第3章 chapter 3

常用基础类与集合

用C#开发软件的一大优势就是能获得.NET Framework的各种支持，而.NET的类库就是其中重要的软件开发资源，它继承了大部分Windows API函数的功能，还提供了更高级别的操作，如数据访问XML串行化和字符串与集合的处理。离开了这些类库，就很难编写实用的C#应用程序，即使是简单的控制台程序也要依赖于.NET类库。

对于C#开发人员来说，熟悉常用的类库及其成员是十分重要的，能否熟练地掌握和使用类库是衡量程序员编程能力的一个很直观的标准。

本章主要内容包括：

- 常用基础类
- 集合和接口

3.1 常用基础类

3.1.1 .NET Framework基础类库

.NET的类库提供了各种类、接口、委托、结构和枚举，这些资源按照它们经常的应用领域分布在不同的命名空间中。我们曾经在1.5.3节中介绍过C#中一些常用的命名空间，下面再对这些命名空间做进一步详细介绍。

1. System，System.Collections和System.Text

System是.NET Framework的核心类库，包含了运行C#程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、异常处理类等。System.Collections是有关集合的基本类库，包括实现栈的Stack类和Hashtable类等。System.Text是有关文字字符的基本类库。

2. System.IO

System.IO是输入、输出的基础类库，包含了实现C#程序与操作系统、用户界面及其他C#程序做数据交换所使用的类，如基本输入、输出流，文件输入、输出流，二进制输入、输出流，字符读写类流等。

3. System. Windows. Forms 和 System. Drawing

System. Windows. Forms 是用来构建 Windows 窗体的类库，而 System. Drawing 提供了基本的图形操作。这两个名字空间为图形用户界面提供了多方面的支持：低级绘图操作，比如 Graphics 类等；图形界面组件和布局管理，如 Form，Button 类等；以及用户界面交互控制和事件响应，如 MouseEventArgs 类。利用这些功能，可以很方便地编写出标准化的应用程序界面。

4. System. Web

System. Web 是用来实现运行与 Internet 相关开发的类库，它们组成了 ASP. NET 网络应用开发的基础类库。

5. System. Xml 和 System. Web. Services

System. Xml 是处理 Xml 的类库，而 System. Web. Services 是处理基于 Xml 的 Web 服务的类库。Xml 和 Web 服务是现代程序设计的一种趋势。

6. System. Data

System. Data 是关于数据及数据库程序设计的.. NET Framework 中处理数据库的技术被称为 ADO. NET。

7. System. Net 和 System. Net. Socket

System. Net 和 System. Net. Socket 是关于底层网络通信的。在此基础上，可以开发具有网络功能的程序，如 Telnet、FTP 邮件服务等。

8. 其他

C#语言中还有其他许多名字空间及类库。如 System. Threading 是关于多线程的等。

在. NET 3. 0/3. 5 中，还增加一些新的基础类库，比如，关于 LINQ 数据访问技术的 System. Linq，有关工作流开发的 System. Workflow 等。

由于. NET Framework 涉及的类库十分庞大，所以本书中将介绍其中最重要的概念和类库及 C#程序设计中最常用的技术。在实际程序设计过程中，要经常参考. NET Framework SDK 的文档。如果使用 Visual Studio IDE，还可以使用其中的帮助功能来查阅这些文档。在文档中，有相关的名字空间、类、属性、方法等的说明，有的还有简单的示例。

3.1.2 Math 类

Math 类提供了若干实现不同标准数学函数的方法。这些方法都是静态的方法（关于静态方法请参见 4. 5. 3 小节），所以在使用时无须创建 Math 类对象，而直接用类名做

前缀即可调用这些方法。表 3-1 列出了 Math 类常用的数学函数。

表 3-1 Math 类的数学函数

函 数 名	说 明
Abs()	返回数的绝对值
Sin(),Cos(),Tan()	标准三角函数
ASin(),ACos(),ATan(),ATan2()	标准反三角函数
Sinh(),Cosh(),Tanh()	标准双曲函数
Max(),Min()	最大值,最小值
Celling()	返回不小于指定数的最小整数
Floor()	返回不大于指定数的最大整数
Round()	返回指定数的四舍五入值
Truncate()	返回数字整数部分
Log(),Log10()	自然对数或以 10 为底的对数
Exp()	指数函数
Pow()	返回指定数的乘方
Sign()	返回指定数的符号值,负数为－1,零为 0,正数为 1
Sqrt()	返回指定数的平方根
IEEERemainder()	返回两数相除的余数,如 Math. IEEERemainder(13. 5, 3),结果为 1. 5

【实例 3-1】 Math 类用法实例。代码如表 3-2 所示。

表 3-2 实例 3-1 源代码

行号	源 代 码
01	using System;
02	namespace 实例 3_1Math 类用法{
03	class Program{
04	static void Main(string[] args) {
05	Console. WriteLine("－12 的绝对值为:{0}", Math. Abs(－12));
06	Console. WriteLine("不小于－12. 567 的最小整数为:{0}",
07	Math. Ceiling(－12. 567));
08	Console. WriteLine("不大于－12. 567 的最大整数为:{0}",
09	Math. Floor(－12. 567));
10	Console. WriteLine("－12. 567 保留为小数的四舍五入值为:{0}",
11	Math. Round(－12. 567, 2));
12	Console. WriteLine("2 的指数函数为:{0}", Math. Exp(2));
13	Console. WriteLine("2 的次方为:{0}", Math. Pow(2, 3));
14	Console. WriteLine("13. 5/3 余数为:{0}", Math. IEEERemainder(13. 5, 3));
15	}
16	}
17	}

3.1.3 DateTime 和 TimeSpan 类

1. DateTime 类

使用 System 命名空间中定义的 DateTime 类可以完成日期与时间数据的处理工作。在一个日期时间变量中，可以使用 Year、Month、Day、Hour、Minute 和 Second 属性分别获取年、月、日、时、分、秒的数据信息。

2. TimeSpan 类

TimeSpan 类表示一个时间间隔。范围在 Int64. MinValue 至 Int64. MaxValue 之间。

【实例 3-2】 DateTime 类和 TimeSpan 类用法实例。源代码如表 3-3 所示。

表 3-3 实例 3-2 源代码

行号	源 代 码
01	using System;
02	namespace 实例 3_2{
03	class Program {
04	static void Main(string[] args) {
05	//使用 DateTime 类创建一个 DateTime 对象 dt，并赋值 2015-9-8
06	DateTime dt=new DateTime(2015,9, 8);
07	//将对象 dt 以短日期格式显示出来
08	Console. WriteLine(dt. ToShortDateString());
09	Console. WriteLine("2015 年 9 月 8 日是本年度的第{0}天",dt. DayOfYear);
10	//输出对象 dt 的月份值
11	Console. WriteLine("月份：{0}", dt. Month. ToString());
12	//使用 TimeSpan 类创建一个 TimeSpan 对象 ts，并赋值
13	TimeSpan ts=dt-DateTime. Now;//DateTime. Now 表示当前日期
14	Console. WriteLine("距离 2015 年国庆还有{0}天", ts. Days. ToString());
15	}
16	}
17	}

无论使用 DateTime 类所创建的对象 dt，还是使用 TimeSpan 类所创建的对象 ts 都具有许多属性与方法。例如，下面几个很实用的方法：

IsLeapYear()方法，判断一个年份是否为闰年，如“DateTime. IsLeapYear(2016);”语句返回 true。

DaysInMonth()方法，返回指定年份中某个月份的天数，如“DateTime. DaysInMonth(2015,2);”语句返回 28。

3.1.4 Random 类

Random 类用来产生随机数。Random 类的 Next()方法可产生一个 int 型随机数；

Next(int maxValue)方法可产生一个小于所指定最大值的非负随机整数;NextDouble()方法可产生一个介于 0 和 1.0 之间随机数。

例如,下面一段代码能够使用 Random 类产生 10 个[0,100]之间的随机整数。

```
Random rd=new Random();
for(int i=0; i<10; i++)
{
    Console.Write("{0},",rd.Next(100));
}
```

3.1.5 String 类

字符串(String)是引用类型的一种,表示一个 Unicode 字符序列。一个字符串可存储约 231 个 Unicode 字符。

1. 字符串建立

字符串常量是用一对半角双引号("")表示。例如语句,

```
string str="Hello world!";
```

在字符串中如果包含了"\"字符,有以下两种处理方法:

第一种方法是采用转义字符。如:

```
string str="c:\\windows \\OLP.DLL";
```

第二种方法是在字符串前面加上字符@,第二种方法中的@字符表示该字符串的所有字符是其原来的含义,而不解释为转义字符。如:

```
string str=@"c:\windows\OLP.DLL";
```

2. 字符串的表示格式

使用 string.Format()方法或 Console.WriteLine()方法均可以将字符串表示为规定格式。但这两种方法式完全不同的:string.Format()方法返回一个字符串,而 Console.WriteLine()方法自动调用 string.Format()并将格式化后的字符串显示出来。这两种方法都要用到格式参数,格式参数的一般形式为:

```
{N[,M][:formatcode]}
```

其中,N 是以 0 为起始编号的、将被替换的参数号码。M 是一个可选整数,表示最小宽度值。若 M 为负数,则左对齐;若 M 为正,则右对齐;若 M 大于实际参数的长度,则用空格填充。Formatcode 也是一个可选参数,其含义参见表 3-4。

此外,如果标准格式选项不能满足要求,则需要使用形象描述格式。形象描述格式采用多个形象描述字符来表示输出格式。表 3-5 给出了一些常用的形象描述字符。

表 3-4 标准格式选项

格式符	含　义	示例(int i=19; double x=19.7;)	结　果
C	按金额形式输出	Console.WriteLine("{0,8:C}",i);	￥19.00
D	按整数输出	Console.WriteLine("{0,8:D}",i);	19
E	科学计数格式	Console.WriteLine("{0:E}",i);	1.900000E+001
F	小数点后位数固定	Console.WriteLine("{0,8:F3}",x);	19.700
G	使用 E 和 F 中合适的一种		
N	输出带有千位分隔符的数字	Console.WriteLine("{0,8:N3}", 19890);	19,890.000
P	百分数格式	Console.WriteLine("{0,5:P0}",0.78);	78%

表 3-5 常用形象描述字符

形象描述字符	含　义	示例(double x=456.78)	结　果
0	数字或 0 占位符	Console.WriteLine("{0:0000.000}", x);	0456.7800
#	数字占位符	Console.WriteLine("{0:####.000}", x);	456.7800
.	小数点		
,	数字分隔符	Console.WriteLine("{0:#,###.000}", 3456.78);	3,456.780
%	百分号	Console.WriteLine("{0:0.00%}",0.78);	78.00%

【实例 3-3】 字符串输出格式。源代码如表 3-6 所示。

表 3-6 实例 3-3 源代码

行号	源　代　码
01	using System;
02	namespace 实例 3_3 字符串输出格式{
03	class Program {
04	static void Main(string[] args) {
05	double x=3456.78;
06	string s0=string.Format("{0,10:F3}", x);
07	string s1=string.Format("{0:#######.0000}", x);
08	Console.WriteLine(s0);
09	Console.WriteLine(s1);
10	Console.WriteLine("{0,10:f3},{1,10:E}", x,x);
11	Console.ReadLine();
12	}
13	}
14	}

输出结果为：

```
3456.780
3456.7800
```

```
3456.780,3.456780E+003
请按任意键继续…
```

3. 常用的字符串操作方法

表 3-7 列举了字符串常用的几个方法。有的方法有多种重载(关于重载的概念请参见 4.5.4 节)。

表 3-7 常用的字符串操作方法

方法名称	方法格式	功能说明
比较两个字符串	string. Compare(string strA, string strB)	如果 strA 大于 strB,结果为 1 如果 strA 小于 strB,结果为-1 如果 strA 等于 strB,结果为 0
	string. Compare(string strA, string strB,bool ignoreCase)	比较两个字符串时是否忽略大小写,true 表示忽略,false 表示区分大小写
	string. Equals(string strA, string strB)	两串相等返回 true,否则返回 false
字符串是否为空	string. IsNullOrEmpty(string str)	判断 str 是否为空,返回 bool 值
查找	strS. IndexOf(char value)	返回字符 value 在字符串 strS 中首次出现的位置。注意,起始位置从 0 开始。返回结果为整数
	stS. IndexOf(string value)	返回字符串 value 在字符串 strS 中首次出现的位置。返回结果为整数
	strS. IndexOf(char value, int startIndex)	在字符串 strS 中从第 startIndex 个字符开始查找字符 value 首次出现的位置
	strS. LastIndexOf(string str)	返回字符串 str 在 strS 中最后一次出现的位置
插入	strS. Insert(int startIndex, string str)	在字符串 strS 的第 startIndex 位置插入字符串 str
删除	strS. Remove(int startIndex, int count)	在 strS 中删除从 startIndex 开始的 count 个字符串
替换	strS. Replace(string oldStr, string newStr)	将字符串 strS 中所有 oldStr 替换为 newStr
分离	stS. Split(char [] separator)	将字符串 strS 按照指定的字符进行分离,返回 string 型数组
	strS. ToCharArray()	将字符串 strS 分离成字符,返回 char 型数组
取子串	strS. Substring(int startIndex, int length)	从字符串 strS 的 startIndex 开始取 length 个字符

续表

方法名称	方法格式	功能说明
大小写转换	strS. ToUpper()	将字符串 strS 全部转换为大写
	strS. ToLower()	将字符串 strS 全部转换为小写
去掉空格	strS. TrimStart()	删除字符串 strS 左端的空格
	strS. TrimEnd()	删除字符串 strS 右端的空格
	strS. Trim ()	删除字符串 strS 左右两端的空格
字符串是否含有数字	Char. IsNumber(string strS,int index)	判断字符串 strS 第 index 位置的字符是否是数字,是,返回 true 值,否,返回 false

说明:在表 3-7 中,通过类名 string 调用的方法是静态方法,通过字符串实例 strS 调用的方法是实例方法,在实际应用时要正确调用。

【实例 3-4】 假设有一字符串 strS="This Is An Apple.",使用字符串方法,完成下面的要求。源代码如表 3-8 所示。

(1) 取出字符串 strS 中的第 9 个字符,然后统计该字符在字符串 strS 中出现的次数。

(2) 统计字符串中单词的个数。

(3) 将字符串反序并全部转换为大写字符输出。

表 3-8 实例 3-4 源代码

行号	源代码
01	using System;
02	namespace 实例 3_4{
03	class Program {
04	static void Main(string[] args) {
05	string strS="This Is An Apple.";
06	if(!string. IsNullOrEmpty(strS)
07	{
08	char findChar=Convert. ToChar (strS. Substring(8,1));//取出字符串中的第 9 个字符
09	int count=GetCharCount(strS, findChar); //统计指定的字符出现的次数
10	string[] word=strS. Split(' '); //将字符串 strS 按照空格分隔成数组
11	int len=word. Length; //统计单词的个数
12	string strSReverse=MyReverse(strS). ToUpper (); //将字符串反序,并转换为大写
13	Console. WriteLine ("\"This Is An Apple.\"共有{0}个单词,{1}出现了{2}次",
14	len,findChar ,count);
15	Console. WriteLine(strSReverse);
16	}
17	else
18	Console. WriteLine("字符串为空");
19	Console. ReadKey();
20	}

续表

行号	源 代 码
21	public static int GetCharCount(string strS, char findChar) {
22	int count=0;
23	char[] c=strS. ToCharArray(); //将字符串分离成字符数组
24	for(int i=c. Length-1; i>=0; i--) {
25	if (c[i]==findChar)
26	count++;
27	}
28	return count;
29	}
30	public static string MyReverse(string strS) {
31	string strReverse="";
32	char[] c=strS. ToCharArray(); //将字符串分离成字符数组
33	for(int i=c. Length-1; i>=0; i--) {
34	strReverse+=c[i]. ToString();
35	}
36	return strReverse;
37	}
38	}
39	}

3.1.6 StringBuilder 类

前面介绍过,String 类的索引函数是只读的,其各种操作方法不是修改字符串本身,而是生成新的字符串。由于 String 的值一旦建立就不能修改,修改 String 的值实际上是返回一个包含新内容的新的 String 实例。显然,如果这种操作非常多,对内存的消耗是很大的。

例如,下面的代码并不能改变字符串 str 的内容。

```
string str="C#";
str+="实例教程";
str=str.Substring(4, 2);
```

准确地说,一旦创建了一个 String 对象,其内容就是不可变的,每次操作都是生成一个新字符串,而后将当前对象的引用指向新字符串,如图 3-1 所示。

图 3-1 字符串对象操作示意图

此过程中一共生成了 4 个字符串对象(包括常量对象"实例教程"),前 3 个字符串将脱离程序的控制范围(没有任何对象指向它们),等待 CLR 进行回收。对于很长的或是需要频繁操作的字符串,这样往往会消耗大量的系统资源。

.NET 类库专门提供了一个 StringBuilder 类(位于 System.Text 命名空间下),它对字符串进行动态管理,即允许直接修改字符串本身的内容,而不是每次操作都生成新字符串。使用 StringBuilder 类每次重新生成新字符串时不再生成一个新实例,而是在原来字符串占用的内存空间上处理,而且它可以动态地分配占用的内存空间大小。因此,在字符串处理操作比较多的情况下,使用 StringBuilder 类可以显著提高系统性能。

StringBuilder 与 String 类的用法有很多相似之处,包括通过 Length 属性来获取长度,通过索引函数(在 StringBuilder 中是可读写的)来访问字符,以及 Insert、Remove、Replace 这些子串操作方法。尽管这些方法的返回类型也为 StringBuilder,但并没有创建新的对象,返回值也就是调用这些方法的对象本身。通过 StringBuilder 的 ToString 方法就可以获得其中的字符串。

StringBuilder 类还提供了 Capacity 和 MaxCapacity 属性,分别表示字符串的初始容量和最大容量。应尽量为 StringBuilder 对象指定合适的初始容量,如果过大就会占用不必要的内存空间,过小则会导致频繁的重新调整。如果某些操作使字符串超出了初始空间,那么 StringBuilder 对象会自动增加内存空间。该类特有的 3 个方法是 Append、AppendLine 和 AppendFormat,它们都用于在字符串的尾端追加新内容。表 3-9 列出了 StringBuilder 特有的属性和方法。

表 3-9 StringBuilder 类特有的属性和方法

属　性	描　述
Capacity	StringBuilder 实例的初始容量,可读可写
MaxCapacity	StringBuilder 实例的最大容量,只读
方　法	**描　述**
Append	向 StringBuilde 实例的尾端追加字符串
AppendLine	向 StringBuilde 实例的尾端追加一行字符串
AppendFormat	按照指定的格式向 StringBuilde 实例的尾端追加字符串

例如在实例 3-4 中,字符串反序的方法 MyReverse 总共创建了 2 倍的 c.Length 个字符串实例。使用 StringBuilder 只在一个实例上进行操作,方法代码修改如下所示。

```
public static string MyReverse(string strS) {
    StringBuilder strReverse=new StringBuilder();
    char[] c=strS.ToCharArray();          //将字符串分离成字符数组
    for(int i=c.Length-1; i>=0; i--)
    {
```

```
            strReverse.Append(c[i]);
        }
        return strReverse.ToString();
    }
```

3.1.7 Array 类

在.NET Framework 环境中,我们并不能直接创建 Array 类型的变量,但是所有的数组都可以隐式地转换为 Array 类型。这样一来,就可以在数组中使用 Array 类中定义的一系列属性和方法了。下面重点介绍这些属性、方法中经常用到的几个。

1. Rank 属性、Length 属性

Rank 属性用于获取数组的维数(又称为秩)。Length 属性用于获取数组所有维数中元素的总和。Rank 属性和 Length 属性只能用于数组对象。

2. GetLength()、GetLowerBound()、GetUpperBound ()方法

GetLength(dimension) 方法用于获取指定维中元素的个数。GetLowerBound (dimension)和 GetUpperBound (dimension)方法分别用于获取指定维的下界和上界。这三个方法也只能用于数组对象。

3. Sort() 方法

Sort(array)方法用于对指定数组升序排序。

```
int[] nums={2, 7, 5, 3, 6};
Array.Sort(nums);
```

执行上面语句之后,数组 nums 中各数组元素已经按升序顺序进行了排列,各数组元素依次为:2,3,5,6,7。

4. Reverse ()方法

Reverse (array)用于对数组元素进行逆序,即首尾倒置。

```
int[] nums={2, 7, 5, 3, 6};
Array.Reverse(nums);
```

执行上面语句之后,数组 nums 中各数组元素依次为:6,3,5,7,2。

5. Copy()方法

Array.Copy(nums, destArray, destArray.Length)用于将源数组中的元素复制到目标数组中。该方法中可以省略第三个参数,表示所有元素。

```
int[] nums={2, 7, 5, 3, 6};
```

```
int [] arrb=new int [nums.Length];
Array.Copy(nums,arrb)
```

上面语句实现了将数组 nums 中各元素复制到数组 arrb 中。

6. IndexOf()方法

IndexOf(array,value,startindex)方法返回指定数组中、从指定位置开始、与 value 匹配的元素的位置,返回值类型为整型。该方法中可以省略第三个参数,表示从头开始查找。

```
int[] nums={2, 7, 5, 3, 6};
int n=Array.IndexOf(nums,5,0);
```

执行上面的语句之后,n 的值为 2。

7. BinarySearch 方法

BinarySearch (array,value)方法对已排序的数组使用二分查找法进行搜索。返回值类型为整型,表示值为 value 的元素在已排序数组中的位置。如果未找到返回一个负值。

```
int[] nums={2, 7, 5, 3, 6};
Array.Sort(nums);                    //排序,排序后 nums 中各元素: 2,3,5,6,7
int n=Array.BinarySearch(nums, 6);  //二分查找
```

执行上面的语句, n 的值为 3,表示值为 6 的元素在第 4 个位置。

8. Clear ()方法

Clear (array,index,length)方法将数组中的一系列元素置零、false 或 null,具体取决于元素类型。

```
int[] nums={2, 7, 5, 3, 6};
Array.Clear(nums,2,3);
```

执行上面的语句, 数组 nums 中数组元素依次为: 2,7,0,0,0。

3.1.8 并行计算

在.NET 4.0 之前开发并行的程序非常困难。在.NET 4.0 中,C # 通过引入 Parallel 类,提供了对并行开发的支持。Parallel 类提供了 Parallel.For、Parallel.ForEach、Parallel.Invoke 等方法,其中 Parallel.Invoke 用于并行调用多个任务。下面通过 Parallel.For 应用实例展示 Parallel 类的基本用法。

【实例 3-5】 使用 Parallel 类进行并行计算。创建该实例步骤:

启动 VS 2010,依次选择“文件”→“新建”→“项目”→“控制台应用”命令,打开 Program.cs 文件,用下面代码替换原先代码,按 F5 键运行。实例源代码如表 3-10 所示。

表 3-10 实例 3-5 源代码

行号	源 代 码
01	using System;
02	using System. Threading;
03	using System. Threading. Tasks;
04	namespace ytu{
05	class myParallel{
06	static void Main(string[] args){
07	Normal();
08	ParallelFor();
09	Console. Read();
10	}
11	public static void Normal(){
12	DateTime dt=DateTime. Now;
13	for(int i=0; i<10; i++){
14	for(var j=0; j<10; j++) DoSomething();
15	}
16	Console. WriteLine("Normal 方法耗时{0}",
17	(DateTime. Now-dt). TotalMilliseconds. ToString());
18	}
19	public static void ParallelFor(){
20	DateTime dt=DateTime. Now;
21	Parallel. For(0, 10, i=>
22	{
23	for(int j=0; j<10; j++) DoSomething();
24	});
25	Console. WriteLine("ParallelFor 方法耗时{0}",
26	(DateTime. Now-dt). TotalMilliseconds. ToString());
27	}
28	public static void DoSomething(){
29	Thread. Sleep(100);
30	}
31	}
32	}

运行结果如图 3-2 所示。

图 3-2 实例 3-5 并行计算运行结果

3.2 集　　合

如果将紧密相关的数据组合到一个集合中，则能够更有效地处理这些紧密相关的数据。这是由于集合类(Collections)具有自动内存管理功能、支持枚举访问、某些集合类还具有排序和索引功能等。合理地使用集合可以简化代码数量，提高代码效率。

3.2.1 什么是集合

集合是一组组合在一起的类似的类型化对象。在.NET 中提供了专门用于存储大量元素的集合类(Collections)。这些集合通常可以分为表 3-11 中的三种类型。

表 3-11　Collections 类的类型

类　型	描　　述
常用集合	这些集合是数据集合的常见变体，如动态数组(ArrayList)、哈希表(HashTable)、队列(Queue)、堆栈(Stack)、SortedList 等。常用集合有泛型和非泛型之分
位集合	这些集合中的元素均为位标志，它们的行为与其他集合稍有不同
专用集合	这些集合具有专门的用途，通常用于处理特定的元素类型，如 StringDictionary

C# 2.0 引入了泛型的概念之后(关于泛型概念将在 4.11.1 小节中介绍)，表 3-11 中的常用集合就有了泛型集合与非泛型集合之分。.NET Framework 2.0 版类库提供一个名为 System.Collections.Generic 的命名空间，其中包含了基于泛型的集合类。非泛型集合与泛型集合主要区别如下：

- 所有非泛型集合类都有一个共同的特征(除 BitArray 以外，它存储布尔值)，那就是弱类型。换句话说，它们存储 System.Object 的实例。弱类型使集合能够存储任何类型的数据，因为所有数据类型都是直接或间接地从 System.Object 派生得来。但是，弱类型也意味着使用者需要对集合中的元素执行附加的处理，例如装箱、拆箱或转换，这些操作会影响集合的性能。
- 与所有非泛型集合不同，泛型集合同时具备可重用性、类型安全和效率，这是非泛型集合无法具备的。

由于泛型集合与非泛型集合存在这种区别，C# 2.0 版之后的应用程序在使用集合类时几乎都采用泛型集合类。掌握泛型集合类的使用是大家学习重点。因此，本节重点介绍动态数组(ArrayList)、哈希表(HashTable)、队列(Queue)、堆栈(Stack)、SortedList 五个常用的非泛型集合，以及 Icollection、Ienumerable 和 Ilis 三个常用的接口。

3.2.2 ArrayList

我们知道，数组在用 new 创建后，其大小(Length)是不能改变的，而 ArrayList 中的数组元素的个数(Count)是可以改变的，元素可以随意添加、插入或移除，ArrayList 实际

上是C#中的“动态数组”。

在ArrayList类型的数据中，成员都为object类型，这样就可以存放任意类型的数据了。定义ArrayList类型变量时，可以使用如下格式：

```
ArrayList  <数组名称>=new  Arraylist();
```

在定义ArrayList类型变量后，就可以使用一些方法和属性来操作数组。表3-12列出了ArrayList常用的属性与方法。

表3-12 ArrayList属性与方法

属 性	描 述	属 性	描 述
Capacity	ArrayList的容量，容量是指ArrayList中可包含的元素数	Count	ArrayList的元素个数，指ArrayList中实际包含的元素数
方 法	**描 述**	**方 法**	**描 述**
Add	向数组增加一个元素	AddRange	向数组增加一定范围内的元素
Clear	清除所有元素	Contains	判断某个元素是否在数组中
Insert	使用索引插入某个元素	Remove	删除某个元素
RemoveAt	使用索引来指定要删除条目的位置	ToArray	将ArrayList元素复制到指定数组中
IndexOf	查找某个元素的索引		

下面通过一个实例说明ArrayList类使用。

【实例3-6】 ArrayList类用法实例。源代码如表3-13所示。

表3-13 实例3-6源代码

行号	源 代 码
01	using System;
02	using System.Collections;
03	namespace 实例3_6{
04	class Program{
05	static void Main(string[] args){
06	ArrayList myAL=new ArrayList();
07	myAL.Capacity=6; //当容量超过6时，其容量自动增加一倍
08	for(int i=0; i<10; i++) {
09	myAL.Add(i);
10	}
11	myAL.RemoveAt(2); //删除索引为2的数据，即第三个数据
12	myAL.Reverse();
13	foreach (int item in myAL)
14	Console.Write("{0},",item.ToString ());
15	Console.WriteLine("元素个数:{0}", myAL.Count);
16	Console.WriteLine("动态数组|容量:{0}", myAL.Capacity);
17	Console.ReadKey();
18	}
19	}
20	}

针对上面实例运行结果，对 ArrayList 类解释如下：

- 注意区别 ArrayList 的容量(Capacity)和元素个数(Count)两个属性。容量是指 ArrayList 能装多少个元素，元素个数是指此时到底装了多少个元素。
- 对于 ArrayList 类，当元素个数超过容量时，其容量会自动增加一倍。例如，在上面的例子中，设置 ArrayList 的容量为 2，然后往里面添加 3 个元素。此时元素个数超出其容量，所以容量自动增长为 4，容量仍然不足，再次自动增长为 8。

运行结果如图 3-3 所示。

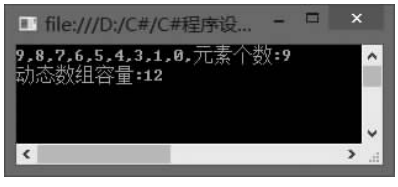

图 3-3　实例 3-6 运行结果

ArrayList 尽管扩充了数组的功能，但是同数组相比，ArrayList 也有缺点：ArrayList 只能是一维的，而数组可以是多维的；ArrayList 下标必须从零开始，而数组下标可以不从零开始；另外数组执行效率也高于 ArrayList。

3.2.3　Hashtable

1. Hashtable 概述

Hashtable 通常被称为哈希表。在 Hashtable 类型的变量中的每一个元素都以键(key)/值(value)对的格式保存。Hashtable 中键(Key)唯一，不能有重复值，不能为空值，但值(Value)可以为空。简单地说，Hashtable 像一个字典，根据键可以查找到相应的值。

Hashtable 中的键/值对均为 object 类型，所以 Hashtable 可以支持任何类型的键/值对。当然，这一特性也将影响 Hashtable 性能。为此，在 C＃ 2.0 之后，建议采用 Hashtable 的泛型版本：Dictionary＜TKey，TValue＞。

Hashtable 常用的属性和方法如表 3-14 所示。

表 3-14　Hashtable 常用的属性与方法

属　性	描　述	属　性	描　述
Count	表示哈希表中元素的个数	Keys	表示哈希表中所有键的集合
Values	表示哈希表中所有列的集合		
方　法	**描　述**	**方　法**	**描　述**
Add	向哈希表末尾增加一个元素	Clear	清除哈希表中所有元素
Contains	判断哈希表中是否包含该键	ContainsValue	判断哈希表中是否包含该值
Remove	删除哈希表中一个元素		

以下代码演示了 Hashtable 属性和方法的最基本用法：

```
Hashtable hst=new Hashtable();                    //声明 Hashtable 对象
hst.Add("郭晓冬", 235);                           //添加元素(键和值)
if (!hst.Contains ("赵刚")) {
```

```
    hst.Add("赵刚", 143);
}
hst["郭晓冬"]=200;                          //修改元素
hst["赵刚"]=(int)hst["赵刚"]+10;
hst.Remove("赵刚");                         //移除元素
foreach (DictionaryEntry item in hst)       //输出元素
    Console.WriteLine("{0},{1}", item.Key, item.Value);
```

C#中提供了foreach语句以对Hashtable进行遍历。由于Hashtable的元素是一个键/值对,因此需要使用DictionaryEntry类型来进行遍历。DictionaryEntry类型在此处表示一个键/值对的集合。

2. Hashtable实例

Hashtable是以一种键/值对的形式存在的,因此要通过键来访问Hashtable中的值,即Hashtable[key]。以下代码演示了Hashtable最基本的用法:

【实例3-7】 车辆进出闸口自动刷卡扣费,Hashtable哈希表用于检查是否重复读卡的问题。

问题的提出:由于进出闸口读卡器距离很近,存在刷一次卡,被入口读卡器、出口读卡器同时读到的问题。解决办法是:由于哈希表的key为卡的ID+IP、value为时间。这样,只要能保证目标读卡器首先读到信号,其他读卡器再读到信号也会被当作重复读卡处理。

部分源代码如表3-15所示。

表3-15 实例3-7部分源代码

行号	部分源代码
01	class Program{
02	static Hashtable hsTable1=new Hashtable();
03	public static bool Repeat(string key, int interval) {
04	try {
05	if (!hsTable1.Contains(key))　　//如果key未保存在哈希表中
06	{
07	hsTable1.Add(key, DateTime.Now);
08	return false;
09	}
10	//如果key已存在哈希表中,则两种情况:超过时限间隔的(可能是返回车辆)、
11	重复读卡
12	//其中,重复读卡又可能是:同一读卡器、不同读卡器
13	TimeSpan ts=DateTime.Now-Convert.ToDateTime(hsTable1[key]);
14	if (ts.TotalSeconds>interval)
15	{
16	hsTable1[key]=DateTime.Now;
17	return false;
18	}

续表

行号	部分源代码
19	//属于重复读卡
20	return true;
21	}
22	catch (Exception ex) { throw ex; }
23	}
24	static void Main(string[] args) {
25	Console. WriteLine(Repeat("KEY1", 3)); //interval 设置为 3 秒
26	System. Threading. Thread. Sleep(4000); //使用线程延迟 4 秒
27	Console. WriteLine(Repeat("KEY1", 3));
28	}
29	}

3. Hashtable 优点

Hashtable 的基本原理是通过节点的关键码确定节点的存储位置，即给定节点的关键码 k，通过一定的函数关系 H(散列函数)，得到函数值 H (k)，将此值解释为该节点的存储地址。因此，Hashtable 的优点主要在于其索引的方式：不是通过简单的索引号，而是采用一个键(key)。这样可以方便地查找 Hashtable 中的元素，而且查找速度非常快，在对速度要求比较高的场合可以考虑使用 Hashtable。

3.2.4 Queue 和 Stack

1. Queue 和 Stack 概念

Queue(队列)和 Stack(栈)是两种重要的线性数据结构。队列遵循“先进先出”(First In First Out，FIFO)的原则，而栈则遵循一种“后进先出”(Last In First Out，LIFO)的原则。

队列的特性就是固定在一端输入数据(称为加队，Enqueue)，另一端输出数据(称为减队，Dequeue)。队列中数据的插入必须在对头进行，删除数据必须在队尾进行，而不能直接在任何位置插入和删除数据。

栈只能在一端输入输出，它有一个固定的栈底和一个浮动的栈顶。栈顶可以理解为是一个永远指向栈最上面元素的指针。向栈中输入数据的操作称为“压栈”，被压入的数据保存在栈顶，并同时使栈顶指针上浮一格。从栈中输出数据的操作称为“弹栈”，被弹出的总是栈顶指针指向的位于栈顶的元素。如果栈顶指针指向了栈底，则说明当前的栈是空的。

当需要临时存储信息时(也就是说，可能想在检索了元素的值后放弃该元素)，栈和队列都很有用。如果需要按照信息存储在集合中的顺序来访问这些信息，则应使用 Queue。如果需要以相反的顺序访问这些信息，则应使用 Stack。

2. Queue 操作

对 Queue 及其元素执行的操作主要有如下三种。

- Enqueue()：将一个元素添加到 Queue 的末尾。
- Dequeue()：从 Queue 的开始处移除最旧的元素。
- Peek()：从 Queue 的开始处返回最旧的元素，但不将其从 Queue 中移除。

【实例 3-8】 Queue 的操作。源代码如表 3-16 所示。输出结果如图 3-4 所示。

表 3-16 实例 3-8 源代码

行号	源 代 码
01	using System;
02	using System.Collections;
03	namespace 3_8{
04	class Program {
05	static void Main(string[] args){
06	string[] months={"January", "February", "March", "April", "May"};
07	Queue queue=new Queue();
08	//使用 Enqueue()方法将一个元素添加到 Queue 的末尾
09	foreach (string item in months)
10	queue.Enqueue(item);
11	Console.WriteLine("队列中元素个数是:{0}",queue.Count);
12	//使用 Dequeue()从 Queue 的开始处移除最旧的元素
13	while(queue.Count>0)
14	Console.WriteLine(queue.Dequeue());
15	}
16	}
17	}

3. Stack 操作

对 Stack 及其元素执行的操作主要有如下三种。

- Push()：将指定对象压入栈中。
- Pop()：将最上面的元素从栈中取出，并返回这个对象。
- Peek()：返回栈顶元素，但不将此对象弹出。

【实例 3-9】 Stack 的操作。源代码如表 3-17 所示。运行结果如图 3-5 所示。

表 3-17 实例 3-9 源代码

行号	源 代 码
01	using System;
02	using System.Collections;
03	namespace Stack_Sample{
04	class Program {
05	static void Main(string[] args){

续表

行号	源 代 码
06	string[] months={"January", "February", "March", "April", "May"};
07	Stack stack=new Stack();
08	foreach (string item in months)
09	stack. Push(item); //入栈
10	while(stack. Count>0)
11	Console. WriteLine(stack. Pop()); //出栈
12	}
13	}
14	}

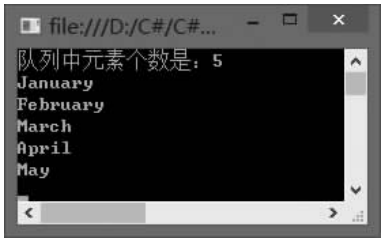

图 3-4 实例 3-8 运行结果

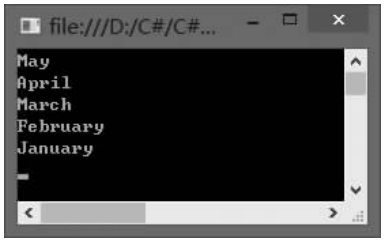

图 3-5 实例 3-9 运行结果

通过上面的实例可以看出：

(1) Stack 与 Queue 在用法上非常类似，但从二者的运行结果不难看出，Stack 与 Queue 对数据处理方式是截然不同的。

(2) 连续输出 Queue 和 Stack 中的数据时，使用 while(queue. Count>0)和 while (stack. Count>0)，不能使用 for 循环，因为 Count 属性的值在不断变化减少。

3.2.5 SortedList 类

SortedList 提供了类似于 ArrayList 和 Hashtable 的特性，可以将其理解为一种结合体。SortedList 的元素是键/值对，这点与 Hashtable 相似；而其提供了索引的方法，这点又与 ArrayList 类似。

1. SortedList 的概念

SortedList 表示键/值对的集合。使用两个数组存储其数据，一个保存关键字，一个用于存放值。因此，列表中的一个表项由“关键字/值”对组成。SortedList 不允许出现重复的关键字，并且关键字也不允许是 null 引用。

SortedList 以关键字顺序维持数据项，并且能够根据关键字或者索引来检索它们。SortedList 的常用属性和方法与前面介绍的 Hashtable(请参见 3. 2. 3 小节)的属性和方法完全一样，在此不再赘述。