

第 5 章

数 组

数组是一种包含多个同种类型元素的数据结构。其元素的数据类型可以是基本类型，如整型，也可以是引用类型。数组有一维数组，也有多维数组。其访问方式是采用数组名称及不同的索引来实现。

5.1 声明及初始化

在使用数组前，首先要声明数组，声明数组的过程，即确定数组所存储的数据类型、存储空间大小的过程。一维数组声明的一般形式是：

```
类型 [ ] 数组名称 = new 类型[ 数组大小 ];
```

其中特别需要注意的是，“=”左侧一对中括号的位置。另外，数组大小指明了数组所能容纳的元素个数，数组的索引是从 0 开始的。

```
int [ ] iSeason = new int[4]; // 声明一个整数类型数组，数组共 4 个元素

// 下面声明一个字符串类型，数组共 3 个元素 即 sName[0]、sName[1]、sName[2] 三个元素
string [ ] sName = new string [3];
```

当然，上述声明方式可以分为两步来进行，形式如下：

```
类型 [ ] 数组名称;
数组名称 = new 类型[ 数组大小];
```

例如：

```
int [ ] iMonth;
iMonth = new int[4]; // 声明一个整数类型数组，数组共 4 个元素
```

虽然一维数组的使用频率最高，然而在某些场合下，可能需要高维数组。其一般声明方式如下：

```
类型 [, , ..., ] 数组名称 = new 类型[ 第一维大小, 第二维大小, ..., 第 N 维大小];
```

例如,下面声明了一个二维数组:

```
int [,] iTable = new int[5,6]; // 声明一个二维整数类型数组,数组共 30 个元素
```

当然,也可以将其更改为两行,先声明数组变量,再通过 new 为其开辟内存空间。

声明数组的目的是在其中存放数据,要使用数组,往往需要先初始化该数组。假如一个数组只是声明而没有完成初始化赋值,其中存放的数据如何呢? 看下面的例子。

```
// 枚举类型定义
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

// 测试代码
int [] iArr = new int[2];
double [] dArr = new double[2];
bool [] bArr = new bool[2];
char [] cArr = new char[2];
string [] sArr = new string[2];
Season [] eArr = new Season[2];
Program [] pArr = new Program[2];

for (int i = 0; i < 1; i++)
{
    Console.WriteLine("-----");
    Console.WriteLine(iArr[i]); // 0
    Console.WriteLine(dArr[i]); // 0
    Console.WriteLine(bArr[i]); // False
    Console.WriteLine(cArr[i] == 0 ? "0" : cArr[i].ToString()); // 0
    Console.WriteLine(sArr[i] == null ? "null" : sArr[i]); // null
    Console.WriteLine(eArr[i]); // Spring
    Console.WriteLine(pArr[i] == null ? "null" : sArr[i]); // null
}
```

注意: 上述 Program 为类名称。

可见,数组元素的默认值规则如表 5-1 所示。

表 5-1 数组元素默认值规则

类 型	默 认 值	类 型	默 认 值
数值类型	0	枚举类型	0
字符类型	0	引用类型	null
布尔类型	false		



思考: 上述枚举类型的输出为什么是 Spring?

再来看数组的初始化问题。数组的初始化可以采用如下通用形式：

```
类型 [ ] 数组名称 = new 类型[ 数组大小]{ 与数组大小相等个数的元素值列表 };
```

当初始化时,初值的个数一定要与数组大小相等,否则会出现编译错误。

例如下面同时完成数组的定义和初始化。

```
int [ ] iSeason = new int[4]{1,2,3,4};    // 声明一个整数类型数组,数组共 4 个元素
```

当初值的个数一定要与数组大小不相等时,则会出现错误,例如:

```
int [ ] iSeason = new int[4]{1,2,3};      // 错误  
int [ ] iSeason = new int[4]{1,2,3,4,5};  // 错误
```

为了书写方便,也可以采取如下简洁的初始化方式:

```
int [ ] iSeason = {1,2,3,4};              // 声明一个整数类型数组,数组共 4 个元素
```

该方式也可以用于多维数组。例如:

```
int [, ] MultiArr = {{1,2},{3,4},{5,6}};  // 声明了一个 3 * 2 的数组
```

以该方式声明时,并不需要指定数组的大小,而由右侧初始值个数来决定数组的大小。但是,需要注意的是,该方式只能用于在声明的同时并初始化。若不是初始化的场合,则不能使用此种方式,例如下面的代码是错误的。

```
int [ ] iSeason;  
iSeason = {1,2,3,4};                      // 错误
```

 在 C# 中还有一种特殊的二维数组,即锯齿数组,其声明方式如下:

```
类型 [][] 数组名 = new 类型[n+1][];  
数组名[0] = new int[i];  
数组名[1] = new int[j];  
...  
数组名[n] = new int[k];
```

例如:

```
int [][] intArr = new int[3][];  
intArr[0] = new int[3]{100,200,300};  
intArr[1] = new int[5];  
intArr[2] = new int[7];
```

不过该声明方式不能跨语言,违背公共语言规范,故不推荐。

5.2 访问与遍历

数组是通过数组名和索引来进行访问的,索引从 0 开始。例如:

```
int [ ] iSeason = new int[4]{1,2,3,4};           // 声明一个整数类型数组,数组共 4 个元素
```

由于数组索引从 0 开始,故上述声明与下述代码效果等价。

```
int [ ] iSeason = new int[4];                     // 声明一个整数类型数组,数组共 4 个元素
iSeason[0] = 1;
iSeason[1] = 2;
iSeason[2] = 3;
iSeason[3] = 4;
```

上述代码的作用是将数据存入数组中,存入时通过数组名和索引来进行的。同理,也可以采用此种方式从数组中读取数据。例如:

```
int [ ] iSeason = new int[4]{1,2,3,4};           // 声明一个整数类型数组,数组共 4 个元素
int iResult = iSeason[0] + iSeason[1] * iSeason[2] - iSeason[3];
Console.WriteLine(iResult);                       // 输出 3
// Console.WriteLine(iSeason[4]);                 // 越界错误
```

数组的每个元素都可以采用如上方式来访问,然而最实用的访问数组元素的方式是循环遍历。当对数组元素进行循环遍历时,需要借助于数组的一个属性——Length,借助于该属性,可以避免数组访问出现越界错误。

示例: 数组遍历——for 循环

```
int [ ] iSeason = new int[4]{1,2,3,4};
for (int i = 0; i < iSeason.Length; i++)
{
    Console.Write(iSeason[i] + "\t");              // 1 2 3 4
}
```

除了 for 循环,还有一种对数组遍历的常用方式——foreach。其语法如下:

```
foreach (数据类型 变量 in 数组名)
```

使用 foreach 的特点如下。

- 不需要设置循环条件和迭代变量,更简单快捷,也更安全。
- 该循环读出的元素值是只读的,不可以修改。

示例: 数组遍历——foreach

```

int [ ] iSeason = new int[4]{1,2,3,4};
foreach (int i in iSeason)
{
    Console.Write(i + "\t");           // 1 2 3 4
    // i = i + 1;                       // 错误,foreach 中无法对读出的值进行修改
}

```

下面再举一个例子来实现数组的复制,同时将通过两种复制方式的对比来加深读者对有关内容的理解。

```

int[] iSrc = new int[4] { 1, 2, 3, 4 };
int[] iDes1 = new int[iSrc.Length];
int[] iDes2 = new int[iSrc.Length];
iDes1 = iSrc;                               // 数组复制
Console.WriteLine(" iDes1 的元素为:");
foreach (int i in iDes1)
    Console.WriteLine(i + "\t");

// 给 iDes2 循环赋值
for (int i = 0; i < iSrc.Length; i++)
    iDes2[i] = iSrc[i];
Console.WriteLine("\n iDes2 的元素为:");
foreach (int i in iDes2)
    Console.WriteLine(i + "\t");

Console.WriteLine("\n 更改 iSrc[2] = 100 ");
iSrc[2] = 100;
Console.WriteLine(" iDes1 的元素为:");
foreach (int i in iDes1)
    Console.WriteLine(i + "\t");

Console.WriteLine("\n iDes2 的元素为:");
Console.WriteLine(string.Join("\t", iDes2)); // string.Join() 参见第 6.2.1 节.

```

程序运行结果如图 5-1 所示。

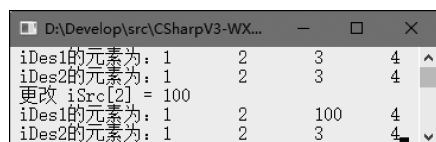


图 5-1 数组复制演示

对比上述运行结果,发现两种数组复制方式都可以成功地复制,在对 iDes1 和 iDes2 中的元素进行输出时即可看出。然而在源数组的内容发生改变之后,再对 iDes1 和 iDes2 的元素进行输出时,发现 iDes1 和 iDes2 不再完全一样了。

上述这种区别的根本原因在于:使用 iDes1=iSrc 完成数组的复制,复制的是引用,其

本质在于使得 iDes1 与 iSrc 指向了相同的存储空间,故对 iSrc 的更改会反映到 iDes1 身上。而 iDes2 则具有自己的存储空间,完成复制后,它与 iSrc 不再有任何牵连。

思考: 请仔细体会第 4.5.5 节关于 ref 的解释和上述示例的内在关联。

在很多情况下,方法的参数个数是不可预知的。当多个参数的数据类型一致时,可以以数组来作为参数,这个时候看似参数只有一个,但是却可以传递很多的数值到方法内部去。典型的应用如求和、求最大值、最小值等。

示例: 数组参数

根据提供的一系列数值,求其和值、最大值、最小值。

```
public class Score
{
    public int GetMax(int[] list)
    {
        int max = 0;
        foreach (int i in list)
        {
            if (i > max)
                max = i;
        }
        return max;
    }

    public int GetSum(int[] list)
    {
        int sum = 0;
        foreach (int i in list)
            sum += i;
        return sum;
    }
}

static void Main(string[] args)
{
    Score s = new Score();
    int[] scores = new int[] { 5, 6, 0, 8, 9, 1 };
    Console.WriteLine("最大值是:" + s.GetMax(scores));           // 最大值是:9
    Console.WriteLine("和值是:" + s.GetSum(scores));             // 和值是:29
}
```

思考: 请在上述 Score 类中添加一个方法,该方法用于完成求最小值的功能。

提示: 可以定义 `int min = int.MaxValue;`,然后逐个与 min 相比较,只要某个值小于 min,则将该值赋给 min 来实现。

思考: 还记得前文所介绍的 params 关键字吗? 请使用 params 关键字实现上述例子,并比较数组参数和 params 参数的异同。

5.3 Array 类

C# 中的数组继承自 System.Array 类。该类提供了一系列实用的方法,用于进行数组的相关操作:创建、修改、搜索、排序等。

1. Array 的常用属性

Array 的常用属性如表 5-2 所示。

表 5-2 Array 常用属性

属 性	说 明
IsFixedSize	总是 true
IsReadOnly	总是 false
Length	32 位整数,表示所有元素个数
LongLength	64 位整数,表示所有元素个数
Rank	获取 Array 的维数(秩)

演示: Array 属性演示

```
int[,] iNum = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
Console.WriteLine(iNum.Length);           // 6
Console.WriteLine(iNum.Rank);              // 2
Console.WriteLine(iNum.IsReadOnly);        // False
```

2. Array 的常用方法

Array 的常用方法如表 5-3 所示。

表 5-3 Array 常用方法

方 法	说 明
Clear()	将元素设置为默认输出值 0 或 null
Clone()	创建数组的拷贝
Copy()	将当前一维数组复制到指定的一维数组中
CreateInstance()	根据提供的参数创建一个 Array 类的新实例,即动态创建数组
GetLength()	获取数组指定维的元素个数
GetLowerBound()	获取数组中指定维度的下限
GetUpperBound()	获取数组中指定维度的上限
GetValue()	获取当前数组中指定元素的值
Reverse()	反转给定的一维数组元素的顺序
SetValue()	给当前数组中的指定元素赋值
IndexOf()	某个值在数组中首次出现的索引
Sort()	对数组元素进行排序

示例: Array 的常用方法

```
Array myArr = Array.CreateInstance(typeof(Int32), 7); // 动态创建数组
for (int i = 0; i < myArr.Length; i++)
    myArr.SetValue((i + 1) * 10, i); // 分别给每个元素赋值
Console.WriteLine(" myArr 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + myArr.GetValue(i).ToString()); // 分别读取数组的每个值

int[] iDes = new int[myArr.Length];
Array.Copy(myArr, iDes, myArr.Length - 2); // 将 myArr 的前 7 - 2 = 5 个元素复制到 iDes

int[] iDes2 = new int[myArr.Length + 3];
myArr.CopyTo(iDes2, 2); // 将 myArr 的值复制到 iDes2 中, 目标位置从索引 2 开始存储
Console.WriteLine("\n iDes 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + iDes.GetValue(i).ToString());

Console.WriteLine("\n iDes2 数组元素为:");
for (int i = 0; i < iDes2.Length; i++)
    Console.WriteLine("\t" + iDes2[i].ToString());

myArr.SetValue(200, 2); // myArr 的改变不会影响到 iDes 和 iDes2
Console.WriteLine("\n myArr 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + myArr.GetValue(i).ToString());

Console.WriteLine("\n iDes 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + iDes[i].ToString());

Console.WriteLine("\n iDes2 数组元素为:");
for (int i = 0; i < iDes2.Length; i++)
    Console.WriteLine("\t" + iDes2.GetValue(i).ToString());

Array.Sort(myArr); // 对 myArr 排序
Console.WriteLine("\n myArr 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + myArr.GetValue(i).ToString());

Array.Clear(myArr, 2, 3); // 将 myArr 数组从 index = 2 的位置开始, 清零 3 个值
Console.WriteLine("\n myArr 数组元素为:");
for (int i = 0; i < myArr.Length; i++)
    Console.WriteLine("\t" + myArr.GetValue(i).ToString());
```

程序执行结果如图 5-2 所示。

从程序运行结果可见：

- `CreateInstance(Type, Length)`：用于创建指定类型和大小的数组。
- `SetValue(value, index)`：用于给数组中索引为 `index` 的元素赋值 `value`。
- `GetValue(index)`：获取数组中索引为 `index` 的元素值。

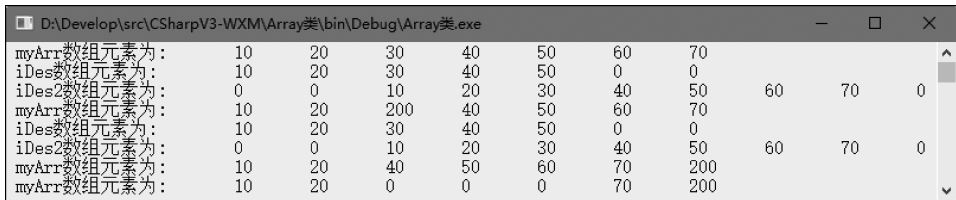


图 5-2 Array 方法演示

- `Array.Copy(arrSrc, arrDes, Length)`: 将源数组的前 `Length` 个值复制到目标数组中。
- `myArr.CopyTo(arrDes, index)`: 将 `myArr` 复制到 `arrDes` 中, 目标位置从 `index` 处开始。
- 无论是 `Copy()` 方法还是 `CopyTo()` 方法完成的复制, 源数组的改变不会影响到目标数组。
- `Array.Sort(myArr)`: 将 `myArr` 的元素按照升序排列; 其他重载请自行试验。
- `Array.Clear(myArr, 2, 3)`: 将 `myArr` 数组从 `index=2` 的位置开始, 对 3 个值清零。

5.4 聪明的数组——索引器

在第 2 章中曾演示过字符串和字符的例子, 代码如下:

```
string s = "China 中国";
Console.WriteLine(s[0]);           // 第 0 个位置 是 C
Console.WriteLine(s[2]);           // 第 2 个位置 是 i
Console.WriteLine(s[5]);           // 第 5 个位置 是中
```

即可以通过字符串变量名和索引位置来访问字符串中的每个字符, 此即典型的数组访问方式。我们知道, 字符串是一种引用类型, 而类也是一种引用类型, 既然如此, 自己定义的类, 能否也能实现像 `string` 这样的功能——类似数组的访问方式呢? 答案是肯定的。要实现这种效果, 需要使用索引器。

索引器是一个与属性很类似的类成员。也可以具有 `get` 和 `set` 两个访问器, 分别用于实现读和写功能。但索引器的主要不同之处在于: 定义索引器的时候一定要使用 `this` 关键字, 而不需要像定义属性时需要指定属性名字; 另外, 索引器一定需要参数; 最后, 索引器不能用 `static` 修饰。

索引器的用处主要在于为封装在类内部的数组或者集合提供一种类似于数组的访问方式, 即类似于上述 `string` 示例的访问方式。这样, 索引器同时兼具属性的特性和数组的便利访问特性。所以可以狭隘地这么认为, 索引器是一个比数组更聪明的一类数组, 或者说是一类像数组一样访问的属性。当然, 用索引器对集合进行封装也是可以的, 请读者在学习完集合后再自行实现。

索引器定义的一般形式如下:

```
访问修饰符 类型 this [参数列表]
{
    get { // 返回参数所指定的元素值 }
    set { // 给参数所指定的元素赋值 }
}
```

需要提及的是,虽然索引器可以使用多个参数,也可以使用多种类型的参数,但实际应用中一般只使用一个参数,并且该参数类型为 `int`。与属性一样, `get` 和 `set` 访问器可以根据具体需求来取舍,不一定两个都要实现。

下面通过例子来学习索引器,读者可以在示例的学习过程中体会索引器如何利用属性的特征,从而使得它比数组更聪明。

下面的示例将在类内部定义一个 `int` 型数组,然后以索引器完成对该数组的封装访问。由于该数组定义为 `int` 类型,而索引器的目的就是实现对该数组的访问,故索引器的类型也应该定义为 `int` 类型。例如:

```
class IndexDemo
{
    int[] iArr;           // 定义数组
    private int length;
    public int Length
    {
        get { return length; }
    }
    public bool IsSuccessful; // 显示操作结果是否成功,供调用方使用
    public IndexDemo(int length)
    {
        iArr = new int[length];
        this.length = length; // 当参数和字段重名时,可以借助 this 关键字来区分两者
    }
    // 索引器
    /* 看下面的代码,很容易可以看出,借助索引器的属性特性,可以实现聪明的数组,现在该数组不会再出现越界错误了
    */
    public int this[int index]
    {
        get
        {
            if (index >= 0 & index < Length)
            {
                IsSuccessful = true;
                return iArr[index];
            }
            else
            {
                IsSuccessful = false ;
                return 0;
            }
        }
    }
}
```

```

    }
}
set
{
    if (index >= 0 & index < Length)
    {
        iArr[index] = value;
        IsSuccessful = true ;
    }
    else
        IsSuccessful = false ;
}
}
}

```

观察上述代码不难发现索引器的聪明特性。由于对 index 的判断,避免了数组的一大错误——越界。不过这也带来一个问题,用户如果访问越界,异常不再触发,即虽然出错了,但是用户不知道,所以程序采用了另外一种机制来弥补这个缺陷,即通过 IsSuccessful 字段来告知调用方其调用结果是否成功,使得用户完全掌握自己的程序执行的成功与否。

演示调用代码如下:

```

static void Main(string[] args)
{
    IndexDemo indexDemo = new IndexDemo(3);
    Console.WriteLine("非法存取而不导致报异常的演示:");

    // 很明显,如果是普通的数组采用如下的 <= 会导致越界异常,但采用如上索引器不会
    for (int i = 0; i <= indexDemo.Length; i++)
        indexDemo[i] = i * 2;
    for (int i = 0; i <= indexDemo.Length; i++)
        Console.Write(indexDemo[i] + "\t");

    Console.WriteLine("\n\n利用 IsSuccessful 完成错误处理的演示:");
    for (int i = 0; i <= indexDemo.Length; i++)
    {
        indexDemo[i] = i * 2;
        if (!indexDemo.IsSuccessful)
            Console.WriteLine("indexDemo[" + i + "] 越界");
    }
}

```

程序执行结果如图 5-3 所示。

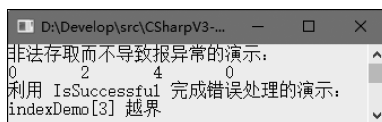


图 5-3 索引器

! 上述示例中的 IsSuccessful 字段其实设计为只读属性更为合理,请读者自行实现。

! 索引器也可以重载。请读者自行试验。

! 索引器的类型不一定非得与索引器内部的数组类型一致。

🕒 思考: 请仔细思考上述示例,如何通过索引器避免普通数组的缺陷。

现在学会了如何给类定义索引器,可以回过头去看看 string 是不是也是这么做的呢? 要想验证这个想法其实很简单,可执行以下步骤。

步骤 1: 输入 string

打开 VS,新建一个控制台项目,在 Main()函数内输入 string。

步骤 2: 查看 string 的定义

将光标定位在 string 上,按 F12 键,此时可以看到如图 5-4 所示效果。

```
...public sealed class String : IComparable, ICloneable, IConvertible, ICompa
{
    ...public static readonly string Empty;

    ...public String(char* value);
    ...public String(char[] value);
    ...public String(sbyte* value);
    ...public String(char c, int count);
    ...public String(char* value, int startIndex, int length);
    ...public String(char[] value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length, Encoding enc);

    ...public static bool operator !=(string a, string b);
    ...public static bool operator ==(string a, string b);

    ...public int Length { get; }

    ...public char this[int index] { get; }
```

图 5-4 string 查看

图 5-4 中着重色的部分即可说明“字符串是一个只读的字符数组”。

! 上面介绍的查看类定义的方法是通用的,读者也可以自己尝试使用上面的方法查看其他的类或者方法。

! 其实还有一个更强劲的工具,可以查看得更为详细彻底,那就是 .NET Reflector。

5.5 问 与 答

5.5.1 如何使用 Array.Sort() 来排序对象数组

✌ 要实现对象数组的排序问题,需要借助接口 System.IComparable,该接口用于比较同一对象的实例是否相等。其返回值为 0 则表示两个比较的对象相等;返回值小于 0,则表明当前实例小于参数实例,否则相反。

示例: Array.Sort() 的自定义排序

```
// 类 Person
public class Person: IComparable
{
```

```

private int sid;
public string Name;
public Person(int sid, string name)
{
    this.sid = sid;
    this.Name = name;
}
// 属性
public int ID
{
    get { return sid; }
    set { sid = value; }
}
// 无 public 等修饰符
int IComparable.CompareTo(object obj)
{
    Person s = (Person)obj;
    // 这里可以自行设定对象的比较究竟采取何种比较规则
    if (this.sid > s.sid)
        return 1;
    if (this.sid < s.sid)
        return -1;
    else
        return 0;
}
}
// 调用代码
static void Main(string[] args)
{
    string str = string.Empty;
    Person[] arr = new Person[4];
    arr[0] = new Person(65, "张三");
    arr[1] = new Person(21, "李四");
    arr[2] = new Person(1, "王五");
    arr[3] = new Person(3, "小赵");
    foreach (Person item in arr)           // 遍历数组中的数据
    {
        str = string.Format("{0} {1}", item.ID, item.Name);
        Console.WriteLine(str);
    }
    // 对象排序
    Array.Sort(arr);
    Console.WriteLine("***** 排序后的数据 *****");
    foreach (Person item in arr)
    {
        str = string.Format("{0} {1}", item.ID, item.Name);
        Console.WriteLine(str);
    }
}

```

运行结果如图 5-5 所示。

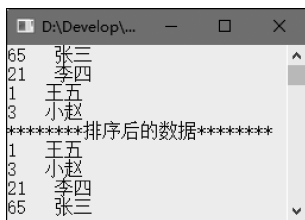


图 5-5 利用 Array.Sort()对数组排序

5.5.2 数组的大小真的没法调整吗

✌ 在比较集合和数组时,经常会说,数组大小固定。其实数组的大小也是可以调整的。使用的方法就是 Array 类的 Resize()方法,该方法是泛型方法,其声明如下:

```
public static void Resize<T> (ref T[] array, int newSize)
```

其中 array 为待调整大小的一维数组,如果为空,则新建大小为 newSize 的数组。如:

```
static void Main(string[] args)
{
    int[] iArr = { 100, 200 };           // iArr 数组初始 2 个元素
    Array.Resize<int>(ref iArr, 5);
    iArr.SetValue(300, 2);
    iArr.SetValue(500, 4);
    iArr.SetValue(1000, 0);
    foreach (int i in iArr)
        Console.Write(i + "\t");        // 1000 200 300 0 500
}
```

需要说明:该方法表面可以修改数组大小,但是其实它并非在原有的数组上做改变,而是产生了一个新的数组实例。所以数组一旦创建,其大小就不可以再改变。

5.5.3 如何判断数组中是否包含指定的元素

虽然该功能可以通过对数组进行遍历来判断,然而其实现较为烦琐。一个简洁的方式是使用其扩展方法 Contains()来实现。

```
static void Main(string[] args)
{
    string[] sArr = { "甲", "乙", "丙" };
    Console.WriteLine(sArr.Contains("甲"));           // true, 包含 甲
    Console.WriteLine(sArr.Contains("丁"));           // false, 不包含 丁
}
```

5.5.4 如何查找数组中具有特定特征的元素

✌ 要解决该问题,可以先遍历数组,然后逐个分析元素是否具有指定特征。这里介绍一个使用 Array 类的 FindAll()方法实现的方案。该方法声明为:

```
public static T[] FindAll<T>(T[] arrayToFind, Predicate<T> match);
```

其中 match 用来指定查找特征,传入一个与 Predicate<T>匹配的方法即可,Predicate<T>是一个委托,声明如下:

```
public delegate bool Predicate<T>(T obj);
```

用于判断 obj 是否匹配某种特征,若匹配返回 true,否则返回 false。
倘若只需要查找数组中的第一个符合条件的元素,则只需要使用 Find()方法即可。

```
static void Main(string[] args)
{
    // 下面示例演示如何在该数组中找到.cn 域名
    string[] sDomains = { "wsh.cn", "abc.com", "beido.net", "168.cn" };
    string[] sCN = Array.FindAll<string>(sDomains, CheckCN<string>);
    foreach (string s in sCN)
        Console.WriteLine(s + "\t"); // 输出 wsh.cn 168.cn
}
static bool CheckCN<T>(string sToCheck)
{
    if (sToCheck.EndsWith(".cn"))
        return true;
    else
        return false;
}
```

5.5.5 索引器的参数类型一定要为 int 吗

不是的。不过索引器的参数类型一般都习惯采用 int。例如:

```
class IndexDemo
{
    static DateTime sdt = new DateTime(DateTime.Now.Year, 1, 1);
    DateTime[] dts = { sdt, sdt.AddYears(1), sdt.AddYears(2), sdt.AddYears(3) };
    public string this[DateTime dateTime]
    {
        get
        {
            foreach (DateTime dt in dts)
            {
```

```
        if (dateTime.Year == dt.Year)
            return dt.ToString();
    }
    return "非未来 3 年之内的日期";
}
}
```

上面的示例中,内部数组类型为 DateTime 类型的,而索引器的返回类型为 string 类型的;索引器的参数类型也不是 int 类型的。调用代码如下:

```
static void Main(string[] args)
{
    IndexDemo idx = new IndexDemo();
    Console.WriteLine(idx[DateTime.Now]);           // 2018/3/25 0:00:00
    Console.WriteLine(idx[DateTime.Now.AddYears(10)]); // 非未来 3 年之内的日期
    Console.WriteLine(idx[DateTime.Now.AddYears(-1)]); // 非未来 3 年之内的日期
}
```

5.5.6 如何不计算即可获得最大值、最小值、和值、平均值

 对于本章所学的数组,可方便地利用扩展方法完成上述功能。看下面的例子,定义一个数组,然后使用循环给各个元素赋值,最后使用扩展方法完成上述功能。例如:

```
static void Main(string[] args)
{
    int[] num = new int[100];
    for (int i = 0; i < 100; i++)
        num[i] = i + 1;
    Console.WriteLine("最大值:" + num.Max());           // 最大值:100
    Console.WriteLine("最小值:" + num.Min());           // 最小值:1
    Console.WriteLine("和值:" + num.Sum());             // 和值:5050
    Console.WriteLine("平均值:" + num.Average());       // 平均值:50.5
}
```

5.6 思考与练习

- (1) 请比较总结使用普通数组和使用 params 方式传值的异同之处。
- (2) 请总结数组复制的方法及各个方法的特性。
- (3) 随机生成 20 个整数,并且这 20 个随机数的正负性也随机处理,然后将这 20 个随机数存入数组,最后将这 20 个随机数中的正数存入另外一个数组。
- (4) 随机生成 1000 个 1~50 的正整数,并统计各个数值出现的频次,且输出频次时,每行输出 5 个数值的频次。

(5) 实现一个简单的数组处理类,要求如下:

- 实现整型数组元素的排序输出。
- 通过重载实现字符数组的排序输出。
- 对整型数组进行求和。
- 对整型数组求最大值。
- 实现字符反转。
- 对整型数组求最小值(使用 params 方式)。
- 若无特别说明,传入参数时,要求传入数组名称。