也可以是用户自定义类型。下面看一个具体例子。

class Person

{

    private string \_name; //姓名

    private int \_age; //年龄

    //创建Person对象

    public Person(string Name, int Age)

    {

        this.\_name= Name;

        this.\_age = Age;

    }

    //姓名

    public string Name

    {

        get { return \_name; }

    }

    //年龄

    public int Age

    {

        get { return \_age; }

    }

}

//创建Person对象

Person p1 = new Person("张三", 30);

Person p2 = new Person("李四", 20);

Person p3 = new Person("王五", 50);

//创建类型为Person的对象集合

List<Person> persons = new List<Person>();

//将Person对象放入集合

persons.Add(p1);

persons.Add(p2);

persons.Add(p3);

//输出第2个人的姓名

Console.Write(persons[1].Name);

    泛型集合的排序

    排序基于比较，要排序，首先要比较。比如有两个数1、2，要对他们排序，首先就要比较这两个数，根据比较结果来排序。如果要比较的是对象，情况就要复杂一点，比如对Person对象进行比较，则既可以按姓名进行比较，也可以按年龄进行比较，这就需要确定比较规则。一个对象可以有多个比较规则，但只能有一个默认规则，默认规则放在定义该对象的类中。默认比较规则在CompareTo方法中定义，该方法属于IComparable<T>泛型接口。请看下面的代码：

class Person ：IComparable<Person>

{

    //按年龄比较

    public int CompareTo(Person p)

    {

        return this.Age - p.Age;

    }

}

    CompareTo方法的参数为要与之进行比较的另一个同类型对象，返回值为int类型，如果返回值大于0，表示第一个对象大于第二个对象，如果返回值小于0,表示第一个对象小于第二个对象，如果返回0,则两个对象相等。

定义好默认比较规则后，就可以通过不带参数的Sort方法对集合进行排序，如下所示:

//按照默认规则对集合进行排序

persons.Sort();

//输出所有人姓名

foreach (Person p in persons)

{

    Console.WriteLine(p.Name); //输出次序为"李四"、"张三"、"王五"

}

    实际使用中，经常需要对集合按照多种不同规则进行排序，这就需要定义其他比较规则，可以在Compare方法中定义，该方法属于IComparer<T>泛型接口，请看下面的代码：

class NameComparer : IComparer<Person>

{

    //存放排序器实例

    public static NameComparer Default = new NameComparer();

    //按姓名比较

    public int Compare(Person p1, Person p2)

    {

        return System.Collections.Comparer.Default.Compare(p1.Name, p2.Name);

    }

}

    Compare方法的参数为要进行比较的两个同类型对象，返回值为int类型，返回值处理规则与CompareTo方法相同。其中的Comparer.Default返回一个内置的Comparer对象，用于比较两个同类型对象。

    下面用新定义的这个比较器对集合进行排序：

    还可以通过委托来进行集合排序，首先要定义一个供委托调用的方法，用于存放比较规则，可以用静态方法。请看下面的代码：然后通过内置的泛型委托System.Comparison<T>对集合进行排序：

    可以看到，后两种方式都可以对集合按照指定规则进行排序，但笔者更偏向于使用委托方式，可以考虑把各种比较规则放在一个类中，然后进行灵活调用。

//按照姓名对集合进行排序

persons.Sort(NameComparer.Default);

//输出所有人姓名

foreach (Person p in persons)

{

    Console.WriteLine(p.Name); //输出次序为"李四"、"王五"、"张三"

}class PersonComparison

{

    //按姓名比较

    public static int Name(Person p1, Person p2)

    {

        return System.Collections.Comparer.Default.Compare(p1.Name, p2.Name);

    }

}

    方法的参数为要进行比较的两个同类型对象，返回值为int类型，返回值处理规则与CompareTo方法相同。

System.Comparison<Person> NameComparison = new System.Comparison<Person>(PersonComparison.Name);

persons.Sort(NameComparison);

//输出所有人姓名

foreach (Person p in persons)

{

    Console.WriteLine(p.Name); //输出次序为"李四"、"王五"、"张三"

}

可以看到，后两种方式都可以对集合按照指定规则进行排序，但笔者更偏向于使用委托方式，可以考虑把各种比较规则放在一个类中，然后进行灵活调用。

    泛型集合的搜索

    搜索就是从集合中找出满足特定条件的项，可以定义多个搜索条件，并根据需要进行调用。首先，定义搜索条件，如下所示：

class PersonPredicate

{

    //找出中年人(40岁以上)

    public static bool MidAge(Person p)

    {

        if (p.Age >= 40)

            return true;

        else

            return false;

    }

}

    上面的搜索条件放在一个静态方法中，方法的返回类型为布尔型，集合中满足特定条件的项返回true，否则返回false。

System.Predicate<Person> MidAgePredicate = new System.Predicate<Person>(PersonPredicate.MidAge);

List<Person> MidAgePersons = persons.FindAll(MidAgePredicate);

//输出所有的中年人姓名

foreach (Person p in MidAgePersons)

{

    Console.WriteLine(p.Name); //输出"王五"

}然后通过内置的泛型委托System.Predicate<T>对集合进行搜索：

    泛型集合的扩展

    如果要得到集合中所有人的姓名，中间以逗号隔开，那该怎么处理？

    考虑到单个类可以提供的功能是有限的，很自然会想到对List<T>类进行扩展，泛型类也是类，因此可以通过继承来进行扩展。请看下面的代码：

//定义Persons集合类

class Persons : List<Person>

{

    //取得集合中所有人姓名

    public string GetAllNames()

    {

        if (this.Count == 0)

            return "";

        string val = "";

        foreach (Person p in this)

        {

            val += p.Name + ",";

        }

        return val.Substring(0, val.Length - 1);

    }

}

//创建并填充Persons集合

Persons PersonCol = new Persons();

PersonCol.Add(p1);

PersonCol.Add(p2);

PersonCol.Add(p3);

//输出所有人姓名

Console.Write(PersonCol.GetAllNames()); //输出“张三,李四,王五”

List的方法和属性 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

小结：

    本文着重于介绍运用C# 2.0中的泛型来实现集合，以及对集合功能进行扩展，恰当的运用泛型集合，可以减少很多重复工作，极大的提高开发效率。实际上，集合只不过是泛型的一个典型应用，如果想了解更多关于泛型的知识，可以查阅其他相关资料。希望本文对你有用:

**定义：**List<T>类表示可通过索引访问的对象的强类型列表，提供用于对列表进行搜索、排序和操作的方法。

**作用：**泛型最常见的用途是泛型集合  
我们在创建列表类时，列表项的数据类型可能是int,string或其它类型，如果对列表类的处理方法相同，  
就没有必要事先指定数据类型，留待列表类实例化时再指定。相当于把数据类型当成参数，这样可以最  
大限度地重用代码，保护类型的安全以及提高性能。

**List的一般用法**所属命名空间: System.Collections.Generic  
public class List<T>:IList<T>,Icollection<T>,IEnumerable<T>,IList,Icollection,Ienumerable  
List<T>是ArrayList类的泛型等效类，该类使用大小可按需动态增加的数组实现IList<T>泛型接口

**(1)声明 List<T>mlist = new List<T>();** eg: string[] Arr = {"a","b","c"};  
     List<string> mlist = new List<string>(Arr);

**(2)添加一个元素 List.Add(T item)**  
   eg: mlist.Add("d");

**(3)添加集合元素**   eg: string[] Arr2 ={"f","g"."h"};  
       mlist.AddRange(Arr2);

**(4)在index位置添加一个元素 Insert(int index,T item)**  
   eg: mlist.Insert(1,"p");

**(5)遍历List中元素**

　　foreach(T element in mlist) T的类型与mlist声明时一样  
　　　　　{  
　　　　　　 Console.WriteLine(element);  
          }

    eg:  
    foreach(string s in mlist)  
          {  
             Console.WriteLine(s);  
           }

**(6)删除元素**

    List.Remove(T item) 删除一个值  
    eg: mlist.Remove("a");

    List.RemoveAt(int index);删除下标为index的元素  
    eg: mlist.RemoveAt(0);  
      
    List.RemoveRange(int index,int count);　下标index开始，删除count个元素  
    eg：mlist.RemoveRange(3,2);

**(7)判断某个元素是否在该List中**

    List.Contains(T item)　返回true或false  
    eg:  
    if(mlist.Contains"("g"))  
       Console.WriteLine("g存在列表中");  
    else  
       mlist.Add("g");

**(8)给List里面元素排序 List.Sort()　默认是元素每一个字母按升序**  
   eg: mlist.Sort();

**(9)给List里面元素顺序反转 List.Reverse() 可以与List.Sort()配合使用**

**(10)List清空 List.Clear()**  
   eg: mlist.Clear();

(11)获得List中元素数目 List.Count()　返回int值  
   eg: mlist.count();

List进阶，强大方法

**(1)List.FindAll方法：检索与指定谓词所定义的条件相匹配的所有元素**   
      
    class program  
    {  
       static void Main(stirng[] args)  
       {  
         student stu = new student();  
         stu.Name="arron";  
         List<student> students= new List<student>();  
         students.Add(stu);  
         students.Add(new student("candy"));  
         FindName myname = new FindName("arron");  
         foreach(student s in students.FindAll(new Predicate<student>(myname.IsName)))  
         { Console.WriteLine(s);}  
       }

    public class student  
    {  
       public string Name{get;set;}  
       public student(){}  
       public override string ToString()  
        {  
            return string.Format("姓名:{0}",Name);  
         }  
     }

    public class FindName  
    {  
      private string \_name;  
      public FindName(string Name)  
      {  this.\_name=Name;}  
      public bool IsName(student s)  
       { return (s.Name==\_name)?true:false;}  
    }

**(2)List.Find方法　搜索与指定谓词所定义的条件相匹配的元素，并返回整个List中的第一个匹配元素**

　 eg:

    //Predicate是对方法的委托，如果传递给它的对象与委托定义的条件匹配，则该方法返回true,当前List的元素  
　　被逐个传递给Predicate委托，并在List中间前移动，从第一个元素开始，到最后一个元素结束，当找到匹配项  
　　时处理停止

**第一种方法 委托给拉姆达表达式：**　　eg:　　  
     string listFind = mlist.Find(name=>  
       {  
          if(name.length>3)  
             return true;  
          return false;  
       });

**第二种方法　委托给一个函数**    eg:  
     public bool ListFind(string name)

        {

            if (name.Length > 3)

            {

                return true;

            }

            return false;

        }

      这两种方法的结果是一样的

**(3) List.FindLast方法  public T FindLast(Predicate<T> match);确定是否 List 中的每个元素都与指定的谓词所定义的条件相匹配。用法与List.Find相同。**

**(4) List.TrueForAll方法：  确定是否 List 中的每个元素都与指定的谓词所定义的条件相匹配。**

 public bool TrueForAll(Predicate<T> match);

**(5) List.Take(n)：  获得前n行 返回值为IEnumetable<T>，T的类型与List<T>的类型一样**

E.g.：

IEnumerable<string> takeList=  mList.Take(5);

          foreach (string s in takeList)

          {

              Console.WriteLine("element in takeList: " + s);

          }

       这时takeList存放的元素就是mList中的前5个

**(6) List.Where方法：检索与指定谓词所定义的条件相匹配的所有元素。跟List.FindAll方法类似。**

E.g.：

            IEnumerable<string> whereList = mList.Where(name =>

                {

                    if (name.Length > 3)

                    {

                        return true;

                    }

                    else

                    {

                        return false;

                    }

                });

         foreach (string s in subList)

         {

             Console.WriteLine("element in subList: "+s);

         }

         这时subList存储的就是所有长度大于3的元素

**(7)List.RemoveAll方法：移除与指定的谓词所定义的条件相匹配的所有元素。**

public int RemoveAll(Predicate<T> match);

E.g.：

            mList.RemoveAll(name =>

                {

                    if (name.Length > 3)

                    {

                        return true;

                    }

                    else

                    {

                        return false;

                    }

                });

            foreach (string s in mList)

            {

                Console.WriteLine("element in mList:     " + s);

            }

      这时mList存储的就是移除长度大于3之后的元素。

　参考之 <http://www.cnblogs.com/ustc_msra_ase/articles/1890395.html>

版权声明：本文为博主原创文章，未经博主允许不得转载。