第 3 章

面向对象编程

面向对象编程(Object Oriented Programming,OOP)是一种很重要的程序设计模型,也是一种广泛应用的编程思想。通俗地讲,面向对象编程认为应用程序是由许多单个对象组成的,对象自身具有很大的灵活性、封装性和扩展性,既能方便开发者管理代码结构,也比较容易将代码模型和程序的业务逻辑紧密结合。

关于面向对象的各种理论和概念,读者不需要花过多的精力去记忆,因为可以通过网络搜索或查阅相关书籍,大致知道这些概念是怎么回事就可以了。要理解面向对象编程,还得从实战入手,要通过编写代码,不断地练习,才能对面向对象编程有更为直观的理解。

C#是一种完全面向对象的编程语言。在C#代码中可以使用的对象有类、结构、枚举、接口和委托。这些内容将在本章中一一讲述。

3.1 类

类是从客观事物中进行抽象和总结出来的“蓝图”。例如,自行车是一种类型,它有几个轮子,高度是多少,是否具备变速功能等。

笔者在第 2 章中讲述基本数据类型时曾提到过,定义数据类型是为了能够更好地组织和存储数据,这些数据是临时的,只存在于内存中,随时可以被清理,变量就是用于存放与某个类型相关的数据。

既然数据类型要存储数据,它内部肯定会包含必要的成员。比如一个企业内部有多个职能部门(财务部、市场部、人力资源部等),每个部门负责不同的工作,彼此协作,整个企业才能正常运作。因此,类的内部也会定义不同的成员,这些成员包括:

(1) 属性。属性用于描述对象的特征。例如,对于一个汽车类来说,可以用产品型号、颜色、最大时速等特点来描述,这些都是汽车的属性。

(2) 方法。我们可以把方法比喻为对象的行为。例如,一个表示人的类,他可以在打球,在跑步,在说话,在看电视等。

(3) 事件。事件是在特定条件下触发的行为,可以理解为“条件反射”,例如下课铃响了,学生们就知道放学了。再如,一个气球内部充满了气体,然后拿一根针去扎它一下,由于

遇到被扎这一事件,气球会做出响应——爆裂。

(4) 构造器。也叫构造函数、构造方法。它是一种特殊的方法,在创建对象实例时调用,用来进行一些初始化工作。

定义类使用 class 关键字,如下面代码所示,定义了一个表示图书的 Book 类。

```
class Book
{
    // 类的成员
}
```

注意,关键字和类型名称之间要有空格。

3.1.1 字段

字段其实是在类(或结构)内部定义的一种变量。例如:

```
struct Point
{
    public int X;
    public int Y;
}
```

上面的代码的定义了一个 Point 结构,它表示一个平面坐标点,其中 X 和 Y 两个字段分别表示横坐标和纵坐标的值。再如:

```
class Student
{
    string name;
    int age;
    string address;
}
```

上面的代码定义了一个表示学生信息的 Student 类,其中包含三个字段:name 表示学生姓名,age 表示学生年龄,address 表示学生的住址。

3.1.2 属性

属性用于描述类的特征,它可以对字段进行封装。通常属性带有 get 和 set 访问器,get 访问器用来获取属性的值,而 set 访问器则用来设置属性的值。

再定义一个 Student 类,不过,把 name、age、address 三个字段用属性来封装。代码如下:

```
class Student
{
    // 姓名
    string name;
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }

    // 年龄
    int age;
    public int Age
    {
        get { return this.age; }
        set { this.age = value; }
    }

    // 住址
    string address;
    public string Address
    {
        get { return this.address; }
        set { this.address = value; }
    }
}
```

以 Name 属性为例，当获取属性的值时，通过 get 访问器将 name 字段的值直接返回；修改 Name 属性的值，通过 set 访问器把新值传递给 value 关键字，然后再把 value 赋值给 name 字段。另外两个属性情况类似。如果希望让属性只读，即只能获取其值而不允许对其进行赋值，直接去掉 set 访问器即可，仅保留 get 访问器。

通过上面的分析，读者会发现，字段是真正存储数据值的变量，而属性只是一个对外公开的一个“窗口”，数据通过属性来传递。当获取属性的值时，可以通过 return 关键字直接把字段中存放的值返回。当要设置属性的值时，调用 set 访问器把外部传进来的数据存放到 value 中，再以 value 作为纽带把数据赋给字段。

上面的示例似乎不足以说明为什么要使用属性。所以，接下来可以把上面的 Student 类进行如下修改：

```
class Student
{
    // 姓名
    string name;
    public string Name
```

```
        {
            get { return this.name; }
            set
            {
                if (value == "")
                {
                  throw new ArgumentException("姓名不能为空字符串.");
                }
                this.name = value;
            }
        }

        // 年龄
        int age;
        public int Age
        {
            get { return this.age; }
            set
            {
                if (value < 1 || value > 100)
                {
                  throw new ArgumentException("年龄超出了有效范围.");
                }
                this.age = value;
            }
        }
    …
    }
```

经过修改后，Name 属性不接受空白字符串，Age 属性不接受小于 1 或大于 100 的整数。如果设置的属性值不符合要求，就会抛出异常，即发生错误。因此上面代码充分展示了属性封装的好处，无论是获取还是设置属性的值，代码都可以事先做出相应的验证和处理，避免属性被设置为意外的值。如果直接把字段暴露给外部的调用代码，则字段很有可能被赋了不满足要求的值，严重时可能会破坏整个类的数据结构。

如果属性值不需要特殊验证处理，可以使用简化的属性声明语法。例如：

```
public string Name { get; set; }
```

在编译时，会自动生成存储属性值的字段。由于这种简练语法省去了封装私有字段的过程，若希望在声明属性时设置默认值，可以在属性声明后面直接赋值。例如：

```
public int MyValue { get; set;} = 700;
```

对于只读属性，只需要在声明时直接忽略 set 语句即可，例如：

```
public string ProductNo { get; }
```

对于只读属性，还可以使用类似 Lambda 表达式的形式来声明。例如：

```
public int MaxTaskNum => 500;
```

MaxTaskNum 属性是只读属性，返回整数值 500。当使用“=>”操作符来声明只读属性时，不需要写 get 语句，也不需要 return 关键字，“=>”后面直接写上要返回的值即可。

3.1.3　方法

方法可以认为是类的行为，通常指的是一个动作。请考虑下面代码：

```
class Music
{
    public string Title { get; set; }

    public int Year { get; set; }

    public void Play()
    {
        // 方法体内容
    }
}
```

上面的代码使用了属性的快速定义方式，请读者回忆上文中与属性有关的内容。

代码定义了一个 Music 类，表示一段音乐的基本信息 Titlet 和 Year 是属性，用于描述音乐的特征（标题、发行年份），而 Play 是方法，因为播放音乐是一种行为，void 表示方法不带有返回值。

如果希望方法返回处理结果，可以定义带返回值的方法，例如：

```
int ReturnInt()
{
    return 100;
}
```

ReturnInt 方法调用后会返回一个整数 100。有时候需要提供一些数据给方法内部的代码进行处理，这样一来，方法不仅需要返回值，而且还得用上参数。例如下面的 Add 方法：

```
int Add(int a, int b)
{
    return a + b;
}
```

Add 方法的功能是计算两个整数的和，所以它不但要返回计算结果，还需要提供两个参数 a 和 b，以便代码在调用时可以传递用来进行加法运算的两个操作数。例如，可以这样调用：Add(2, 3)，方法执行完成后返回 5。

在调用方法时，最常用的方法是依据参数定义的类型和顺序来传递，如上面的 Add(2, 3)，2 就传递给参数 a，3 便传递给参数 b。那么，如果不想按照参数的声明顺序来传递，又如何处理呢？

方法也很简单，在调用方法时写上参数的名字就可以了，例如：

```
Add(b:5, a:3)
```

写上参数的名字，后跟一个冒号(英文)，然后再写上要传递的值，如上面的代码，传递给 a 参数的值是 3，而传递给 b 的参数是 5。

读者还可以在方法中定义可选参数，顾名思义，就是在调用方法时，可以忽略的参数。正因为如此，可选参数要赋默认值。举个例子：

```
void DoWork(string p1, string p2 = "abc")
{
    // 方法内容
}
```

在这个方法中，p1 是必选参数，p2 由于已赋了默认值，就成了可选参数了。DoWork 方法可以这样调用：

```
DoWork("123");
```

因为 p2 是可选参数，所以以上调用是允许的。但是，如果把 DoWork 方法改为以下形式，就会出错：

```
static void DoWork(string p1 = "abc", string p2)
{
    // 方法内容
}
```

此时，p1 就成了可选参数了，如图 3-1 所示，如果仍然采取上面的调用方式，就会提示错误。即使在代码提示中为 p1 加上中括号(凡是可选参数，在提示中都会加上中括号)，也无济于事。

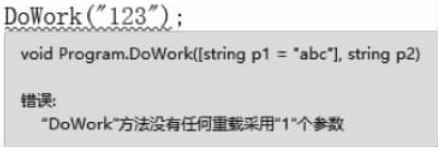


图 3-1　错误提示

由此可见，可选参数要放在参数列表的最后才合理，因为如果可选参数放在前面，而代码在调用时又忽略掉，那么编译器就无法让传入的值与参数列表一一对应了。

跟前面属性的声明相似，方法也可以使用 Lambda 表达式的形式来声明。例如：

```
public string PickName() => "Jack";
```

PickName 方法没有参数，返回一个字符串实例。

同样，带参数的方法也可以用 Lambda 表达式来声明。不妨把上面举例的 Add 方法修改为：

```
public int Add(int a, int b) => a + b;
```

在“=>”操作符右边可以省略 return 关键字，直接写上 b + b 的运算结果即可。

3.1.4　构造函数与析构函数

构造函数是在类被实例化的时候（即创建类的对象实例时）调用，它也是类的成员，具有以下特点：

（1）构造函数的名称必须与类名相同。

（2）构造函数没有返回值。

（3）默认构造函数没有参数，但也可以定义参数。

即使开发人员不为类编写构造函数，它默认就有一个不带参数的构造函数。考虑以下代码：

```
class Car
{
    // 内部代码
}

class Car1
{
    public Car1() { }
}
```

Car 和 Car1 的定义其实是一样的，如下面代码所示，在使用 new 运算符创建类的实例时，所产生的结果是相同的。

```
Car c1 = new Car();
Car1 c2 = new Car1();
```

既然是有默认的无参数的构造函数，开发者为什么还要自己去写一个呢？如果希望在类型初始化的过程中加入自己的处理代码，就有必要自己来定义构造函数了。另外，如果要使用带参数的构造函数，就得自己来写了，因为类型默认的构造函数是无参数的。

考虑以下代码：

```
class Toy
{
    public Toy(string name)
    {
        Console.WriteLine("正在创建{0}玩具.", name); ;
    }
}
```

我们为 Toy 类定义了一个带字符串类型参数的构造函数。但是,读者会发现,如果在创建 Toy 类的实例时无法再使用无参数的默认构造函数了,如图 3-2 所示。

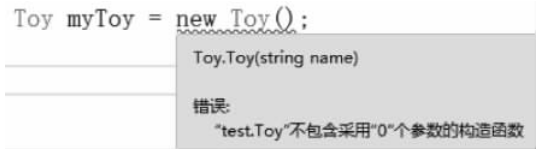


图 3-2　默认构造函数无法使用

这表明,如果自己编写了构造函数,那么默认构造函数就会被覆盖了,如果仍然希望用无参数的构造函数,就必须把无参构造函数也一并写上,所以上面代码应该作如下修改:

```
class Toy
{
    // 无参数的构造函数
    public Toy() { }

    // 带参数的构造函数
    public Toy(string name)
    {
        Console.WriteLine("正在创建{0}玩具.", name); ;
    }
}
```

要使用带参数的构造函数来创建类型的实例,就需要传递数据给对应的参数,例如:

```
Toy thetoy = new Toy("小汽车");
```

对象实例是暂存在内存中的,它不可能永远存在,在不需要使用时就会被清理。在 C++中,类实例是通过调用构造函数创建,通过调用析构函数来销毁的。在 C# 中也可以为类型编写析构函数,如下面代码所示,为 Toy 类加上析构函数。

```
class Toy
{
    // 无参数的构造函数
```

```
    public Toy() { }
    // 析构函数
    ~Toy() { }
…
}
```

注意，析构函数是以“~”开头的，没有返回值，后紧跟类名，无参数。析构造函数只能在类中使用，而且只能有一个析构函数。不能在代码中去调用析构函数，它是在资源被销毁时由运行时库调用。同时会调用 Object 类的 Finalize 方法，前面曾提到过，.NET 中的所有类型都是从 Object 类派生的，因此不管被定义的是类还是结构，或者是其他数据类型，在这些类型的实例被销毁时都会调用公共基类 Object 的 Finalize 方法。但是，由于 C＃编译器无法直接调用 Finalize 方法，所以在代码中无须直接重写 Finalize 方法。相关的内容读者可以参考 MSDN 文档上的说明。

如果存在需要开发者手动进行清理的资源，除了使用析构函数，还有以下替代方案：

```
class Speaker:IDisposable
{
    public void Dispose()
    {
        // 在这里进行清理
    }
}
```

参考上面的代码，读者可以看到，实现 IDisposable 接口，在 Dispose 方法中写上自己的处理代码。使用方法如下：

```
Speaker sp = new Speaker();
// 其他代码
sp.Dispose(); //执行清理
```

读者还可以把实现了 IDisposable 接口的类的实例化写到一个 using 语句块中，当代码执行完成 using 语句块时会自动调用对象的 Dispose 方法以释放占用的资源。例如：

```
using (Speaker sp = new Speaker())
{
    // 处理代码
}
```

这里的 using 语句和引入命名空间的 using 语句含义不同。这里的 using 语句是限定一个范围，当代码执行到这个范围的结尾时会自动释放实现了 IDisposable 接口的对象实例。

读者心中可能会产生一个疑问：一个类的实例在创建时真的会调用构造函数，在被销毁时调用析构函数吗？有没有办法来验证呢？

当然有，而且方法也不复杂，下面就来验证一下上述问题吧。请读者启动 Visual Studio 开发环境，然后新建一个控制台应用程序，然后在 Program.cs 文件中定义一个 Test 类，代码如下：

```
public class Test
{
    // 构造函数
    public Test()
    {
        System.Diagnostics.Debug.WriteLine("构造函数被调用.");
    }

    // 析构函数
    ~Test()
    {
        System.Diagnostics.Debug.WriteLine("析构函数被调用.");
    }
}
```

在 Test 类的构造函数和析构函数中分别使用 System.Diagnostics.Debug.WriteLine 方法来输出调试信息，这里不使用 Console.WriteLine 方法来输出是因为当 Test 类的实例被回收时应用程序已经结束，读者就看不到输出结果了，而使用 Debug 类输出的内容是显示在 Visual Studio 的“输出”窗口中，程序退出后这些信息还会保留，如果其中包含相关的文本就说明构造函数和析构函数被调用过。

接下来，在 Main 入口点中加入以下代码，创建 Test 类的实例。

```
static void Main(string[] args)
{
    Test t = new Test();
}
```

现在可以按下键盘上的【F5】键调试运行，很快应用程序就退出了。然后，通过菜单栏中的【视图】→【输出】命令打开“输出”窗口，如果代码正确执行了，就会在“输出”窗口中看到如图 3-3 所示的内容，这也说明 Test 类的构造函数和析构函数是被调用了的。

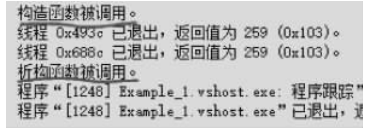


图 3-3 输出调试信息

完整的示例代码请参考\第 3 章\Example_1。

3.2 结构

结构与类比较相似，它内部同样可以包含字段、属性、方法等成员。但与类相比，结构有着许多限制，例如：

(1) 结构只能声明带参数的构造函数，不能声明默认构造函数，即不带参数的构造函数，而类是可以的，前面读者也学习过了。

(2) 结构不能进行继承和派生，但可以实现接口。结构默认是从 System. ValueType 派生，而类默认是从 System. Object 派生。所以类是引用类型，结构是值类型，有关这二者的区别，将在 3.3 节介绍。

(3) 结构在实例化时可以忽略 new 运算符，而类则不可以。

结构使用 struct 关键字来声明，例如：

```
struct Pet
{
    public string Name;
    public int Age;
}
```

在实例化 Pet 结构时，可以不用 new 来创建，如下面代码所示：

```
// 声明变量，但不需要 new 来实例化
Pet pet;
// 给 Pet 实例的成员赋值
pet.Name = "Jack";
pet.Age = 3;
```

当然也可以用 new 来创建实例：

```
// 使用 new 来创建实例
Pet pet2 = new Pet();
pet2.Name = "Tom";
pet2.Age = 2;
```

读者要注意一个问题，在结构中声明的字段不能进行初始化，如果把上面的 Pet 结构改为以下形式：

```
struct Pet
{
    public string Name = "";
    public int Age = 1;
}
```

就会发生错误,如图 3-4 所示。这是因为结构中的字段成员不能设定初始值。

图 3-4 结构中字段不能赋初始值

如果在结构中定义了属性或者方法等成员,那么也要注意一个问题。举个例子,请考虑下面代码:

```
struct Book
{
    public string Name { get; set; }

    public string ISBN { get; set; }

    public void Read()
    {
        Console.WriteLine("this book is reading...");
    }
}
```

Book 结构定义了两个属性,Name 表示书名,ISBN 表示图书的 ISBN 编码;而 Read 方法表示阅读此书的行为。现在,读者按照前面的做法,把 Book 结构实例化,代码如下:

```
Book theBook;
theBook.Name = "书名";
theBook.ISBM = "XXX-XX-X-XXXXX";
theBook.Read();
```

这时就会发生错误,如图 3-5 所示。提示未赋值的变量,这告诉我们,在未使用 new 关键字实例化结构的前提下,只能调用其字段,而属性、方法等成员无法调用。

于是,把代码进行如下修正:

```
Book theBook = new Book();
theBook.Name = "书名";
theBook.ISBM = "XXX-XX-X-XXXXX";
theBook.Read();
```

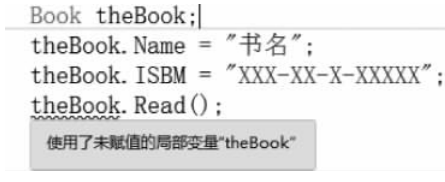


图 3-5 错误提示

经过以上的例子,读者会发现,结构的使用有着很多限制,不像类那样灵活。因此,如果要定义属性、方法、事件等成员,应当优先考虑使用类,结构一般用来定义一些比较简单的类型,比如只包含几个公共字段的结构就比较合理,类似于 C 语言中的结构体。当然,类和结构还有一个更值得关注的区别——类是引用

类型，结构是值类型。关于这个问题，将在 3.3 讲述。

3.3 引用类型与值类型

通常，类是引用类型，结构是值类型。那么什么是引用类型，什么是值类型？笔者希望通过两个实例来演示两者的区别。

第一个示例演示的是引用类型（完整代码位于第 3 章\Example_2）。首先定义一个 Person 类，代码如下：

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

随后，声明两个 Person 类型的变量，第一个变量创建一个 Person 实例；接着把第一个变量赋值给第二个变量，代码如下：

```
// 创建两个 Person 实例
Person ps1 = new Person { Name = "Time", Age = 22 };
// 把 ps1 赋值给 ps2
Person ps2 = ps1;
// 输出 ps2 的属性值
Console.WriteLine("ps1 被修改前：\nps2.Name：{0}\nps2.Age：{1}", ps2.Name, ps2.Age);
// 修改 ps1 的属性值
ps1.Name = "Jack";
ps1.Age = 28;
// 再次输出 ps2 的属性值
Console.WriteLine("\nps1 被修改后：\nps2.Name：{0}\nps2.Age：{1}", ps2.Name, ps2.Age);
```

第一次输出 ps2 的属性值的时候，ps1 的属性没有被修改，因此 ps2 的各个属性的值与 ps1 相同。但之后代码修改了 ps1 的 Name 和 Age 属性的值，我们重点关注这时候，ps2 中的各个属性的值会不会跟随着 ps1 一起发生改变。程序运行后输出的内容如图 3-6 所示。

图 3-6 两次输出 ps2 各个属性的值

请读者记录一下上述示例的执行结果。接下来要实现第二个示例，该示例演示值类型（完整的代码位于第 3 章\Example_3）。同样，先定义一个 Person 结构，注意是结构，不是类，代码如下：

```
struct Person
{
```

```
        public string Name { get; set; }
        public int Age { get; set; }
    }
```

和上一个示例类似,声明 Person 结构的两个变量,ps1 创建新的实例,然后赋值给 ps2,并输出 ps2 的各个属性的值。代码如下:

```
// 实例化 Person 结构
Person ps1 = new Person { Name = "Bob", Age = 21 };
// 将 ps1 赋值给 ps2
Person ps2 = ps1;
// 输出 ps2 的各个属性的值
Console.WriteLine("ps1 被修改前: \nps2.Name : {0}\nps2.Age : {1}", ps2.Name, ps2.Age);
// 修改 ps1 的属性值
ps1.Name = "Tom";
ps1.Age = 33;
// 再次输出 ps2 的各个属性值
Console.WriteLine("\nps1 被修改后: \nps2.Name : {0}\nps2.Age : {1}", ps2.Name, ps2.Age);
```

图 3-7 两次输出 ps2 的属性值

屏幕的输出结果如图 3-7 所示。

现在,读者可以对比一下以上两个示例的运行结果。在第一个示例中,把 ps1 赋值给 ps2 后,修改 ps1 也会同时改变 ps2,因为对于引用类型来说,实例化是在托管堆中动态分配内存的,变量只是保存该实例的地址。即 ps1 存的只是指向 Person 类的实例的引用的符号。哪怕是将 ps1 赋值给 ps2,ps2 中保存的还是指向那个 Person 实例的引用,它们所引用的是同一个实例,所以对 ps1 进行修改其实改变的是它所引用的那个实例,输出的 ps2 的属性值自然就是更新后的值了。可以用图 3-8 来演示这一过程。

而在第二个示例中,由于 Person 结构是值类型,它所创建的实例不在托管堆中分配内存,而是直接存储在变量中。当 ps1 赋值给 ps2 时,就等于把自己复制了一遍,包括其内部成员,即 ps1 把 Name 和 Age 属性的值一同复制到 ps2 中,就变成两个实例了,因此当代码修改了 ps1 的属性值后,与 ps2 并没有直接关系,它们是两个独立的实例。同样也可以用图 3-9 来演示这一过程。

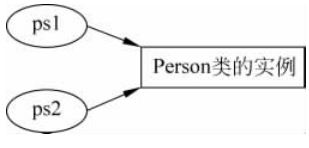


图 3-8 引用类型的实例示意图

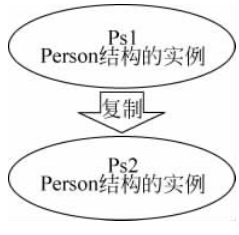


图 3-9 值类型的实例示意图

我们知道，在 Windows 操作系统中有一种特殊的文件类型叫快捷方式，其实引用类型与快捷方式很像。引用类型就相当于用户为某个文件创建快捷方式（比如桌面快捷方式），不管创建了多少个快捷方式，只要指向的是同一个文件，那么当这个文件被修改或者被删除后，会影响到指向该文件的所有快捷方式。

而值类型就相当于用户在操作系统中复制文件。例如把文件 A 从 C 盘复制到 D 盘，然后打开 D 盘下的 A 文件进行修改并保存。但是，存放在 C 盘中的 A 文件丝毫不受影响，因为这两个 A 文件是相互独立的。

3.4　ref 参数与 out 参数

在 3.3 节中已经对引用类型和值类型做出了比较，因此就会引出一个新的疑问：变量作为参数传给方法，同时希望在方法执行完成后，对参数所作的修改能够反映到变量上，该怎么处理呢？

对于引用类型是比较好处理的，因为引用类型的变量保存的是对象实例的地址，直接传递给方法的参数即可。可以用一个示例来验证。

完整的示例位于\第 3 章\Example_4。首先定义一个 Person 类，稍后用于作测试。

```
public class Person
{
        public string Name { get; set; }
        public int Age { get; set; }
}
```

接着定义一个 TestMethod1 方法，接受一个 Person 类型的参数，并在方法中修改它的属性。

```
/// <summary>
/// 参数传递的是引用，因此可以把更改反映到外部变量上
/// </summary>
static void TestMethod1(Person p)
{
        p.Name = "Loo";
        p.Age = 33;
}
```

然后，声明一个 Person 类型的变量，并调用 TestMethod1 方法。

```
Person ps = new Person { Name = "John", Age = 25 };
// 调用 TestMethod1 前
Console.WriteLine("在调用 TestMethod1 方法前，ps.Name : {0}, ps.Age : {1}", ps.Name, ps.Age);
TestMethod1(ps);
```

```
// 调用 TestMethod1 方法后,再次输出 ps 的属性值
Console.WriteLine("在调用 TestMethod1 方法后,ps.Name : {0}, ps.Age : {1}", ps.Name, ps.Age);
```

屏幕输出的结果如图 3-10 所示。在 TestMethod1 方法调用完成后,ps 的 Name 属性和 Age 属性将会发生改变。

图 3-10　变量 ps 的属性在调用方法后被修改

读者如果细心研究,还会发现一个很奇特的现象。下面再定义一个 TestMethod2 方法。

```
static void TestMethod2(Person p)
{
      p = new Person { Name = "Chen", Age = 29 };
}
```

这一次,代码不是修改 p 参数的属性,而是直接创建了一个新的实例,并且为 Name 属性和 Age 属性赋了值。代码中采用了一种简便的写法,即在 new 运算符后用一对大括号直接设定属性的值。例如,下面的两个写法都是允许的。

```
Person ps = new Person() { Name = "abc", Age = 22 };
Person ps = new Person { Name = "xyz", Age = 33 };
```

这两种写法的区别不大,就是在 new 后面调用构造函数时是否带有一对小括号。为什么会允许这两种写法呢? 不妨设想一下,如果类的构造函数没有参数(默认构造函数),就不必写上一对小括号;要是类的构造函数带有参数的话,肯定要传递参数的,因此小括号就不能省略了。

接着,调用 TestMethod2 方法。

```
Person ps2 = new Person();
ps2.Name = "Xin";
ps2.Age = 35;
Console.WriteLine("在调用 TestMethod2 前,\nps2.Name : {0}\nps2.Age : {1}", ps2.Name, ps2.
Age);
// 调用 TestMethod2 方法
TestMethod2(ps2);
// 再次输出 ps2 的属性值
Console.WriteLine("在调用 TestMethod2 后: \nps2.Name : {0}\nps2.Age : {1}", ps2.Name, ps2.
Age);
```

屏幕的输出结果如图 3-11 所示。

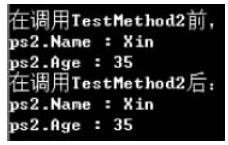


图 3-11　ps2 的属性在调用 TestMethod2 方法后不变

这时候读者可能感到疑惑，ps2 变量是引用类型的实例，传递到 TestMethod2 方法后就 new 了一个新的实例，而且还给属性赋了值的，为什么方法执行完成后 ps2 的属性值没有改变？ps2 在传给 TestMethod2 方法的参数时它自身被复制到参数 p，只不过它复制的是引用地址而不是对象实例本身罢了，也就是说，ps2 把它引用的实例的地址复制给了 TestMethod2 方法的参数 p，如果 TestMethod2 方法的参数 p 引用了其他对象的实例，那么参数中保存的引用就变了，但是它的改变并不与 ps2 有直接关系，因为 ps2 与参数 p 是两个独立的变量，只不过在进入 TestMethod2 方法时它们都引用了共同的实例而已。这就好比用户在操作系统中为文件 A 创建了快捷方式 AA，然后复制快捷方式 AA 到 BB，这时候快捷方式 BB 指向的依然是文件 A。但是如果把快捷方式 BB 更为指向文件 B，这时候快捷方式 AA 是不受影响的，它仍然指向文件 A。可以使用图 3-12 和图 3-13 来演示这一过程。

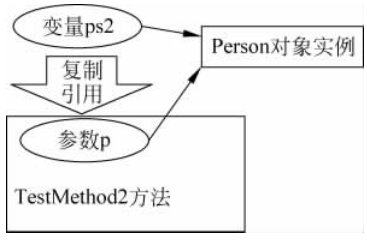


图 3-12　ps2 传入方法时的示意图

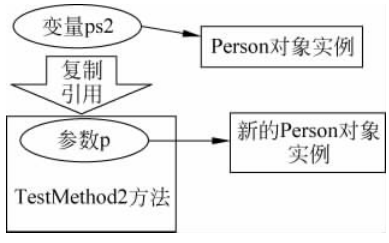


图 3-13　方法执行后 ps2 变量的示意图

要解决以上问题，就要考虑在定义方法的参数时加上 ref 或 out 关键字，这两个关键字比较相似，ref 参数在传入前必须先初始化；而 out 参数是不需要事先进行初始化，只要在传入前声明变量即可。现在，我们可以用 ref 和 out 关键字来解决上面的问题了。

```
static void TestMethod3(ref Person p)
{
        p = new Person();
        p.Name = "Lee";
        p.Age = 12;
}

static void TestMethod4(out Person p)
{
        p = new Person();
        p.Name = "Huang";
```

```
        p.Age = 27;
    }
```

上述两个方法的实现方式是一样的，只是 TestMethod3 方法使用 ref 关键字来修饰参数 p，TestMethod4 方法则使用 out 关键字来修饰参数 p。

随后我们分别调用这两个方法。

```
Person ps3 = new Person();
ps3.Name = "Li";
ps3.Age = 10;
Console.WriteLine("在调用 TestMethod3 方法前: \nps3.Name : {0}\nps3.Age : {1}", ps3.Name,
ps3.Age);
// 调用 TestMethod3 方法
TestMethod3(ref ps3);
// 调用 TestMethod3 方法后再次输出 ps3 的属性值
Console.WriteLine("在调用 TestMethod3 方法后: \nps3.Name : {0}\nps3.Age : {1}", ps3.Name,
ps3.Age);

Console.Write("\n\n");
Person ps4;
// 调用 TestMethod4 方法
// 变量在传递给 out 参数前可以不进行初始化
TestMethod4(out ps4);
// 调用 TestMethod4 方法后输出 ps4 的属性值
if (ps4 != null)
{
Console.WriteLine("在调用 TestMethod4 方法后: \nps4.Name : {0}\nps4.Age : {1}", ps4.Name,
ps4.Age);
}
```

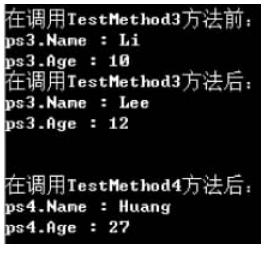


图 3-14 ref 和 out 参数调用结果

由于 out 参数允许传递未初始化的变量，所以在调用 TestMethod4 方法时，ps4 没有进行初始化，即为 null(空引用)。然后传递给 out 参数，在 TestMethod4 方法中为其赋值。屏幕的输出结果如图 3-14 所示。

在上面代码中，不管是使用 ref 关键字还是 out 关键字来修饰参数，在调用方法时也要带上相应的关键字。例如上面的 TestMethod3(ref ps3)和 TestMethod4(out ps4)。

用 ref 或 out 关键字来修饰方法参数还能解决值类型变量的传递问题。读者已经了解到，值类型变量之间的赋值是把自身进行一次复制。因此，把值类型的变量传递给方法的参数后，就把自身复制到参数中，而在方法中对参数的修改是不会影响到方法外部的变量的，因为它们是相互独立的。所以，为了让值类型的变量也能按引用传递，以达到修改外部变量的目的，可以在方法的相应参数上加上

ref 或 out 关键字。

同样，也可以用一个示例来演示（示例代码位于\第 3 章\Example_5）。首先定义用来做测试的 Dress 结构。

```
public struct Dress
{
    public string Color;
    public double Size;
    // 构造函数
    public Dress(string color, double size)
    {
      Color = color;
      Size = size;
    }
}
```

Dress 结构包含一个 string 类型的 Color 字段和一个 double 类型的 Size 字段。接下来定义三个方法：

```
static void F1(Dress d)
{
      d.Color = "红色";
      d.Size = 17.213d;
}

static void F2(ref Dress d)
{
      d.Color = "紫色";
      d.Size = 19.5d;
}

static void F3(out Dress d)
{
      d = new Dress("浅灰", 18.37d);
}
```

F1 方法的参数 d 不带任何修饰符，F2 方法的参数 d 用 ref 关键来修饰，F3 方法的参数 d 则使用 out 关键字来修饰。随后，代码分别调用这三个方法，读者要注意观察和比较屏幕上输出的信息。

```
Dress d1 = new Dress("白色", 3.125d);
Console.WriteLine("调用 F1 方法前: \nd1.Color:{0}\nd1.Size:{1}", d1.Color, d1.Size);
```

```
// 调用 F1 方法
F1(d1);
// 调用 F1 方法后输出 d1 的成员的值
Console.WriteLine("调用 F1 方法后: \nd1.Color:{0}\nd1.Size:{1}", d1.Color, d1.Size);
…
Dress d2 = new Dress("绿色", 8.5777d);
Console.WriteLine("调用 F2 方法前: \nd2.Color:{0}\nd2.Size:{1}", d2.Color, d2.Size);
// 调用 F2 方法
F2(ref d2);
// 调用 F2 方法后再次输出 d2 的成员的值
Console.WriteLine("调用 F2 方法后: \nd2.Color:{0}\nd2.Size:{1}", d2.Color, d2.Size);
…
Dress d3;
// 调用 F3 方法
F3(out d3);
// 调用 F3 方法后输出 d3 的成员的值
Console.WriteLine("调用 F3 方法后: \nd3.Color:{0}\nd3.Size:{1}", d3.Color, d3.Size);
```

F1 方法的参数不带任何修饰,因此调用时是按值来传递的,故在 F1 方法内部对参数 d 的修改不影响变量 d1,输出结果如图 3-15 所示。

F2 方法的参数加了 ref 关键字,因此调用时是按引用来传递的,所以在 F2 方法内部修改 d 参数会影响变量 d2,结果如图 3-16 所示。

F3 方法使用了 out 关键字来修饰参数 d(输出参数),所以在调用方法后会给变量 d3 赋值,如图 3-17 所示。

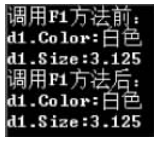


图 3-15 按值传递不影响变量 d1

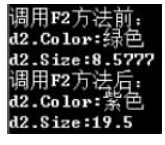


图 3-16 按 ref 传参后会影响变量 d2

图 3-17 修改 out 参数影响变量 d3

3.5 方法重载

所谓方法重载,就是在类型内部允许存在名字相同的方法,但不是重复,重复是不允许的。以下情况可以构成重载:

(1) 具有不同类型的返回值的同名方法可以重载。例如:

```
public string DoWork();
public int DoWork();
```

两个方法都命名为 DoWork，第一个 DoWork 方法的返回类型是 string；第二个 DoWork 方法则返回 int 类型。返回值的类型不同，所以它们可以重载。

（2）参数列表的类型及顺序不同，可以构成重载。例如：

```
void OnTest(string a) { }
void OnTest(float b, double a) { }
```

第一个 OnTest 方法只有一个参数，类型为 string；第二个 OnTest 方法有两个参数，分别是 float 类型和 double 类型，因此它们可以构成重载。但是下面的方法声明不能与上面的 OnTest 构成重载：

```
void OnTest(float a, double b) { }
```

虽然参数的名字不同，但是参数的类型和顺序相同，不能构成重载。编译器只关注参数的个数、类型和顺序，而参数的名字并不编译。例如：

```
void Send(int a, int b) { }
void Send(string a, string b) { }
```

上面两个 Send 方法虽然都有名为 a、b 的两个参数，但是它们的类型不同，前者是 int 类型，后者是 string 类型，故构成重载。

（3）带 ref 或 out 修饰符的参数。如果一个方法的参数带有 ref 关键字的参数，而另一个同名方法不带有 ref 关键字修饰的参数，那么这两个方法构成重载。例如：

```
void Compute(ref short v) { }
void Compute(short v) { }
```

同理，如果一个方法的参数使用 out 关键字修饰，而另一个与之同名的方法的参数不使用 out 关键字修饰，也能构成重载。如下面两个方法：

```
void Compute(short v) { }
void Compute(out short v) { v = 2; }
```

由于编译器不区分 ref 和 out 参数，所以如果一个方法使用 ref 参数，而另一个方法使用 out 参数，而且参数的个数、类型和顺序相同，则不能构成重载。

```
void Compute(ref short v) { }                    // 编译错误
void Compute(out short v) { v = 2; }             // 编译错误
```

以上两个方法名称相同，参数个数和类型相同，第一个 Compute 方法使用了 ref 参数，而第二个 Compute 方法使用了 out 参数，因此这两个方法不能重载，在编译时会报错误。

构造函数也是一种特殊的方法,因而也支持重载,前面在讲述类的定义的时候,其实读者已经接触过构造函数的重载了,即为类型定义多个构造函数。例如下面的 Goods 类:

```
public class Goods
{
    public Goods() { }
    public Goods(string goodsName) { }
}
```

在 Goods 类中,定义了两个构造函数,第一个是默认构造函数;第二个构造函数是一个重载,带有一个 string 类型的参数。

3.6 静态类与静态成员

static 关键字既可以修饰类型,也可以修饰类型中的成员。使用了 static 关键字即表示声明为静态类型或静态成员。静态是相对于动态而言的,本书在前面曾讲述过变量和常量,读者会了解到变量是动态声明的,而且可以动态地进行赋值。也就是说变量是在使用的时候才去分配内存的,即对象的实例。而静态类型或静态成员正好相反,它们不是基于实例的,所以在使用前不需要实例化,它们是基于类型本身的,直接就可以调用静态成员。

下面代码定义了一个 DataOperator 类,内部定义了两个静态方法 AddNew 和 UpdateNow。

```
public class DataOperator
{
    public static void AddNew() { }
    public static void UpdateNow() { }
}
```

在调用的时候,无须声明 DataOperator 类型的变量,也不需要进行实例化,而是直接调用它的公共方法即可。例如:

```
DataOperator.AddNew();
DataOperator.UpdateNow();
```

其实这个也很好掌握,直接写上类型的名字,然后用点号(成员运算符)来访问其公共成员即可。static 关键字不仅能修饰方法,同样也可以用来修饰字段、属性、事件等成员。例如,下面代码定义了两个静态属性。

```
public class Car
{
    public static string CarName { get; set; }
```

```
    public static double Speed { get; set; }
}
```

同理，向这两个属性赋值前也不用声明变量，也不用实例化，直接访问类型的成员即可。

```
Car.CarName = "高档汽车";
Car.Speed = 170d;
```

如果将 static 关键字用于修饰类型，就表明整个类型都是静态。在这种条件下，类型只能定义静态成员。例如，下面代码在编译时就会报错。

```
public static class Test
{
    public void SayHello() { }              // 错误
    public string Message { get; set; }     // 错误
}
```

Test 已声明为静态类，因此它只能定义静态成员，以上代码中的 SayHello 方法和 Message 属性是不能通过编译的。正确的代码如下：

```
public static class Test
{
    public static void SayHello() { }              // 正确
    public static string Message { get; set; }     // 正确
}
```

3.7　继承与多态

通过前面的学习，读者已学会类和结构的定义，尤其是类，因为它的应用更为广泛，因此本书随后会把与类有关的内容作为重点讲述对象。类型的定义可以体现面向对象编程中的封装性，在本节中，读者将会接触到继承性和多态性。

3.7.1　可访问性

在了解继承相关的知识之前，读者应当弄清楚各种可访问性的限制，才能更好地保护和管理自己编写的类型。常用的可访问修饰符可以参考表 3-1。

许多初学者会认为有关可访问性的内容不好记忆，其实学习编程是不用记什么东西的，希望读者不要去死记硬背。可访问性并不需要去记忆，只要多动手去写一下代码就能够掌握，在学习过程中，应该学会自己编写代码去验证问题。

表 3-1 可访问性修饰符

修 饰 符	说 明
public	无限制
internal	只允许在同一个程序集内访问
protected	通常是允许派生类访问
protected internal	实际上是 protected 和 internal 的合并
private	只能在当前类型中访问

请读者启动 Visual Studio 开发环境,然后按照以下步骤动手练习一遍,相信会有所收获的。

(1) 按快捷键【Ctrl + Shift + N】,打开“新建项目”窗口,在左边导航窗格中,找到已安装模板中的“Visual C#”→“Windows”→“经典桌面”节点并选中,然后从窗口中间的模板列表中选择“控制台应用程序”,输入项目名、解决方案名以及存放路径,最后单击“确定”按钮完成项目的创建。

(2) 在 Program 类所在的命名空间下声明一个 A 类。

```
public class A
{
    private int Value { get; set; }
}
```

(3) 然后在 Main 方法中创建 A 类的实例,并尝试访问 Value 属性。

```
A va = new A();
va.Value = 5;      // 错误
```

这时候读者会得到一条错误提示说 Value 属性不可访问,因为它被定义为 private,只能在类内部使用。

(4) 现在再定义一个 B 类,同样也定义一个 Value 属性,但访问修饰符为 public。

```
public class B
{
    public int Value { get; set; }
}
```

(5) 随后在 Main 方法中实例化一个 B 对象,并向其 Value 属性赋值。

```
B vb = new B();
vb.Value = 100;      // 正确
```

由于 B 类的 Value 属性定义为 public,即公共属性,没有访问限制,故在类的外部可以访问。

（6）接下来再看看 internal 关键字的用法，它只允许同一程序集内的代码访问。通常情况下，在使用 Visual Studio 进行开发时，正好一个项目就是一个程序集。所以接下来可以在当前解决方案中再新建一个项目。打开“解决方案资源管理器”窗口，在解决方案节点上右击，并从弹出的快捷菜单中选择【添加】→【新建项目】，在打开的“新建项目”窗口中选择“类库”，然后输入项目的名字，单击“确定”按钮完成。

（7）在新建的项目中用 internal 关键声明一个 M 类。

```
internal class M
{
    public void MakeMessage()
    {
        // 方法内容
    }
}
```

（8）按【Ctrl＋S】快捷键保存，然后在“解决方案资源管理器”窗口中，在前面的控制台应用程序项目的“引用”节点上右击，从弹出的快捷菜单中选择【添加引用】。

（9）在“引用管理器”窗口左侧导航到“解决方案”→“项目”，然后在窗口的中间区域选中刚才新建的类库项目（注意要选上前面的对勾），然后单击“确定”按钮，如图 3-18 所示。

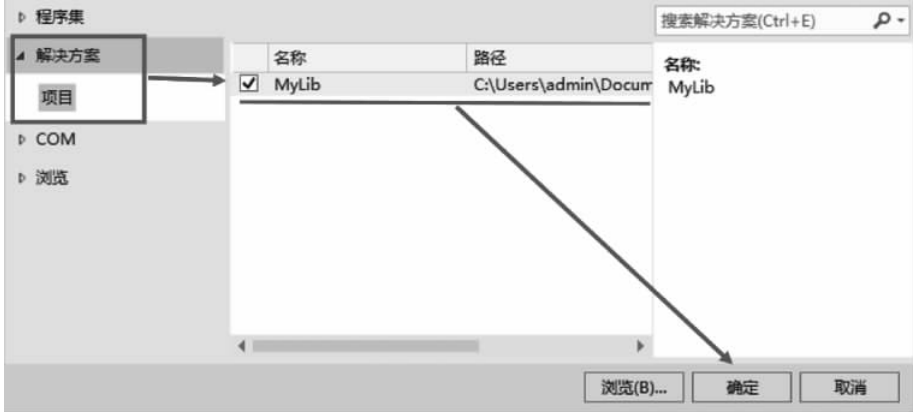


图 3-18　添加引用

（10）同样，我们在 Main 方法中尝试把类库项目中的 M 类进行实例化。

```
MyLib.M m = new MyLib.M();      // 错误
```

这时候就会出现错误提示，因为 M 类声明为 internal，只能在它所在程序集中使用，在其他程序集中无法访问。当然，把 M 类改为 public 就可以访问了，public 是不受限制的。另外，由于在命名空间下类的默认访问方式为 internal，所以在命名空间下直接定义类可以

省略 internal 关键字。

本示例的完整代码位于\第 3 章\Example_6。

3.7.2 继承

通过完成前面的示例,使读者对对象的可访问性有了直观的认识。现在可以开始讨论类的继承问题,因结构是不能派生的,因此继承是对类而言的。

在对客观事物进行抽象提取过程中,人们需要根据客观事物的发展特点做出有层次性的描述。举个例子,星球是对宇宙中各类星体的总概括,具体划分起来,可能会有恒星、行星等,再往下分就会有木星、土星、地球等。然而对于星球这个类来说,它可以囊括所有星体的一些共同特征,比如自转速度、公转速度、直径等。所以,类与类之间可以存在一种层次关系,这种关系就类似于“父子”关系。例如,衣服是一个基类,它可以用尺寸、颜色、布料等特点来描述,但是衣服是可以分为很多种,于是可以从衣服类派生出其他衣服类,如 T 恤、毛衣、运动裤、裙子等。这些类不仅继承了衣服类中定义的属性或其他成员,而且它们可以根据自身的特点来扩展一些新的成员。

下面一起来动手完成一个示例,直观地体会一下如何实现类的继承。在 Visual Studio 开发环境中新建一个控制台应用程序项目。首先定义一个 Person 类,它具有 Name、Address、Age 三个属性。

```
public class Person
{
    /// <summary>
    /// 姓名
    /// </summary>
    public string Name { get; set; }
    /// <summary>
    /// 住址
    /// </summary>
    public string Address { get; set; }
    /// <summary>
    /// 年龄
    /// </summary>
    public int Age { get; set; }
}
```

接着定义一个表示学员信息的 Student 类,从 Person 类派生,并增加一个新的属性 Course,表示学员学习的课程。

```
public class Student : Person
{
    /// <summary>
    /// 课程
```

```
        /// </summary>
        public string Course { get; set; }
    }
```

随后，在代码中创建一个 Student 的实例，并向其属性赋值，然后输出到屏幕上。

```
Student st = new Student();
st.Name = "Tom";
st.Age = 21;
st.Address = "test";
st.Course = "C++编程入门";
// 输出到屏幕
Console.WriteLine("学员信息: \n 姓名: {0}\n 住址: {1}\n 年龄: {2}\n 课程: {3}",
                  st.Name,
                  st.Address,
                  st.Age,
                  st.Course);
```

读者会看到，Student 实例的成员除了新增的 Course 属性外，也保留了 Person 类所定义的几个属性，这也说明，派生类（子类）是在基类（父类）的基础上进行了扩展，其实就是“子承父业”。输出结果如图 3-19 所示。

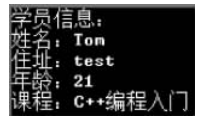


图 3-19　屏幕输出结果

读者可能还会想到一个问题，既然派生类是基类的扩展，那么在创建派生类的实例的时候，调用的是派生类的构造函数还是基类的构造函数？如果都被调用，那么谁先谁后呢？要回答这个问题并不难，只需要修改一下 Person 类和 Student 类定义的代码，分别为它们加上构造函数和析构函数，并使用 Debug 类输出调试信息。

```
public class Person
{
    /// < summary >
    /// 构造函数
    /// </summary>
    public Person()
    {
        System.Diagnostics.Debug.WriteLine("Person 类的构造函数被调用.");
    }
    /// < summary >
    /// 析构函数
    /// </summary>
    ~Person()
    {
        System.Diagnostics.Debug.WriteLine("Person 类的析构函数被调用.");
    }
```

```
…
}

public class Student : Person
{
    /// <summary>
    /// 构造函数
    /// </summary>
    public Student()
    {
        System.Diagnostics.Debug.WriteLine("Student 类的构造函数被调用.");
    }
    /// <summary>
    /// 析构函数
    /// </summary>
    ~Student()
    {
        System.Diagnostics.Debug.WriteLine("Student 类的析构函数被调用.");
    }
…
}
```

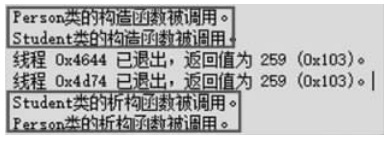


图 3-20　输出的调试信息

现在可以运行一下程序，然后按键盘上的任意一个键将其关闭。打开“输出”窗口(可以在快速启动搜索框中输入“输出”来查找)，读者就会看到输出的调试信息，如图 3-20 所示。

读者会看到，在实例化 Student 类时，先调用基类的构造函数，再调用派生类的构造函数；在实例被释放时，析构函数的调用顺序与构造函数的调用正好相反。

完整的示例代码请参考\第 3 章\Example_7。

3.7.3　注意可访问性要一致

为什么强调派生类的可访问性要与基类保持一致呢？至少派生类的可访问性不要比基类高。请考虑以下代码：

```
internal class A
{
    public void OnTest() { }
}

public class B : A
{
```

```
    public float Value { get; set; }
}
```

乍一看，这段代码并没有问题，B类自身定义了一个Value属性，并从A类继承了一个OnTest方法。然而，当将上述代码进行编译时，就会发现错误。

导致错误的原因是基类和子类的可访问性不一致。那么，为什么要求派生类的可访问性要跟基类一致呢。读者不妨想一下，A类的可访问性是定义为internal的，即只能在当前程序集中访问，而B类从A类派生，并且B类的可访问性是public的，访问不受限制。当A类和B类放在同一个程序集中，似乎不成问题，但是如果要从另一个程序集来访问B类，由于B类是公共的，自然可以创建B类的实例，只是OnTest方法是从A类继承过来的，可A类又声明为internal，这就使得OnTest方法无法被访问。显然这里就出现访问冲突了。

因此，为了保证从基类继承下来的成员都能被有效访问，派生类的可访问性不应该比基类高。派生类的可访问性可以比基类低，因为这样不影响对基类成员的访问，所以如果A类声明为public，B类从A类派生，并声明为internal，是没有问题的，因为它保证了基类的公共成员可以被有效访问。

3.7.4　隐藏基类的成员

先看下面一段代码，A类定义了一个Play方法，可访问性为private，即私有方法；B类从A类派生，也定义了一个Play方法，可访问性为public。

```
public class A
{
    private void Play() { Console.WriteLine("A"); }
}

internal class B : A
{
    public void Play() { Console.WriteLine("B"); }
}
```

如果实例化和调用B类的Play方法，屏幕上输出的是B，因为A类中的Play方法是不会被调用。A类中的Play方法是私有(private)方法，只能在A类内部访问，派生类也无法访问。

现在，把代码改为：

```
public class A
{
    public void Play() { Console.WriteLine("A"); }
}
```

```
internal class B : A
{
    public void Play() { Console.WriteLine("B"); }
}
```

A 类和 B 类的 Play 方法被定义为 public,那么,B 类的实例到底会调用哪个 Play 方法呢?仍然是选择了 B 类的 Play 方法。这里就涉及隐藏基类成员的问题。上面的代码在 B 类中定义了一个和 A 类中相同的 Play 方法,这样就会把 A 类中的 Play 方法给隐藏了,因此在 B 类中就不再调用 A 类中的 Play 方法了。

虽然这样做在编译时不会报错,但会收到警告。这种警告提醒我们,是不是不小心隐藏了基类的成员。有时候代码量较大,开发者在写代码的时候,可能忘了在 A 类中已经写过了 Play 方法,于是在编写 B 类的时候,又重复定义了 Play 方法。所以编译器才会发出警告,是不是有意去隐藏基类的成员。

如果确实是要隐藏基类的成员,就应该明确地告诉编译器,方法是在方法的声明上加一个 new 关键字。这个 new 和用来实例化对象时用的 new 操作符是同一个单词,但在这里它的含义不是创建实例,而是隐藏基类的成员。所以上面的代码可以改为:

```
internal class B : A
{
    public new void Play() { Console.WriteLine("B"); }
}
```

3.7.5 覆写基类成员

在编写派生类的时候,根据当前类的具体需要,可能要用相同的成员来覆盖或者扩展基类的成员。在 3.7.4 节中,已经学习使用 new 关键字来隐藏基类的成员,但会引发一个问题。下面用一个例子就可以说明问题:

```
public class F
{
    public string ThisName
    {
        get { return "F"; }
    }
}
public class G : F
{
    public new string ThisName
    {
        get { return "G"; }
```

```
    }
}
```

F类定义了一个属性ThisName，返回字符串F；G类从F类派生，也定义了一个ThisName属性，并且隐藏了F类的ThisName属性。

接着创建一个G类的实例，并输出其ThisName属性的值。

```
F g = new G();
Console.WriteLine(g.ThisName);
```

最后输出的字符串是F，而不是G。因为变量g被声明为F类型，但是赋值的时候是引用了G类的实例。这里做了一次隐式转换，是允许的，派生类的实例是可以赋值给用基类类型声明的变量的，后面在讲述类型转换的时候还会提到。在这个例子中，我们预期的结果是输出字符串G，但是因为变量g被声明为F类型，尽管它引用了派生类G的实例，变量仍会选择调用基类F的成员。

要解决上述问题，就要使用virtual和override关键字，方法是将基类中需要被覆写的成员加上virtual关键字使其“虚化”，接着把派生类中覆写的成员加上override关键字。

接下来将通过一个实例来演示处理过程(示例代码位于\第3章\Example_8)。下面的代码声明了D类，包含一个公共的Work方法，定义为虚方法(virtual)；E类从D类派生，用override覆写了D类的Work方法。

```
public class D
{
    public virtual void Work()
    {
        Console.WriteLine("调用了D类的Work方法.");
    }
}
public class E : D
{
    public override void Work()
    {
        Console.WriteLine("调用了E类的Work方法.");
    }
}
```

随后，我们分别以D类来声明两个变量，变量d引用D类的实例，变量e引用E类的实例。之后分别用这两个变量调用Work方法。

```
// 分别声明两个D类型的变量
D d, e;
```

```
// 变量 d 引用 D 类的实例
d = new D();
d.Work();                          // 调用 D 类的 Work 方法
// 变量 e 引用 E 类的实例
e = new E();
e.Work();                          // 调用 E 类的 Work 方法
```

虽然变量 d、e 都以 D 类型来声明，但由于 virtual 和 override 关键字协同实现了类型的多态性，运行时库会根据变量所引用的实例类型来判断应该调用谁的 Work 方法。我们预期的结果也达到了，输出结果如图 3-21 所示。

图 3-21 两个 Work 方法的调用结果

override 不仅能覆写基类的成员，还能实现对基类成员的扩展，因为在使用 override 关键字覆写基类成员的同时，也可以使用 base 关键字来调用基类的成员。base 关键字与 this 关键字是相对的，this 关键字引用的是当前类的实例，而 base 关键字引用的是基类的实例。

下面继续完成示例。定义一个 X 类，包含一个 virtual 的 Output 方法，再定义一个 Y 类，从 X 类派生，并且覆写 X 类的 Output 方法，同时也调用基类的 Output 方法。

```
public class X
{
    public virtual void Output()
    {
        Console.WriteLine("调用了 X 类的 Output 方法.");
    }
}
public class Y : X
{
    public override void Output()
    {
        base.Output();                    // 调用基类成员
        Console.WriteLine("调用了 Y 类的 Output 方法.");
    }
}
```

然后使用以下代码进行测试。

```
X y = new Y();
y.Output();
```

由于在 Y 类的 Output 方法中加了 base.Output();一句，使得基类的 Output 方法也被

调用。最终得到如图 3-22 所示的结果。

上一节中提到的隐藏基类成员以及本小节所讲述的成员覆写，构成了面向对象编程中的多态性，应用程序在运行过程中会根据类的继承状态自动识别出应该调用哪些成员。

图 3-22 扩展基类成员的输出结果

3.7.6 如何阻止类被继承

有时候开发者并不希望自己写的类被继承，可以在定义类时加上 sealed 关键字。用 sealed 关键字声明的类也叫密封类。例如下面的代码：

```
public sealed class Room { }
```

Room 被定义为密封类，因此就无法从 Room 类派生。

如果只是想阻止基类中的虚成员被覆写，而并不打算阻止整个类被继承，那么方法与密封类相同，在定义虚成员时加上 sealed 关键字即可。请考虑下面一段代码：

```
public class A
{
    protected virtual void Run() { }
}

public class B : A
{
    protected sealed override void Run()
    {
        base.Run();
    }
}
```

A 类中定义了虚方法 Run，B 类继承 A 类，并覆写 Run 方法，同时使用 sealed 关键字，使得从 B 类派生的类不能再覆写 Run 方法。将成员声明为 protected 只允许当前类和派生类访问，其他外部对象无法访问。

3.8 抽象类

人们通常是把抽象类视为公共基类。抽象类最明显的特征是不能实例化，所以通常抽象类中定义抽象成员，即不提供实现代码。请考虑下面代码，T 类是一个抽象类，它包含一个抽象方法 Check。

```
public abstract class T
{
```

```
        public abstract void Check();
    }
```

通过上面的代码,我们会发现:使用 abstract 关键字表示类或成员是抽象的;抽象方法因为不提供具体的实现,所以没有方法体(一对大括号所包裹的内容),语句以分号结束。抽象类仅对成员进行声明,但不提供实现代码,就等于设计了一个“空架子”,描绘一幅大致的蓝图,具体如何实现取决于派生类。正因为抽象类自身不提供实现,所以不能进行实例化,调用没有实现代码的实例没有实际意义。

接下来,读者可以动手做一个练习,通过这个练习,能够很轻松地掌握抽象类的使用。在 Visual Studio 开发环境中新建一个控制台应用程序项目(完整的示例代码可以参考\第 3 章\Example_9),先定义一个表示所有球类的基类 Ball,该类为抽象类。

```
public abstract class Ball
{
    /// <summary>
    /// 获取球类的名称
    /// </summary>
    public abstract string CateName { get; }
    /// <summary>
    /// 打球
    /// </summary>
    public abstract void Play();
}
```

CateName 属性返回某种球类的名称,如果是足球就返回“足球”,如果是排球就返回“排球”。Play 方法会根据不同的派生类提供不同的实现,如果是足球,就输出“正在踢足球……”。

接下来分别用 FootBall 类和 BasketBall 类来实现抽象类 Ball。读者可以使用 Visual Studio 代码编辑器提供的辅助功能来自动完成抽象类实现。操作方法是当我们输入完抽象类的名字后,在抽象类的类名下方会显示一个智能标记,然后单击该标记,从下拉菜单中选择【实现抽象类“<类名>”】,如图 3-23 所示。

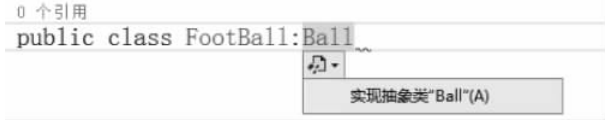


图 3-23 自动完成抽象类的实现

生成代码后,读者会看到,实现抽象类的抽象成员也是使用 override 关键字,与前面提到的成员覆写相似,实现抽象类,也可以看作是覆写基类的成员。随后需要对生成的代码进行修改,最终的实现代码如下:

```
public class FootBall : Ball
{
    public override string CateName
    {
        get { return "足球"; }
    }

    public override void Play()
    {
        Console.WriteLine("正在踢足球……");
    }
}

public class BasketBall : Ball
{
    public override string CateName
    {
        get { return "篮球"; }
    }

    public override void Play()
    {
        Console.WriteLine("正在打篮球…");
    }
}
```

下面是在项目模板自动生成的 Program 类中声明一个 PlayBall 方法，代码如下：

```
static void PlayBall(Ball ball)
{
      Console.WriteLine("\n球类：{0}", ball.CateName);
      ball.Play();
}
```

PlayBall 方法可以体现抽象类的用途，参数 ball 只声明为 Ball 类型，即定义的抽象类，这样的好处在于，不管调用方传递进来的是什么类型的对象，只要是实现了 Ball 抽象类的类型即可。抽象类 Ball 已经规范了派生类肯定存在 CateName 属性和 Play 方法两个成员。显然这种处理方式比较灵活。

最后在 Main 入口点方法中进行调用测试。

```
FootBall football = new FootBall();
BasketBall basketball = new BasketBall();
// 调用 PlayBall 方法
PlayBall(football);
PlayBall(basketball);
```

输出结果如图 3-24 所示。

另外,读者要注意一点,在抽象类中是可以定义实现代码的,即非抽象成员。不妨把上面的 Ball 类修改一下,增加一个 Radius 属性,该属性提供具体的实现。

图 3-24 屏幕输出结果

```
  public abstract class Ball
  {
…
      private decimal _r;
      /// <summary>
      /// 球的半径
      /// </summary>
      public decimal Radius
      {
          get { return _r; }
          set
          {
            if (value <= 0)
            {
              _r = 1;
            }
            if (value > 15)
            {
              _r = 15;
        }
      }
    }
}
```

但是,在非抽象类中不能声明抽象成员,因为非抽象类可以用 new 来实例化,如果存在未实现的抽象成员,代码在调用实例成员时就没有意义了。反过来,在抽象类中定义非抽象成员是允许的,因为抽象类不能用 new 来实例化,而实现抽象类的派生类会继承这些成员,所以代码在调用时,访问的必定是派生类的成员,就不会出现没有意义的代码。

3.9 接口

接口为后续的代码编写与程序开发定制了一个“协定”,也就是一个规范。之后不管代码由谁来编写,都以预先设计好的接口为基准,也可以对项目的后续扩展起到一定的约束作用。接口就好比每个国家都会制定一部宪法,然后其他的法律条文都以宪法为底本来进行补充和深化。

提到接口,很容易让人想到抽象类,因为接口在形式和使用方法上与抽象类很类似。因此,读者不妨根据抽象类的特点来猜测一下接口可能具备以下特点:

(1) 不能被实例化。

(2) 自身不提供实现代码。

第(1)项对于抽象类和接口来说是相同的；第(2)项对于抽象类与接口并不完全相同，可以通过以下两段代码区分出来。

```
public abstract class TestBase
{
    public void DoWork()
    {
        Console.WriteLine("working");
    }
    public abstract int Value { get; set; }
}
```

```
public interface ITest
    {
        void DoWork();
        int Value { get; set; }
}
```

在上面两段代码中，左侧是抽象类的定义，右侧是接口的定义。读者可以仔细观察一下，会找到以下几点差异：

(1) 抽象类的成员定义是带有可访问性关键字的(例如，public)，而接口是不带可访问性关键字的，因为接口中所声明的成员都是公共的，所以没有必要添加访问修饰符。

(2) 抽象类除了包含抽象成员外，还可以包含非抽象成员，也包括构造函数。而接口不能包含具备实现代码的成员，也不能包含构造函数。因为接口是另一种类型，不是类，而抽象类也是类的一种。

通过上面罗列的几个特点可以知道，接口与抽象类确实有着一些共同点。不过，接口的特点并不只是这些，读者将在随后的各小节中逐一去了解。

3.9.1　定义接口

定义接口使用 interface 关键字，如下面代码所示，定义了一个 IBook 接口。

```
public interface IBook { … }
```

在命名接口时，根据习惯，会在前面加上一个大写字母“I”。并不是说一定要这样做，仅仅是一种习惯，.NET 类库中许多接口都是以“I”开头的(取 interface 的首字母)，这种命名方式的好处是便于识别，很多时候，开发者设计好接口，起到了规范作用。而后面实现接口的代码并不一定是由同一位开发者编写，有可能会交给别人实现。如果能做到对接口进行合理命名，那么实现代码的人就可以很方便地识别出接口类型。虽然在 Visual Studio 中会以不同的图标来显示不同类型，但是，笔者还是希望读者在编写接口时能够按照习惯去命名，没有必要去破坏这个习惯。

在默认情况下，接口和类一样，将声明为允许内部访问(internal)，即只能在同一程序集中访问，因此，如果要让接口对外公开，应当加上 public 修饰符。

3.9.2　接口与多继承

在 C++语言中允许多继承，即一个类可以同时继承多个基类。而在 C＃语言中，只允许

单继承,即一个类只能从一个类派生,允许多层次派生,但一个子类不能同时继承多个基类。在C#语言中,以下定义是错误的。

```
public class A { }
public class B { }
public class C : A, B { } // 错误
```

C类可以从A类派生,或者从B类派生,但不能同时派生自A、B两个类。

但是,一个类是可以实现多个接口,这在形式上达到了多继承的效果。比如下面代码:

```
public interface IA { }
public interface IB { }
public class C : IA, IB { }
```

在上面的代码中,C类同时实现了IA,IB两个接口,多个接口间用逗号(英文)分隔。

3.9.3 实现接口

由于接口中定义的成员都是公共成员,因此在实现接口时,无论是结构(结构可以实现接口)还是类,必须以公共成员来实现接口的成员,而且必须实现接口的所有成员。请考虑下面代码:

```
public interface IX
{
    int Num { get; }
    void Work();
}

public class Z : IX
{
    // 错误: 成员没有声明为 public
    protected int Num { get; set; }
    public void Work() { }
}

public class Y : IX
{
    public int Num
    {
        get { return 0; }
    }
    // 错误: 未实现 Work 方法
}
```

在上述代码中，IX 定义了一个属性和一个方法。Z 类的错误是没有将 Num 属性声明为 public，因为接口中定义的都是公共成员，因此在实现接口时也要将成员声明为公共成员。对于属性中的 get 和 set 访问器没有严格的要求，不要求与接口中定义的一致。

Y 类的错误是没有完全实现 IX 接口的成员，只实现了 Num 属性，还有 Work 方法没有实现。

接口也可以继承接口，如果 B 接口继承了 A 接口，那么 B 接口也包含 A 接口中定义的成员，这一点与类的继承相似。请看下面的代码：

```
public interface IA
{
    void DoA();
}

public interface IB : IA
{
    string Name { get; set; }
}

public class Test : IB
{
    public string Name { get; set; }
    public void DoA() { }
}
```

DoA 方法是在 IA 接口定义的，Name 属性是在 IB 接口中定义的。Test 类实现 IB 接口，而 IB 接口继承 IA 接口，因此 IB 接口包括 Name 属性和 DoA 方法。故 Test 类实际上是实现了两个成员——Name 属性和 DoA 方法。

3.9.4　显式实现接口

显式实现接口为了解决接口成员冲突的问题，这种冲突主要表现在成员名称上会出现重复。请考虑下面代码：

```
public interface IA
{
    void Speak();
}
public interface IB
{
    void Speak();
}
public class Test : IA, IB
{
```

```
        public void Speak() { }
    }
```

由于IA接口和IB接口所定义的成员相同,Test类在实现IA和IB两个接口时,也只能实现一个Speak方法。读者可能已经发现,IA和IB接口不就等于一样吗?这样声明没有意义。如果仅从代码层面上说确实没有实际意义。不过,从思路上看,有时候是需要定义不同的接口,但是不能排除多个接口之间会定义重复的成员。例如上面的例子,假设两个接口的Speak方法代表不同含义或不同用途的成员,开发人员希望两个Speak方法实现不同的功能,当然也可以用不同的类分别实现IA和IB接口。可是,如果确实需要用一个类同时实现两个接口,而又要保证两个Speak方法都能实现,有效地解决方法是使用显式实现接口。

所谓显式实现接口,就是在类中实现的接口成员前面加上接口的名字,以说明该成员来自哪个接口。下面会通过一个实例来演示如何显式实现接口。

在Visual Studio开发环境中新建一个控制台应用程序项目。定义两个接口ITest1和ITest2,它们都有一个同名方法Run。

```
public interface ITest1
{
    void Run();
}
public interface ITest2
{
    void Run();
}
```

接着,定义一个Test类,显式实现这两个接口。读者可以使用Visual Studio的自动生成代码的功能来完成。如图3-25所示,在写完Test类实现ITest1和ITest2接口的代码后,在ITest1和ITest2下方都会出现一个智能标记,分别单击智能标记,并从菜单中选择【显式实现接口】命令。

图3-25 显式实现接口

实现接口的代码如下：

```
public class Test : ITest1, ITest2
{
    void ITest1.Run()
    {
        Console.WriteLine("调用 ITest1.Run 方法.");
    }
    void ITest2.Run()
    {
        Console.WriteLine("调用 ITest2.Run 方法.");
    }
}
```

读者会看到，显式实现接口的成员，在名字前面要加上所属接口的名字以及一个成员运算符(.)。

接下来看看如何使用 Test 类。显式实现了接口的类无法通过类的实例来调用成员，只能通过对应的接口来调用。也就是说，Test 类的实例无法调用 Run 方法。可能有些读者朋友会想到通过隐式转换的方法，既然 Test 类实现了 ITest1 和 ITest2 接口，那么分别声明 ITest1 和 ITest2 类型的变量，再赋以 Test 类的实例，就能调用了。代码如下：

```
ITest1 t1 = new Test();
ITest2 t2 = new Test();
t1.Run();
t2.Run();
```

这样确实可以分别调用 Run 方法，但是有一个问题：上面代码中其实是创建了两个 Test 实例，这两个实例是相互独立的，如果两个 Run 方法之间涉及类中的一些数据，例如私有字段等，显然两个 Run 方法不是调用自同一个实例，这样会造成数据的不统一。

因此，正确的调用方法是先创建 Test 类的实例，然后把这个实例分别转换为对应的接口类型来调用特定的 Run 方法。具体代码如下：

```
Test t = new Test();
// 调用 ITest1.Run 方法
((ITest1)t).Run();
// 调用 ITest2.Run 方法
((ITest2)t).Run();
```

最后输出的内容如图 3-26 所示。

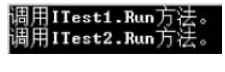


图 3-26　输出结果

有关本示例的完整代码请参考\第 3 章\Example_10。

3.10 扩展方法

扩展方法是一种比较有趣的方法,它可以在不继承现有类型的前提下扩展类型。扩展方法可以合并到要扩展类型的实例上。因此,扩展方法要定义为静态方法,并且第一个参数必须为要扩展类型的当前实例(参数前面要加上 this 关键字)。

扩展方法使用起来并不复杂,在定义扩展方法时,通常要先定义一个类,由于扩展方法都是静态方法,所以可直接声明一个静态类。然后将扩展方法包含在该类中就可以了。

下面的示例将扩展 .NET 类库中现有的 System.String 类,把字符串中的各个字符用两个空格来分隔,例如字符串为“jack”,调用扩展方法后就变为“j a c k”。

首先定义扩展方法,代码如下:

```
public static class StringExt
{
    public static string SplitBySpace(this string str)
    {
        string strRes;
        // 将字符串转化为字符数组
        char[] cs = str.ToCharArray();
        // 调用 Join 方法将字符重新串联起来
        strRes = string.Join(" ", cs);
        return strRes;
    }
}
```

现在,可以通过 String 类的实例调用该扩展方法。

```
string s = "abcdefg";
// 调用 SplitBySpace 扩展方法
Console.WriteLine(s.SplitBySpace());
```

于是,得到如图 3-27 所示的结果。

图 3-27 输出的每个字符间都带有空格

有关本示例的完整代码请参考\第 3 章\Example_11。

3.11 委托与事件

委托是一种在形式上与方法签名相似的类型。委托实例化后可以与方法关联,在调用委托实例的同时会调用与之关联的方法。这使得代码可以把方法当作参数来传递给其他方法。从运行方式来看,委托与 C 语言中的函数指针有些类似。

3.11.1 定义和使用委托

委托的声明和声明方法相似，不过要使用 delegate 关键字，以告诉编译器这是委托类型。由于它是一种类型，所以是可以独立声明的。

使用委托时要先实例化，和类一样，使用 new 关键字来产生委托的新实例，然后将一个或多个与委托签名匹配的方法与委托实例关联。随后调用委托时，就会调用所有与该委托实例关联的方法。与委托关联的可以是任何类或结构中的方法，也可以是静态方法，只要是可以访问的方法都可以。

例如，下面代码定义了一个 DoSome 委托：

```
public delegate void DoSome(string msg);
```

然后，需要分析哪些方法可以与 DoSome 委托匹配。该委托能匹配的方法必须是返回 void 类型（不返回任何内容），而且接受一个 string 类型的参数。不妨看几个例子，下面的 TestDo 方法是匹配的。

```
static void TestDo(string str) { }
```

但是，下面的 Test2 方法就不能匹配，因为它的参数不是 string 类型。

```
public void Test2(int n) { }
```

下面的 WorkAs 方法也不匹配，因为它没有参数，而且有返回值。

```
public float WorkAs() { }
```

再看下面的 ToDone 方法，虽然它返回 void，参数也是 string 类型，但是它有两个参数，而 DoSome 委托只有一个参数，所以也不匹配。

```
private void ToDone(string str1, string str2) { }
```

前文曾说过，任何数据类型都会隐含有公共基类的，因为任何类型都是从 System.Object 类派生。而委托类型隐含的公共基类是 System.Delegate 或者 System.MulticastDelegate 类，后者实现了委托的多路广播，即一个委托类型的实例可以与多个方法关联。在实际使用中，.NET 框架所支持的每种编程语言都会实现与委托类型相关的关键字，在 C＃语言中为 delegate。编译器会自动完成从 System.Delegate 或者 System.MulticastDelegate 类的隐式继承，开发者不能自己编写代码来继承这些类型，只能在代码中使用 delegate 关键字来声明委托类型，剩下的工作将由编译器来完成。

在实例化委托时，可以将要关联的方法作为参数来传递，例如，使用上面举例中的

DoSome 委托来跟与之匹配的 TestDo 方法进行关联，就可以这样来实例化：

```
DoSome d = new DoSome(TestDo);
```

还可以使用更简洁的方法来实例化委托，即直接把与委托匹配的方法赋值给委托类型的变量。

```
DoSome d = TestDo;
```

调用委托时，与调用普通方法相似，有参数就传递参数，有返回值就接收返回值。例如：

```
d("abc");
```

委托之间可以进行相加和相减运算，但这与数学中的加减运算不同，委托的加运算可以增加所关联的方法，而减法则是从委托所关联的方法列表中移除指定的方法。例如：

```
d += new DoSome(TestRun);
```

接下来，读者可以完成一个示例，亲自去感受一下委托的妙用。首先在 Visual Studio 开发环境中新建一个控制台应用程序项目(完整的代码可以参考\第 3 章\Example_12)。

接着，在项目生成的 Program 类中定义三个静态方法，这三个方法必须签名相同，以便稍后使用。

```
static void TestMethod1(string str)
{
      Console.WriteLine("这是方法一.参数: {0}", str);
    }
    static void TestMethod2(string str)
    {
      Console.WriteLine("这是方法二.参数: {0}", str);
    }
    static void TestMethod3(string str)
    {
      Console.WriteLine("这是方法三.参数: {0}", str);
    }
```

三个方法都带一个 string 类型的参数，并且在方法体中向屏幕输出相关文本和参数。

随后，定义委托类型。

```
public delegate void MyDelegate(string s);
```

紧接着，创建三个 MyDelegate 实例，分别与上面三个方法关联，并逐个进行调用。

```
// 定义三个委托变量
MyDelegate d1, d2, d3;
// d1 关联 TestMethod1 方法
d1 = TestMethod1;
// d2 关联 TestMethod2 方法
d2 = TestMethod2;
// d3 关联 TestMethod3 方法
d3 = TestMethod3;
// 分别调用三个委托实例
Console.WriteLine("分别调用三个委托实例,输出结果如下: ");
d1("d1");
d2("d2");
d3("d3");
```

屏幕输出结果如图 3-28 所示。

接下来,再创建一个 MyDelegate 委托实例 d4,并且与三个方法关联,在调用 d4 时就能同时调用这三个方法。

```
// 先与 TestMethod1 方法关联
MyDelegate d4 = TestMethod1;
// 随后再与 TestMethod2 和 TestMethod3 方法关联
d4 += TestMethod2;
d4 += TestMethod3;
// 调用 d4
Console.WriteLine("\n 调用 d4 可同时调用三个方法,结果如下: ");
d4("d4");
```

d4 在实例化时与 TestMethod1 方法进行了关联,而后通过相加运算,又与 TestMethod2 和 TestMethod3 方法进行关联。也就是说,d4 是多播委托,它同时与多个方法关联,调用该委托实例可同时调用多个方法,输出结果如图 3-29 所示。

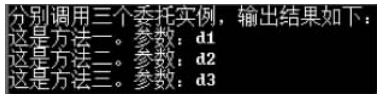


图 3-28　调用三个委托实例的输出结果

图 3-29　同时调用多个方法

现在,把 TestMethod2 方法从 d4 关联的方法列表中移除,并再次调用 d4。

```
// 从 d4 中关联的方法列表中减去 TestMethod2 方法
d4 -= TestMethod2;
// 再次调用 d4
Console.WriteLine("\n 移除与 TestMethod2 方法关联后: ");
d4("d4");
```

结果如图 3-30 所示,我们看到 TestMethod2 方法就不再被调用了。

图 3-30 TestMethod2 方法移除后 d4 的调用结果

3.11.2 将方法作为参数传递

委托可以让方法作为参数传递给其他方法。可以用一个示例就能阐述这一问题了。完整的示例位于\第 3 章\Example_13。首先定义一个委托类型,代码如下:

```
public delegate void MyDelegate();
```

随后在项目生成的 Program 类中定义两个方法 M1 和 M2,为什么需要两个方法呢?因为本例稍后会顺便验证委托的传值方式。

```
static void M1() { Console.WriteLine("方法一"); }
static void M2() { Console.WriteLine("方法二"); }
```

然后再定义一个 Test 方法:

```
static void Test(MyDelegate d)
{
    // 调用委托
    if (d != null)
    {
        d();
    }
    // 改为与 M2 方法关联
    d = M2;
}
```

在方法体中调用委托,随后又将参数 d 与 M2 方法关联。现在来进行测试调用:

```
MyDelegate de = M1;
Test(de);
// 执行 Test 方法后重新调用委托
de();
```

声明委托变量 de 并与 M1 方法关联,然后调用 Test 方法,在调用完 Test 方法后再调用一次委托变量 de。最终得到如图 3-31 所示的结果。

图 3-31 输出结果

在 Test 方法中代码修改了参数 d，与 M2 方法进行了关联，但是，当方法执行完成后，在方法外再次调用 de，输出的仍然是“方法一”。因此，本示例不仅演示了如何通过委托实现将方法作为参数传递，同时也说明了委托类型在传递时是进行自我复制的。参数 d 在方法内部被修改，但不影响方法外部的 de 变量。虽然委托是引用类型，但是在方法内部让委托变量与 M2 方法进行了关联，就等于参数 d 引用了新的委托实例。而外部的委托变量传递给参数 d 时只是把委托实例的地址进行了复制，所以方法调用完成后，外部的变量所引用的仍然是原来的委托实例。

3.11.3　使用事件

事件与委托有着密切的关系，因为事件自身就是委托类型。由于委托可以绑定和调用多个方法，所以会为事件的处理带来方便。类型只需对外公开事件，就可以与外部的其他方法关联，从而实现事件订阅。

对于事件订阅，许多初学者往往不理解。其实，在学习编程过程中，不少概念是可以转移到现实客观事物上去理解的。假设我们订阅了某新闻平台的邮件通知服务，只要有新闻更新，服务提供方就要发送通知邮件。因此，事件订阅也是如此，前文分析过选用委托作为事件的类型的理由，就是委托可以与其他方法关联，当 A 方法与 X 事件进行了关联，只要 X 事件发生（相当于新闻内容有更新），就会调用作为事件的委托（相当于服务提供方发送通知邮件），因为 A 方法与 X 事件关联，所以 A 方法也会被调用，于是代码就能够响应事件。

本书在前面还举过一个例子，响应事件就好比学生一听到下课铃响了，就知道要下课了，或者准备去吃饭了。学生的反应就相当于处理事件的方法，而下课铃声响起就相当于表示事件的委托被调用。

读者已经知道，事件是委托类型，因此，要在类中声明事件，首先要定义用来作为事件封装类型的委托，然后在类中用 event 关键字来声明事件。为了允许派生类重写引发事件的代码，通常会在类中声明一个受保护的方法，习惯上命名为 On＜事件名＞，然后在这个方法中调用事件。读者会看到在 .NET 类库中许多类型都采用这种封装形式。

图 3-32　响应事件的输出信息

示例\第 3 章\Example_14：本示例将演示事件的使用方法。该示例运行后，只要用户按下空格键，就会在屏幕中给出提示，如果按下其他键，不做处理。运行结果如图 3-32 所示。

首先，定义一个委托类型，作为响应按下空格键这一事件的封装类型。

```
public delegate void SpaceKeyPressedEventHandler();
```

定义一个 MyApp 类，代码如下：

```
public class MyApp
{
```

```
        /// <summary>
        /// 声明事件
        /// </summary>
        public event SpaceKeyPressedEventHandler SpaceKeyPressed;
        /// <summary>
        /// 通过该方法引发事件
        /// </summary>
        protected virtual void OnSpaceKeyPressed()
        {
          if (this.SpaceKeyPressed != null)
          {
                SpaceKeyPressed();
          }
          }
          public void StartRun()
          {
              while (true)
              {
                  ConsoleKeyInfo keyinfo = Console.ReadKey();
                  if (keyinfo.Key == ConsoleKey.Spacebar)
                  {
                      // 引发事件
                      OnSpaceKeyPressed();
                  }
                  if (keyinfo.Key == ConsoleKey.Escape)
                  {
                      // 跳出循环
                      break;
                  }
             }
        }
    }
```

在类中，用刚才定义的委托声明了 SpaceKeyPressed 事件。然后封装在 OnSpaceKeyPressed 方法中调用，因为这个表示事件的委托有可能是空的，调用方可能不响应事件处理，即没有与之关联的方法。所以，我们在调用前必须先判断一下表示事件的委托实例是否为 null。

还有一种更简便的写法，不需要写 if 语句进行判断，而是直接调用事件，但在事件名称后面加上一个“?”(英文的问号)，代码如下：

```
protected virtual void OnSpaceKeyPressed()
{
    this.SpaceKeyPressed?.Invoke();
}
```

加上“?”符号之后，程序会自动判断 SpaceKeyPressed 是否为 null，如果为 null，这行代

码就不会执行，直接跳过；如果不为 null 就执行这行代码。

在 StartRun 方法中启动一个无限循环，因为 while 后面的判断条件是 true，true 永远都是 true，也就成了一个死循环，所以需要用下面这段代码来跳出循环：

```
if (keyinfo.Key == ConsoleKey.Escape)
{
      // 跳出循环
      break;
}
```

当被按下的键是 Esc 时，用 break 语句直接退出循环。

如果按下的键是空格键，就调用 OnSpaceKeyPressed 方法，这样一来，事件就被触发了。而代码在使用 MyApp 类时，先新建一个实例，然后通过 += 运算符使 SpaceKeyPressed 事件与相关的方法关联起来。在 Visual Studio 中，只要在事件名后面输入+=，再按一下 Tab 键，这时候会生成一个默认的方法名（变量名_方法名），因为这个方法名处于选定状态，我们可以对其进行重命名（如图 3-33 所示），如果不需要重命名，就再按一下 Tab 键，就会生成事件处理方法。

图 3-33　生成事件处理代码

方法中的处理代码如下：

```
static void app_SpaceKeyPressed()
{
      Console.WriteLine("{0} 按下空格键.", DateTime.Now.ToLongTimeString());
}
```

Main 方法中的代码如下：

```
static void Main(string[] args)
{
      MyApp app = new MyApp();
      // 关联事件处理方法
      app.SpaceKeyPressed += app_SpaceKeyPressed;
      app.StartRun(); // 开始运行
}
```

运行应用程序后，只要按下空格键，屏幕上就会输出提示信息。

在引发事件时，许多时候都会考虑传递一些数据。比如一个捕捉鼠标操作的事件，光是

引发事件是不够的,事件的处理程序还需要知道用户进行了哪些鼠标操作,是移动了鼠标指针,或是按下了鼠标上的某个键;如果是按下了鼠标上的键,是左键还是右键……可见,在引发事件的时候,还有必要传递一些描述性的信息,以便事件处理代码能够获得更详细的数据。

通常,作为事件委托,有两个参数,一个是 Object 类型,表示引发事件的对象,即是谁引发了事件的,多数情况下在调用事件时是把类的当前实例引用(this)传递过去。另一个参数是从 System. EventArgs 派生的类的实例。这是一个标准的事件处理程序的签名,为了规范事件的处理,.NET 类库已经定义好一个 System. EventHandler 委托,用于声明事件。它的原型如下:

```
public delegate void EventHandler(object sender, System.EventArgs e);
```

引发事件的对象实例将传递给 sender 参数,而与事件相关的数据则传递给 e 参数。如果不需要传递过多的数据,可以通过 System. EventArgs. Empty 静态成员返回一个空的 EventArgs 对象来传递。

但是,由于不同的事件要传递的参数不同,更多时候是从 EventArgs 类派生的子类的实例,显然一个 EventHandler 委托是不能满足各种情况的。如果针对不同的事件也定义一个对应的委托,数量一旦多起来,既混乱,也不好管理。为了解决这个问题,.NET 类库又提供了一个带有泛型参数的事件处理委托。原型如下:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

TEventArgs 是一个泛型参数,后面在讲述泛型相关的知识时会提到。但是,TEventArgs 应该是 System. EventArgs 类或者 System. EventArgs 类的派生类型。

图 3-34 响应事件处理结果

有了 EventHandler<TEventArgs>委托,开发者就可以应对各种各样的事件了。因为对于不同的事件,第一个参数是不变的,只是第二个参数的类型有差异而已。

请看下面的示例,该示例捕捉用户的键盘输入,然后引发 KeyPressed 事件,在事件参数中传递用户按下的键。运行结果如图 3-34 所示。

示例大致的实现步骤如下:

(1) 在 Visual Studio 开发环境中新建一个控制台应用程序项目。

(2) 定义一个 KeyPressedEventArgs 类,用来存放事件参数,PressedKey 属性表示用户按下的键。

```
public class KeyPressedEventArgs : EventArgs
{
```

```
    public KeyPressedEventArgs(ConsoleKey key)
    {
      PressedKey = key;
    }
    public ConsoleKey PressedKey { get; private set; }
}
```

(3) 定义 MyApp 类,代码如下:

```
public class MyApp
{
    // 捕捉按键的事件
    public event EventHandler<KeyPressedEventArgs> KeyPressed;

    // 通过该方法引发事件
    protected virtual void OnKeyPressed(KeyPressedEventArgs e)
    {
        if (this.KeyPressed != null)
        {
            this.KeyPressed(this, e);
        }
    }

public void Start()
{
    while (true)
    {
        ConsoleKeyInfo keyinfo = Console.ReadKey();
        // 如果按下了 Esc 键,则退出循环
        if (keyinfo.Key == ConsoleKey.Escape)
        {
            break;
        }
        // 引发事件
        OnKeyPressed(new KeyPressedEventArgs(keyinfo.Key));
        }
    }
}
```

使用 EventHandler<KeyPressedEventArgs>委托声明 KeyPressed 事件,并通过 OnKeyPressed 方法来引发事件。在 Start 方法中,通过一个无限循环来捕捉按键输入,并引发 KeyPressed 事件。

(4) 在 Main 方法中实例化 MyApp 类,并关联 KeyPressed 事件的处理方法。随后调用 Start 方法开启循环。

```
static void Main(string[] args)
    {
        MyApp app = new MyApp();
        app.KeyPressed += app_KeyPressed;
        app.Start();
    }

// 响应处理事件
static void app_KeyPressed(object sender, KeyPressedEventArgs e)
{
    Console.WriteLine("已按下了{0}键.", e.PressedKey.ToString());
}
```

完整的示例代码请参考\第 3 章\Example_15。

3.12 枚举

枚举可以认为是一种由多个整数常量组成的类型。枚举中的每个成员都必须是整数(不包括 char 类型)。因此,枚举的基础类型包括:byte、sbyte、short、ushort、int、uint、long、ulong。非整数的值类型,如 double 是不允许作为枚举的基础类型的。枚举的默认基础类型是 int 类型。

尽管枚举类型的结构比较简单,只是一系列数值的组合,但是使用枚举类型有两个显著的优点:

(1) 严格规范性,防止意外调用。例如,一周有七天,星期日到星期六,某个方法需要传递一个表示一周中某一天的参数,如果使用单个整数值,很难进行规范,程序代码无法事前预知方法的调用方会传递哪个数值作为参数,这会让参数的合法性验证变得十分困难。但是,如果定义了枚举类型作为参数,调用方在传递参数时只能从枚举类型所声明的值中进行选择,就不会出现意外的值。

(2) 增强可读性。枚举中的每个值都可以进行命名,代码调用方通过这些名称就能够轻松地推测各个值所指代的含义。例如,一个表示电源开关状态的枚举,假设用 0 表示关闭,用 1 表示打开,并且命名为 On = 1,Off = 0。代码的调用者只要看到 On 便知道是表示打开的状态了。

枚举类型默认继承自 Enum 类(由编译器实现),而 Enum 类从 ValueType 类派生。据此可以得知,枚举类型属于值类型。

3.12.1 使用枚举类型

声明枚举类型使用 enum 关键字,内部各个常数用英文逗号隔开。例如:

```
enum Test { a, b, c}
```

上面的代码声明了一个 Test 枚举，它包含三个成员 a、b、c。由于使用了默认方式来定义，所以 Test 枚举的基础类型是 int，而其中的常数值是从 0 开始进行排列，即 a 的值为 0，b 的值为 1，c 的值为 2。读者可以用下面代码来验证：

```
Console.WriteLine("a 的值为：{0}\nb 的值为：{1}\nc 的值为：{2}", (int)Test.a, (int)Test.b,
(int)Test.c);
```

结果输出如图 3-35 所示的文本信息。

图 3-35　验证枚举的值

读者可以自定义枚举中各常数的值，方法和赋值一样，例如：

```
enum Test { a = 3, b, c}
```

这时候 a 的值为 3，b 和 c 没指定具体的值，就以 a 的值为基础，累加 1，所以 b 的值为 4，c 的值为 5。例如：

```
enum Test { a, b = 10, c}
```

这一次只为 b 赋了具体的值，对于 a 来说，还是默认值 0，而 b 已赋值 10，c 的值以 b 的值为基础累加 1，所以 c 的值应为 11。

当然也可以向所有命名成员赋值，例如：

```
enum Test { a = 9, b = 25, c = 0x00EC}
```

由于 Test 枚举中所有成员都赋了值，所以此时 a、b、c 的值就不是连续的整数了，a 的值为 9，b 的值为 25，c 的值是 236（只不过用十六进制来表示而已）。

要将枚举声明为 int 以外的整数类型，需要在枚举的类型名称后面加上英文冒号，紧接着是目标类型。例如：

```
enum Mode : byte
{
    None = 0,
    Option = 30,
    Save = 5
}
```

上面的代码定义了一个 Mode 枚举，该枚举基于 byte 类型，因此其中的各个常数都是字节（byte）类型。

不知道读者是否会有这样的疑问：既然枚举是值类型，而且其基础类型与各整数类型有关，那么程序在运行时为枚举所分配的存储空间大小会与它的基础类型相等吗？我们知道，一个 byte 类型的值占 1 个字节的存储空间，一个 int 类型的值占 4 个字节的存储空间。

那么,相应的枚举的值又会占用多少字节的存储空间呢?

先来完成一个示例程序,通过该例子,就可以直观地回答上面的疑问了。请读者在Visual Studio开发环境中新建一个控制台应用程序项目。

首先,为每种基础类型声明一个枚举。

```
// int 类型
enum intEnum { V1, V2 }

// byte 类型
enum byteEnum : byte { V1, V2, V3 = 20 }

// sbyte 类型
enum sbyteEnum : sbyte { B1, B2 }

// short 类型
enum shortEnum : short { S1 }

// ushort 类型
enum ushortEnum : ushort { Q1, Q2 }

// uint 类型
enum uintEnum : uint { I1, I2 }

// long 类型
enum longEnum : long { L1, L2, L3 }

// ulong 类型
enum ulongEnum : ulong { U }
```

随后,在Main方法中使用sizeof运算符来获取并输出每个枚举的值的大小,以字节为单位。

```
Console.WriteLine("int 类型的枚举的大小为{0}个字节.", sizeof(intEnum));
Console.WriteLine("byte 类型的枚举的大小为{0}个字节.", sizeof(byteEnum));
Console.WriteLine("sbyte 类型的枚举的大小为{0}个字节.", sizeof(sbyteEnum));
Console.WriteLine("short 类型的枚举的大小为{0}个字节.", sizeof(shortEnum));
Console.WriteLine("ushort 类型的枚举的大小为{0}个字节.", sizeof(ushortEnum));
Console.WriteLine("uint 类型的枚举的大小为{0}个字节.", sizeof(uintEnum));
Console.WriteLine("long 类型的枚举的大小为{0}个字节.", sizeof(longEnum));
Console.WriteLine("ulong 类型的枚举的大小为{0}个字节.", sizeof(ulongEnum));
```

sizeof运算符可以返回一个对象所占用的内存空间大小,计算单位为字节。运行本程序后,会得到如图3-36所示的结果。

从输出的结果中可以会看到，枚举值的大小与其基础类型的大小相等。如果一个 int 类型的值大小为 4 个字节，则 intEnum 枚举的值的大小也是 4 个字节；再如，byte 类型的值大小为 1 个字节，则 byteEnum 枚举的值的大小也是 1 个字节。尽管枚举内部定义了一组数值，但是在同一时刻只能向枚举类型的变量赋其中一个值。即使是按位运算后的枚举也是如此，因为按位运算后，基本类型不会改变，故枚举的值的大小也不会改变。

图 3-36　输出各个枚举的大小

完整的示例代码请参考\第 3 章\Example_16。

3.12.2　如何获取枚举的值列表

由于枚举类型在编译时默认以 Enum 类为基类，因此 Enum 类的成员对枚举类型是有效的。通过调用一个名为 GetValues 的静态方法，将指定枚举类型中的所有成员的值列表以数组的形式返回。如果枚举类型在定义时以 byte 为基础类型，则返回 byte 类型的数组；如果枚举是以 uint 类型为基础类型定义的，则返回 uint 类型的数组。

下面用一个示例来演示如何获取指定枚举类型中的值列表。完整的程序代码可以参考\第 3 章\Example_17。首先定义一个枚举类型，代码如下：

```
enum Test : ushort
{
    Value1 = 100,
    Value2 = 101,
    Value3 = 103
}
```

Test 枚举是基于 ushort 类型，里面定义了三个值。下面就用 Enum.GetValues 方法把这些值都取出来，并通过 foreach 循环将它们逐一输出到屏幕上。

```
var values = Enum.GetValues(typeof(Test));
// 输出这些枚举值
foreach (ushort v in values)
{
      Console.Write(v + "\t");
}
```

GetValues 是静态方法，因此可以直接调用。GetValues 方法的原型如下：

```
public static Array GetValues(Type enumType);
```

参数是一个 Type 对象，Type 也是一个类，它包装与类型相关的信息。通过这个参数

告诉 GetValues 方法,代码希望获取哪个枚举类型的值列表。返回值为 Array 类型,即数组的基类,由于枚举可能基于 byte 类型,也可能基于 int,返回什么类型的数组取决于枚举的数值的类型。将返回类型定义为 Array,可以兼容各种类型的数组。不管返回的是 byte 类型的数组还是 uint 类型的数组,都是 Array 的派生类,都是允许的。这里也体现了抽象类的一个作用:能够在运行阶段动态引用派生类的实例。

由于 GetValues 方法返回的数组类型不是固定的,而是动态的,因此可以考虑使用 var 关键字来声明变量,如上面的:

```
var values = Enum.GetValues(…);
```

用 var 关键来声明变量不必指定变量的具体类型,而是根据给变量赋的值来推断变量的类型。比如,下面的代码中,变量 c 的类型为字符串类型(string)。

```
var c = "xyz";
```

本示例的运行结果如图 3-37 所示。

```
100     101     103
```

图 3-37 获取到的枚举类型的值

3.12.3 如何取得枚举中各成员的名字

Enum 类有两个静态方法可以获取一个枚举类型中各数值的名字。第一个是 GetName 方法,它的原型如下:

```
public static string GetName(
    Type enumType,                  // 要获取名称的枚举的类型
    Object value                    // 具体的枚举值
)
```

该方法获取单个枚举值的名称。另外,如果希望获取一个枚举类型中所有常数值的名称,应当改用下面的静态方法:

```
public static string[] GetNames(
    Type enumType                  // 枚举类型的 Type
)
```

这两个方法使用起来也比较简单,因为是静态方法,所以直接调用即可。接下来用一个示例来演示这两个方法的用法。该示例用 .NET 类库中的 System.DayOfWeek 枚举来做测试,先通过 GetName 方法获取常数 Thursday 的名字,接着使用 GetNames 方法得到 DayOfWeek 枚举中所有常数的名字。代码如下:

```
// 获取值 DayOfWeek.Thursday 的名称
Console.Write("常数 DayOfWeek.Thursday 的名字: {0}\n\n", Enum.GetName(typeof(DayOfWeek),
DayOfWeek.Thursday));

// 获取 DayOfWeek 枚举中所有常数值的名字
string[] names = Enum.GetNames(typeof(DayOfWeek));
// 输出各值的名字
foreach (string n in names)
{
    Console.Write("{0} ", n);
}
```

GetNames 方法返回一个字符串数组，该数组包含了枚举中每个值所对应的名字。输出结果如图 3-38 所示。

图 3-38　将获取到的名字输出到屏幕

完整的示例源代码请参考\第 3 章\Example_18。

3.12.4　枚举的位运算

既然枚举类型是以整数值为基础的(无论是 int 还是其他整数类型)，故可以对枚举中的各个值进行位运算，比如按位“与”、按位“或”等运算。

通常，如果考虑让枚举中的值进行位运算，应当在定义枚举类型时附加上 FlagsAttribute 特性，有关特性的使用，会在下一节中介绍，这里读者只需记住这一点就行了。

还是用例子来演示吧！完整的示例可以参考随书源码中的\第 3 章\Example_19。首先，定义一个 Test 枚举，记得要加上 FlagsAttribute 特性。

```
[Flags]
enum Test
{
    /// <summary>
    /// 二进制 0
    /// </summary>
    None = 0,
    /// <summary>
    /// 二进制 1
    /// </summary>
    Music = 1,
```

```
        /// <summary>
        /// 二进制 10
        /// </summary>
        Video = 2,
        /// <summary>
        /// 二进制 100
        /// </summary>
        Text = 4
    }
```

Test 枚举包含四个常数,换算成二进制分别为 0、1、10、100。这样声明可以方便读者理解其中是如何进行位运算的,例如,要将 Music 和 Video 进行或运算,其计算过程如下:

```
01 | 10 = 11
```

因为对于或运算,只要其中有一个值为 1,其结果就为 1,所以通常会通过或运算来对多个枚举值进行合并,我们把上面的 Test 枚举看作一个三位数的二进制数,如果枚举值中包含 Music,则第一位为 1,合并后的值为 001;如果用这个值再与 Video 进行合并,得到的值就是 011;接着再与 Text 组合,最终的值就会变为 111。

那么,如何去判断一个经过组合后的枚举变量是否包含指定的值呢?这就要用到与运算了,原理就是在与运算中,必须两个操作数同时为 1 时,计算结果才会为 1。利用这一点,如果要判断上面的组合值 111 中是否包含 Video,那么可以这样运算:

```
判断 111 & 010 是否等于 010
```

把 111 与 010 进行与运算,第一位和第三位都为 0,所以最终的结果就取决于第二位了,如果为 1 就表明组合的值中包含 Video,否则就不包含 Video。

下面的代码声明变量 v,并把 Music、Video 和 Text 三个值进行组合,然后赋给变量 v。

```
// 变量 v 相当于二进制 111
Test v = Test.Music | Test.Text | Test.Video;
```

接着,检查一下 v 中是否包含 Text 值。

```
// 检查变量 v 中是否包含 Text
if ((v & Test.Text) == Test.Text)
{
      Console.WriteLine("变量 v 中包含了 Text.");
}
else
{
```

```
        Console.WriteLine("变量 v 中不包含 Text.");
    }
```

因为 v 中确实包含了 Text，所以(v & Test.Text) == Test.Text 条件成立。屏幕上会输出“变量 v 中包含了 Text。”

如果要从某个组合值中去掉某个值，可以先将该值取反(运算符为～)，比如要从 v 中去掉 Music，先将 Music 取反(就是将每个位上的数 1 变为 0，0 变为 1)，即～001 = 110。再把这个取反后的值与组合数进行与运算，即 111 & 110 = 110，如此一来，就把 Music 所在的位上的值给去掉了，如下面代码所示：

```
// 从 v 中去掉 Music
v = v & ～Test.Music;
// 检查一下是否还含有 Music
if ((v & Test.Music) == Test.Music)
{
        Console.WriteLine("变量 v 中仍含有 Music 的值.");
}
else
{
        Console.WriteLine("变量 v 中不包含 Music 的值.");
}
```

由于 Music 被去掉，因此 v 中不再包含 Music 的值了。示例的运行结果如图 3-39 所示。

图 3-39　输出结果

可是，读者可能还是会有疑问，枚举值也是个整型值，就算声明的枚举类型不带有 FlagsAttribute 特性也能进行组合运算，为什么还要附加上 FlagsAttribute 特性呢？在上面的示例中读者并未看出区别来，因此考虑再用一个示例来验证一下，附加了 FlagsAttribute 特性的枚举与未附加 FlagsAttribute 特性的枚举到底有没有不同之处。

在 Visual Studio 开发环境中新建一个控制台应用程序项目(参考源代码位于\第 3 章\Example_20)，接着，定义两个枚举类型。其中，A 枚举不带 FlagsAttribute 特性，而 B 枚举则加上 FlagsAttribute 特性，两个枚举中定义的常数值相同。代码如下：

```
/// <summary>
/// V1 的二进制值为 1,V2 的二进制值为 10,V3 的二进制值为 100
/// </summary>
enum A { V1 = 1, V2 = 2, V3 = 4 }

/// <summary>
/// V1 的二进制值为 1,V2 的二进制值为 10,V3 的二进制值为 100
```

```
/// </summary>
[Flags]
enum B { V1 = 1, V2 = 2, V3 = 4 }
```

现在,不妨计算一下,把上面枚举中定义的三个数值进行组合(即或运算),得到二进制结果为111,转换为十进制就是7。因此,在上面两个枚举中,如果把三个数值都进行组合,得到的结果就是7。现在,在代码中定义一个int类型的变量,赋值7,然后分别把这个7强制转化为A枚举和B枚举,代码如下:

```
// 001 | 010 | 100 == 111,十进制值为 7
int testValue = 7;
/*
 * 无法从组合的值中识别出 A 枚举
 */
Console.WriteLine((A)testValue);
/*
 * 可以从组合数值中识别出 B 枚举的值
 */
Console.WriteLine((B)testValue);
```

然后,运行应用程序,看看屏幕上输出的内容,如图3-40所示。

```
7
V1, V2, V3
```

图3-40 输出两种不同的结果

从结果中可以看到,不带FlagsAttribute特性的枚举,尽管它的三个值的组合结果为7,但是,强制转换为A枚举类型后无法识别,所以只能输出原值7;而附加了FlagsAttribute特性声明的B枚举,能够识别出7是V1、V2、V3三个枚举值的组合,因此转换为B枚举类型后会输出正确的结果。

3.13 特性

特性可以为程序集、类型,以及类型内部的各种成员添加扩展信息,用于表示一些附加信息。通常,表示特性的类都派生自System.Attribute类,比如AttributeUsageAttribute类等。

在C#语言中使用特性,必须放在一对中括号(英文)中,默认情况下,特性将应用于紧跟其后的对象。举个例子,请考虑下面代码:

```
[Serializable]
public class A { }
```

在上面的代码中,SerializableAttribute特性之后定义了A类,因此该特性将应用于A类;另外,从上面的例子,读者也会发现,在C#中可以略去“Attribute”,因此在代码中只需

输入 Serializable 即可，但一定要注意把特性放在一对中括号中。

SerializableAttribute 类的原型定义如下：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct | AttributeTargets.Enum |
AttributeTargets.Delegate, Inherited = false)]
[ComVisible(true)]
public sealed class SerializableAttribute : Attribute { }
```

查看 SerializableAttribute 类的定义又发现，特性类除了从 Attribute 派生外，也可以向其附加特性，用得最多的就是 AttributeUsageAttribute 特性，它的定义如下：

```
[Serializable]
[AttributeUsage(AttributeTargets.Class, Inherited = true)]
[ComVisible(true)]
public sealed class AttributeUsageAttribute : Attribute
```

让人惊奇的是，在定义 AttributeUsageAttribute 类时也附加了自身作为类的特性。该类指定特性类的适用范围，用 AttributeTargets 枚举来表示，比如特性应用于程序集，或者应用于类、类的属性等，也可以组合多个应用目标，因为 AttributeTargets 枚举也有 FlagsAttribute 特性，前文已经讲述过，作为标记的枚举可以组合使用。

如果特性存在带参数的构造函数，可以在特性后用一对小括号包裹起来，然后在其中传递参数，和调用普通类的构造函数一样。例如：

```
[AttributeUsage(AttributeTargets.Class)]
public class MyInfo : Attribute
```

如上面的例子所示，AttributeUsageAttribute 类有一个带一个参数的构造函数，参数类型为 AttributeTargets 枚举，因此在上面例子中，将 AttributeTargets.Class 传递给 AttributeUsageAttribute 类的构造函数。

如果要为特性类的属性或字段赋值，也是写到一对小括号中，用英文逗号分隔，例如：

```
[MyInfo(AppName = "Compute", Ver = "1.0.0")]
public class Test { }
```

也可以同时附加多个特性，例如：

```
[MyInfo(AppName = "Compute", Ver = "1.0.0")]
[Serializable]
public class Test { }
```

因此，在上面代码中，Test 类应用了 MyInfo 和 Serializable 两个特性。我们同样也可以把多个特性放到一对中括号中，用英文逗号分隔。例如：

```
[MyInfo(AppName = "Compute", Ver = "1.0.0"), Serializable]
public class Test { }
```

3.13.1 自定义特性

定义特性类与定义普通类是一样的,既可以声明构造函数、字段、属性、方法等成员,也可以派生子类,但有一个前提:需要从 System. Attribute 类或者 System. Attribute 的子类派生。总的来说,就是要表明它是一个特性类。

举个例子,下面的代码定义了一个 AppInfoAttribute 特性。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method | AttributeTargets.
Property)]
public class AppInfoAttribute : Attribute
{
    public string Title { get; set; }
    public string VerNo { get; set; }
}
```

AppInfoAttribute 类可以用于类、方法和属性上,并声明了 Title 和 VerNo 两个公共属性。按照习惯,特性类名后应跟上 Attribute 作为后缀,在 C#语言中使用时可以把后面的 Attribute 省略。当然,特性类名也可以不带 Attribute 结尾,加上 Attribute 作为后缀只是为了方便识别该类是特性类。笔者强烈建议在自定义特性类时,应当加上 Attribute 后缀,既方便自己阅读代码,也能方便其他人更容易识别出来。

定义了特性类后,就可以应用到其他类型中了,如下面的代码所示:

```
// 特性应用于类
[AppInfo(Title = "draw", VerNo = "1.0")]
public class Drawer
{
    // 特性应用于属性
    [AppInfo(Title = "color", VerNo = "1.0")]
    public Color Color { get; set; }

    // 特性用于方法
    [AppInfo(Title = "color", VerNo = "1.0")]
    public void DrawRectangle() { }

    // 特性用于字段,编译时会发生错误
    [AppInfo(Title = "thick", VerNo = "1.0")]
    public int Thickness;
}
```

这里要注意，当 AppInfoAttribute 特性用于字段时，发生编译错误，因为 AppInfoAttribute 类在定义时已经指明它只能用于类、方法、属性，并未指定其可用于字段。

3.13.2　如何把特性应用到方法的返回值

在默认条件下，特性将应用于跟随其后的对象，如在类的声明前面加上特性，就是将特性应用于类。将特性应用于方法也一样，在方法声明的前面加上特性；同理，也可以为方法中的参数应用特性。如下面的代码所示：

```
public static string Run([In]string pt, [Optional]int x) { return string.Empty; }
```

为参数应用特性只需放在参数前面即可。但是，如果要为返回值应用特性，那么是不是把特性放在返回值前面就可以了呢？例如：

```
public [MarshalAs(UnmanagedType.SysInt)] int Compute()
```

这样做是错误的，编译无法通过，那是不是说，特性就不能应用于返回值了呢？不是的，在解决这个疑问之前，需要了解一些知识。

在前文中曾提到，默认情况下特性是应用于跟随其后的对象的，因此在许多时候，在使用特性时都会省略表示特性目标的关键字。以下是特性应用于目标对象时的完整格式。

```
[<目标>：<特性列表>]
```

应用目标关键字与特性列表之间用一个冒号(英文)隔开，有效的目标关键字如表 3-2 所示。

表 3-2　特性目标关键字及相关说明

关 键 字	说　明
assembly	表示特性将应用于当前程序集，通常放在程序集中命名空间或所有类型定义之前
module	用于当前模块，该特性用得比较少
field	该特性用于字段，如果特性后紧跟着字段的声明代码，则该关键字可以省略
event	特性用于事件，默认情况下也可以省略
method	该特性用于方法，也可以用于属性中的 get 和 set 访问器，该特性用得较少
param	表示特性用于方法中的参数或属性定义中的 set 访问器中的参数(value)，默认情况下该关键字也可以省略
property	指示特性用于属性，默认情况下也可以省略
type	表示特性用于类型，如类、结构、委托、枚举等。默认情况下也可以省略
return	特性用于方法的返回值，或者属性中 get 访问器的返回值。由于无法在返回值前面附加特性声明，所以，若要为返回值应用特性，return 关键字不能省略

通过上面这个知识点的介绍,读者已经知道如何为方法的返回值应用特性了。即将特性应用到方法上,并且注明特性的应用目标为 return。如下面的代码所示:

```
[return:MarshalAs(UnmanagedType.SysInt)]
public int Compute() { return 0; }
```

3.13.3 通过反射技术检索特性

本节主要向读者说明如何查找特性,这里需要用到反射技术。在本书前面已经讲过,特性可以理解为附加在类型上的一些扩展信息,因此可以通过在类型中找到指定的特性来验证代码的调用方是否符合特定的要求。

下面通过一个实例来演示,便于读者直观理解。本例将定义一个 TypeInfoAttribute 特性,它有一个 Description 属性,表示类型的描述信息。接着通过反射技术来获取到这些描述信息。

在 Visual Studio 开发环境中新建一个控制台应用程序。在 Program 类中声明一个 TypeInfoAttribute 特性类,代码如下:

```
[AttributeUsage(AttributeTargets.All)]
public class TypeInfoAttribute : Attribute
{
    public string Description { get; set; }
}
```

AttributeTargets.All 表示该特性可以应用于所有目标。接下来,声明一个枚举和一个类,并应用 TypeInfoAttribute 特性。

```
[TypeInfo(Description = "这是我们定义的枚举类型.")]
enum TestEnum { One = 1, Two, Three }

[TypeInfo(Description = "这是我们定义的一个类.")]
public class Goods { }
```

然后,在 Main 方法中把以上定义的两个类型的 TypeInfoAttribute 特性读出来。代码如下:

```
// 用 Type 类的 GetCustomAttributes 方法可以获取指定类型上附加的特性列表
// 返回一个 object 类型的数组,数组中的每个元素表示一个特性类的实例
// GetCustomAttributes 方法的其中一个重载可以将一个 Type 作为参数传递
// 该 Type 表示要获取的特性的类型,typeof 运算符返回某个类型的一个 Type
// 本例中我们要获取 TypeInfoAttribute 特性列表
// 由于上面定义 TestEnum 枚举和 Goods 类时,只应用了一个 TypeInfoAttribute 特性
```

```
// 因此获取到的特性实例数组的元素个数总为 1

object[] attrs = typeof(TestEnum).GetCustomAttributes(typeof(TypeInfoAttribute), false);
if (attrs.Length > 0)
{
      TypeInfoAttribute ti = (TypeInfoAttribute)attrs[0];
      Console.WriteLine("TestEnum 枚举的描述信息：{0}", ti.Description);
}

attrs = typeof(Goods).GetCustomAttributes(typeof(TypeInfoAttribute), false);
if (attrs.Length > 0)
{
      TypeInfoAttribute ti = (TypeInfoAttribute)attrs[0];
      Console.WriteLine("Goods 类的描述信息：{0}", ti.Description);
}
```

运行应用程序后，我们会看到如图 3-41 所示的结果。

图 3-41　输出特性的属性值

完整的示例源代码请参考\第 3 章\Example_21。

3.14　数组

数组是从单词 array 翻译过来的，array 的基本含意是“排列，大量，一系列”。因此，从单词的字面意思也可以得知，数组就是把一系列类型相同的元素聚集在一起的一种数据结构。例如，将 1、2、3 三个 int 类型的数值放到一起，便构成了一个带有三个元素的 int 数组。

3.14.1　定义数组的几种方法

定义表示数组的变量与声明普通变量一样，只是要在类型后加上一对空的中括号([])，例如：

```
char[] chars;
```

上面的代码声明了一个 char 数组，即数组中的每个元素都是 char 类型。声明数组后就要对其赋值，一种方法是明确指定数组的元素个数。请考虑下面的代码：

```
int[] nums = new int[3];
nums[0] = 6;
nums[1] = 20;
```

通过 new 运算符创建数组实例,并在后面的一对中括号中指定数组中元素的个数,在上面代码中,nums 数组中包含 3 个 int 类型的元素。要访问数组中的元素,需要通过索引来访问,数组的索引是从 0 开始的,即第一个元素的索引为 0,第二个元素的索引 1,第三个元素的索引为 2……,以此类推。索引值依旧是包含在一对中括号中,如上面的例子中,nums[0]就是访问数组中的第一个元素。上面的代码在创建数组实例后,将第一个元素的值设置为 6,第二个元素的值设置为 20,而第三个元素的值是 0。因为 int 类型的变量默认值为 0。也就是说,在创建数组实例的同时,会对其中所包含的元素进行初始化,如果数组包含的元素是引用类型(比如 string 类型)就初始化为 null。

另一种方法是直接用元素列表来填充数组,数组会根据所列出的元素来计算数组的大小。例如:

```
string[] names = new string[] { "abc", "def" };
```

以上代码在创建数组实例的同时,对各元素也进行了初始化,names 数组中定义了两个元素。还可以简写为以下形式:

```
string[] names = { "abc", "def" };
```

数组默认继承 System.Array 类,所以数组本身是引用类型。开发者无法在代码中显式去从 Array 类派生,它是通过编译器自动完成。明白这一点后,读者会知道,在代码中定义的数组变量是可以使用 Array 类中所公开的成员的。例如可以通过访问 Length 或 LongLength 属性来获取数组中所包含元素的个数。Length 属性是 int 类型,用得较多,如果数组中的元素很多,超出了 int 的有效范围,则可以访问 LongLength 属性来得到元素个数。

可以通过 for 循环来访问数组中的每个元素,例如:

```
for (int i = 0; i < nums.Length; i++)
{
      Console.WriteLine(nums[i]);
}
```

由于 Array 类实现了 IEnumerable 接口,因此,还可以用 foreach 语句来访问数组中的每个元素,例如:

```
foreach (int n in nums)
{
      Console.WriteLine(n);
}
```

接下来读者将完成一个练习,通过这个示例练习,掌握数组的基本使用方法。

（1）在 Visual Studio 开发环境中新建一个控制台应用程序。

（2）声明一个 int 数组，并进行赋值，然后显示数组中的每个元素。代码如下：

```
int[] ints = new int[4];
ints[0] = 100;
ints[1] = 25;
ints[2] = 32;
ints[3] = 900;
Console.WriteLine("通过 for 循环显示数组的所有元素：");
for (int n = 0; n < ints.Length; n++)
{
      Console.Write("{0} ", ints[n]);
}
Console.WriteLine("\n 通过 foreach 循环显示数组的所有元素：");
foreach (int x in ints)
{
      Console.Write("{0} ", x);
}
```

（3）再创建一个 string 数组并初始化，然后输出数组中的元素。

```
string[] strs = { "cat", "car", "food" };
Console.WriteLine("\n\n 字符串数组中的元素列表：");
// 通过 Join 方法将字符串数组的各元素进行拼接
Console.WriteLine(string.Join(", ", strs));
```

（4）修改上面创建的字符串数组中第二个元素的内容，然后再输出一次。

```
// 修改数组中的元素
strs[1] = "sound";
// 重新输出数组的元素
Console.WriteLine("\n 修改后的数组：");
Console.WriteLine(string.Join(", ", strs));
```

整个示例的运行结果如图 3-42 所示。

完整的示例代码请参考\第 3 章\Example_22。

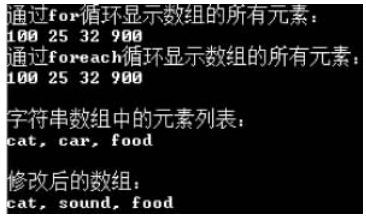


图 3-42　数组示例的输出结果

3.14.2　多维数组

前面提及的是一维数组，其实数组是可以定义为多维的，但是在实际编程中用得不多。使用多维数组通常也仅用到二维数组，维度较高的数组极少会用到。

声明多维数组的方法和一维数组一样，在一维数组

的声明中使用的是一对空的中括号,而对于二维数组,在中括号中加上一个逗号(英文)分隔,以表示两个维度,例如:

```
int[ , ] arr;
```

同理,如果要声明三维数组,中括号中应使用两个逗号,如下面的代码所示:

```
int[, ,] arr;
```

访问多维数组中的元素,方法与访问一维数组中的元素一样,索引依旧是从 0 开始。只是在中括号内部要指明元素的位置。例如:

```
arr[0, 2] = 50;
```

每个维度的索引用逗号隔开。在上面的代码中,该元素位于第一维度的 0 索引和第二维度的 2 索引处。从字面上不太好理解,因此还是回到实践中,通过实例来研究多维数组的结构。此处将以二维数组为例。

在 Visual Studio 开发环境中新建一个控制台应用程序项目(参考示例\第 3 章\Example_23)。待项目创建后,在 Main 方法中声明一个 int 类型的二维数组。代码如下:

```
// 声明二维组数
int[,] arr1 = new int[2, 3];
```

向数组中的元素赋值

```
// 为数组中的元素赋值
arr1[0, 0] = 1;
arr1[0, 1] = 2;
arr1[0, 2] = 3;
arr1[1, 0] = 4;
arr1[1, 1] = 5;
arr1[1, 2] = 6;
```

接着输出数组中的元素

```
// 输出数组中的元素
for (int j = 0; j < 2; j++)
{
      for (int k = 0; k < 3; k++)
      {
            Console.Write(arr1[j, k] + " ");
      }
      Console.Write("\n");
}
```

得到如图 3-43 所示的结果。

图 3-43　输出二维数组的元素

在上面代码中，读者会发现，二维数组中元素的总个数等于各维度上元素个数的乘积。比如上面代码中的 arr1 数组，第一维度是 2 个元素，第二维度是 3 个元素，因此该数组的元素总数为 2×3 = 6。

可以把一维数组用一维表格来表示，二维数组用二维表格来表示，于是便绘制出图 3-44 来模拟数组内部各元素的排列结构。

图 3-44　数组内部元素的排列示意图

与一维数组类似，在实例化二维数组时，可以不指定元素的个数，而是直接向数组中填充元素，数组实例会自动计算元素的个数。如下面的代码所示：

```
// 二维数组的另一种声明方式
char[,] arr2 = { { 'a', 'b', 'c', 'd' }, { 'e', 'f', 'g', 'h' }, { 'i', 'j', 'k', 'l' } };
// 使用 GetLength 方法获取指定维度上元素的个数
int len1 = arr2.GetLength(0);
int len2 = arr2.GetLength(1);
// 输出数组中的元素
      for (int d = 0; d < len1; d++)
      {
            for (int i = 0; i < len2; i++)
      {
                  Console.Write(arr2[d, i] + " ");
      }
      Console.WriteLine();
}
```

对于多维数组，可以通过 GetLength 方法获取特定维度上的元素个数，参数为从 0 开始的整数值，表示维度——0 表示第一维度，1 表示第二维度，以此类推。最后的输出结果如图 3-45 所示。

图 3-45　char 二维数组中的元素

3.14.3　嵌套数组

读者要注意多维数组和嵌套数组二者之间的区别，嵌套数组也叫数组的数组，或者交错数组。通过以下方式来声明变量：

```
int[3][2] arr;
```

就是数组中的每个元素也是数组,也就是数组里面也包含数组。请考虑下面的代码:

```
char[][] ccs = new char[][]
{
      new char[] {'a', 'b'},
      new char[] {'c', 'd'},
      new char[] {'e', 'f', 'w'}
};
```

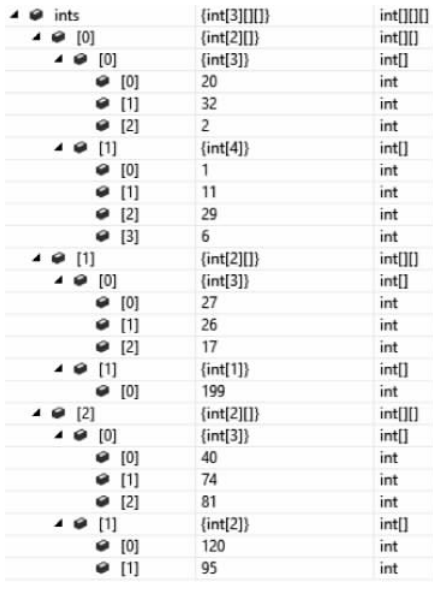
在上面的代码中,声明了一个嵌套数组,该数组从外到内有两层,最外层包含三个元素,而每个元素又是一个 char 数组。第一个 char 数组包含两个元素,第二个 char 数组也包含了两个元素,第三个 char 数组则包含了三个元素。

嵌套数组要比多维数组复杂,它是从外向内一层一层地进行嵌套。其实我们在声明嵌套数组时,可以通过中括号的对数来确定嵌套数组所包含的层数。例如,int[][]表示该数组包含两个层数组,int[][][]则表示其中包含三层数组。

下面用一个示例来演示一个三层嵌套的数组,嵌套数组变量的声明如下:

```
// 三层嵌套的数组
int[][][] ints = new int[3][][]          //第一层
{
      new int[2][]                       //第二层
      {
          new int[] { 20, 32, 2 },       //第三层
          new int[] { 1, 11, 29, 6 }     //第三层
      },
      new int[2][]                       //第二层
      {
          new int[] { 27, 26, 17 },      //第三层
          new int[] { 199 }              //第三层
      },
      new int[2][]                       //第二层
      {
          new int[] { 40, 74, 81 },      //第三层
          new int[] { 120, 95 }          //第三层
    }
};
```

该数组有三个层次(int[][][]),第一层有三个元素,每个元素又是一个两层嵌套的数组(in[][]);然后第二层中每个元素又是一个数组(int[]);到了第三层才是单个 int 数值。可以用图 3-46 来描述这个嵌套数组的内部层次结构。



▲ ints	{int[3][][]}	int[][][]
▲ [0]	{int[2][]}	int[][]
▲ [0]	{int[3]}	int[]
[0]	20	int
[1]	32	int
[2]	2	int
▲ [1]	{int[4]}	int[]
[0]	1	int
[1]	11	int
[2]	29	int
[3]	6	int
▲ [1]	{int[2][]}	int[][]
▲ [0]	{int[3]}	int[]
[0]	27	int
[1]	26	int
[2]	17	int
▲ [1]	{int[1]}	int[]
[0]	199	int
▲ [2]	{int[2][]}	int[][]
▲ [0]	{int[3]}	int[]
[0]	40	int
[1]	74	int
[2]	81	int
▲ [1]	{int[2]}	int[]
[0]	120	int
[1]	95	int

图 3-46　三层嵌套数组的内部结构

然后，把这个嵌套数组的所有元素输出到屏幕。

```
Console.WriteLine(ints.GetType().Name);
for (int i = 0; i < ints.Length; i++)                          //第一层
{
      Console.WriteLine(" {0}", ints[i].GetType().Name);
      for (int j = 0; j < ints[i].Length; j++)                 //第二层
      {
            Console.WriteLine(" {0}", ints[i][j].GetType().Name);
            Console.Write(" ");
            for (int k = 0; k < ints[i][j].Length; k++)        //第三层
            {
                Console.Write("{0} ", ints[i][j][k]);
            }
            Console.WriteLine();
      }
}
```

最后得到如图 3-47 所示的运行结果。

完整的示例代码请参考\第 3 章\Example_24。

其实，嵌套数组的结构有些类似于 Windows 操作系统中的文件目录结构，把嵌套数组的层次与系统中的文件夹层次作类比。从外向内层层嵌套，而最后一层便是数组中的单个元素，类似于文件夹内部的单个文件。在实际开发过程中很少会使用嵌套数组，也不建议读

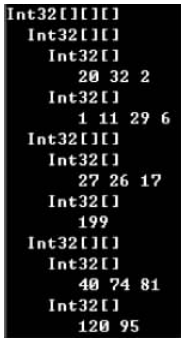


图 3-47 在屏幕上输出嵌套数组

者使用,如果对嵌套数组的层次结构理解不清楚的话,很容易造成不必要的错误;况且,为了方便他人阅读代码,也不宜将数组结构定义得过于复杂。

3.14.4 复制数组

复制数组就是把源数组中的部分或全部元素复制到另一个数组中。Array 类公开了两个方法,可以完成数组的复制功能。

(1) Copy 方法:该方法定义为静态方法,可以直接调用。支持复制一维数组和多维数组。

(2) CopyTo 方法:该方法为实例方法,必须由 Array 类的实例来调用。此方法只能复制一维数组。如果调用 CopyTo 方法来复制多维数组,会引发错误。

用于接收复制元素的数组的容量不能小于源数组的容量。如果源数组的大小为 3,那么目标数组的大小可以大于或等于 3,但不能小于 3。如果目标数组的大小为 2,就无法容纳复制过来的元素;如果目标数组的大小大于源数组的大小,只用复制过来的元素修改相应的位置的值,其他元素的值不变。例如源数组大小为 2,而目标数组的大小为 4,如果复制数组是从目标数组的 0 索引处开始写入,那么目标数组只有前两个元素被复制过来的元素改写,而另外两个元素不变。

下面将完成一个示例(参考源码\第 3 章\Example_25)进行三次数组复制,并且每次复制后都在屏幕上输出目标数组中的元素。

第一次,复制一维数组。

```
int[] srcArr = new int[6] { 1, 2, 3, 4, 5, 6 };
int[] disArr = new int[8];
// 复制
srcArr.CopyTo(disArr, 0);
// 或者 Array.Copy(srcArr, disArr, srcArr.Length);
Array.Copy(srcArr, disArr, srcArr.Length);
// 输出
Console.Write("复制一维数组后目标数组中的元素: \n");
foreach (int b in disArr)
{
      Console.Write(b + " ");
}
```

第二次,复制二维数组。

```
int[,] srcArrd = new int[3, 2]
{
    { 30, 50 }, { 35, 16 }, { 27, 19 }
```

```
};
int[,] disArrd = new int[3, 2];
// 复制
Array.Copy(srcArrd, disArrd, disArrd.Length);
// 输出
Console.Write("\n\n复制二维数组后目标数组中的元素：\n");
foreach (int i in disArrd)
{
      Console.Write(i + " ");
}
```

第三次，复制嵌套数组。

```
int[][] srcArrm = new int[][]
{
      new int[] { 0, 2, 7},
      new int[] { 9, 12, 21}
};
int[][] disArrm = new int[2][];
// 复制
srcArrm.CopyTo(disArrm, 0);
// 或者 Array.Copy(srcArrm, disArrm, disArrm.Length);
// 输出
Console.Write("\n\n复制嵌套数组后目标数组中的元素：\n");
foreach (int[] x in disArrm)
{
      foreach (int y in x)
      {
          Console.Write(y + " ");
      }
}
```

最终的输出结果如图 3-48 所示。

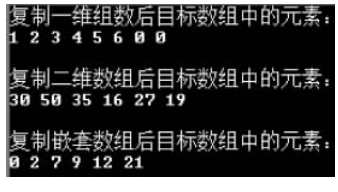


图 3-48　复制结果

3.14.5　反转数组

反转就是数组中的元素反过来重新排序，比如某数组中元素的次序为 1、2、3，那么数组反转后，元素的次序就变为 3、2、1。调用 System.Array 类的 Reverse 方法实现将数组反转，该方法为静态方法，可以直接调用。

示例：\第 3 章\Example_26：该示例将演示如何反转数组中元素的次序。主要代码如下：

```
// 定义数组
byte[] arr = new byte[] { 5, 6, 7, 8 };
```

```
Console.WriteLine("反转前数组的元素及次序: ");
for (int i = 0; i < arr.Length; i++)
{
      Console.WriteLine("[{0}] - {1}", i, arr[i]);
}

// 反转数组
Array.Reverse(arr);

// 反转后再次输出
Console.WriteLine("\n反转后数组的元素及次序: ");
for (int n = 0; n < arr.Length; n++)
{
      Console.WriteLine("[{0}] - {1}", n, arr[n]);
}
```

示例程序运行后,可以看到如图 3-49 所示的结果。

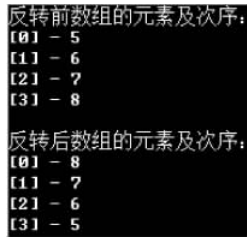


图 3-49 数组反转前后的输出对比

3.14.6 更改数组的大小

读者可能会问:数组实例一旦被创建后,其大小是不是已经被固定了?确实,数组实例创建后,其大小已经固定了,但是通过 Array 类的以下方法可以修改数组实例的大小:

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

该方法是静态方法,且带有类型参数 T,这是属于泛型的一种形式,本书在后面会讲解与泛型有关的知识。简单地说,使用类型参数 T 可以扩大 Resize 方法的适用范围,通过类型参数 T,可以传递任何类型的数组给方法,即 array 参数。如果 T 为 int 类型,就可以传递 int 的数组实例进去;如果 T 为 string 类型,则可以传递 string 数组实例进去。

newSize 是表示新分配的大小。其实,Resize 方法是用 newSize 来创建一个新的数组实例,再把旧数组实例的元素复制到新的数组实例中来取代原来的数组。通过这种方法间接达到修改数组大小的目的。

下面通过实例来学习,请读者在 Visual Studio 开环境中新建一个控制台应用程序项目。接着,声明一个大小为 3(包含 3 个元素)的 int 数组,随后调用 Array.Resize<T>静态方法修改数组的大小为 7(包含 7 个元素)。为了验证原来的数组实例是否被新创建的数组实例替代,代码在修改前后分别输出数组的大小和哈希值。如果原来的数组的实例被替换了的话,那么两次输出的哈希值会不同。具体代码如下:

```
// 创建数组实例
int[] arr = new int[3];
// 修改大小前输出
Console.WriteLine("数组的大小: {0},哈希值为{1}", arr.Length, arr.GetHashCode());
// 修改数组的大小
Array.Resize<int>(ref arr, 7);
// 修改大小后再次输出
Console.WriteLine("数组的大小: {0},哈希值为{1}", arr.Length, arr.GetHashCode());
```

按下键盘上的【F5】键调试运行，会看到如图 3-50 所示的结果。

图 3-50　数组的大小被修改前后的输出对比

第一次输出的大小是 3，第二次输出的大小为 7，但是哈希值不相同。这说明 Resize<T>方法是通过替换旧数组的方式来达到修改数组大小的效果。

完整的示例代码请参考\第 3 章\Example_27。

3.14.7　在数组中查找元素

数组操作基本上都是由 Array 类来负责，因而该类也提供了一系列方法来帮助开发者在数组中进行查找。这些方法按照查找结果划分，大体可以分为两类，下面将分别介绍。

1. 查找元素的索引

此种查找方式将返回被查找到的元素的索引，如果未找到，就返回－1。有两组方法可用：第一组是按照单个元素值来查找；第二组方法则比较灵活，可以通过 System.Predicate<T>委托来自定义查找过程。具体可参考表 3-3。

表 3-3　查找数组中元素的索引的方法

第一组	IndexOf 方法	查找指定元素的索引，只要遇到符合条件的元素就停止查找。如果数组存在多个相同的元素，那么方法只返回第一个满足条件的元素的索引。例如，一个数组中有 2、2、3、5 四个元素，查找元素 2，只返回第一个 2 的索引，第二个 2 将被忽略
	LastIndexOf 方法	与 IndexOf 方法相似，但 LastIndexOf 方法返回匹配的最后一个元素的索引。例如一个数组中包含 a、b、c、c、d 五个元素，在查找元素 c 时，只返回第二个 c 的索引
第二组	FindIndex 方法	与 IndexOf 方法相似，只是可以使用 Predicate<T>委托来自定义查找方式，方法时会把每个元素传给该委托，如果元素符合查找条件，则返回 true，否则返回 false。同样，FindIndex 方法一旦找到第一个符合条件的元素就停止查找
	FindLastIndex 方法	通过 Predicate<T>委托来自定义查找，返回匹配查找条件的最后一个元素的索引

在上面方法的参数中涉及 Predicate 委托,它的定义原型如下:

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

T 是类型参数,该委托接受以 T 类型作为参数,并返回 bool 类型的方法,obj 是数组中待查找的元素,如果 obj 符合查找条件,就返回 true,否则就返回 false。

下面用一个示例演示如何使用上面所列的方法来查找元素的索引。

首先,声明一个数组变量,用于做测试。

```
// 用于测试的数组
string[] testArr = new string[]
{
      /*0*/"ask", /*1*/"check", /*2*/"ask", /*3*/"food", /*4*/"ink"
};
```

数组中包含五个 string 类型的元素。随后分别用 IndexOf、LastIndexOf、FindIndex 和 FindLastIndex 四个方法来对测试数组进行查找,并在屏幕上输出查找到的索引。

```
// check 是数组的第二个元素,索引为 1
int index1 = Array.IndexOf(testArr, "check");
Console.WriteLine("check 元素的索引: {0}", index1);

// 数组中存在两个 ask,索引分别为 0 和 2
// LastIndexOf 方法只返回最后一个 ask 的索引 2
int index2 = Array.LastIndexOf(testArr, "ask");
Console.WriteLine("测试数组中有两个 ask 元素,LastIndexOf 方法返回的索引: {0}", index2);

// 通过自定义方式,查找以 k 结尾的元素
// 第一个元素 ask 就是 k 结尾,已满足条件,不再往下查找
// 因此返回第一个 ask 的索引 0
int index3 = Array.FindIndex(testArr, new Predicate<string>(FindProc));
Console.WriteLine("\nFindIndex 方法查找以 k 结尾的元素的索引: {0}", index3);

// 自定义方式查找以 k 结尾的元素
// 测试数组中索引 0、1、2、4 四处的元素都以 k 结尾
// 但 FindLastIndex 只返回最后一个匹配项 ink 的索引 4
int index4 = Array.FindLastIndex(testArr, new Predicate<string>(FindProc));
Console.WriteLine("FindLastIndex 方法查找以 k 结尾的元素的索引: {0}", index4);
```

“check”是数组的第二个元素,索引为 1,故 index1 的值为 1;测试数组中有两个“ask”元素,分别是第一个和第三个,LastIndexOf 方法返回匹配的最后一项的索引,虽然索引 0 和 2 处都找到“ask”元素,但是由于索引 2 处是最后一处,故 index2 变量的值为 2;在测试数组中有四个元素是 k 结尾的,但 FindIndex 只返回第一个匹配项的索引,因为第一个元素“ask”就是以 k 结尾的,符合条件,所以 index3 的值为 0;FindLastIndex 方法返回最后一个

以 k 结尾的元素“ink”的索引，所以 index4 变量的值为 4。

下面的代码是自定义查找方式的 FindProc 方法的处理过程。

```
private static bool FindProc(string obj)
{
        if (obj.EndsWith("k"))
        {
                return true;
        }
        return false;
}
```

运行应用程序后，会看到如图 3-51 所示的运行结果。

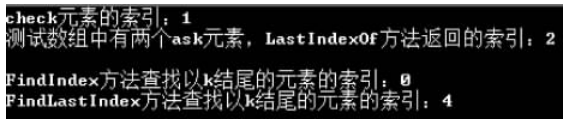


图 3-51　屏幕上输出的查找结果

完整的示例源代码请参考\第 3 章\Example_28。

2. 查找元素自身

这种查找方式的结果不是返回元素在数组中的索引，而是直接返回元素自身，也就是返回所找到的元素的值。Array 类提供了三个方法用于查找元素，它们分别是：

(1) Find 方法：查找符合条件的元素，如果找到，就不再往下查找；如果没有找到满足条件的元素，则返回类型的默认值。例如，如果要查找的目标类型是 int，在找不到符合条件的元素时就返回 int 的默认值 0。

(2) FindLast 方法：查找满足条件的元素，并返回符合条件的最后一个元素，和 FindLastIndex 方法类似。例如，一个整型数组包含 1、2、3、4 四个元素，如果查找的条件是小于 4 的元素，那么符合条件的元素有 1、2、3 三个，FindLast 方法将返回最后匹配的元素，即返回 3。

(3) FindAll 方法：按照指定的条件进行查找，返回所有符合条件的元素，以数组的形式返回。例如，一个 int 数组包含 1、2、3、4 四个元素，查找条件为小于 3 的元素，则 FindAll 方法将返回一个新的 int 数组，该数组包含符合条件的两个元素 1 和 2。

接下来，读者可以通过一个示例，来学习如何使用上述方法在数组中查找元素。

(1) 在 Visual Studio 开发环境中新建一个控制台应用程序项目。

(2) 定义一个 int 数组，稍后用于测试。

```
// 声明数组变量
int[] arr = { 3, 6, 35, 10, 9, 13 };
```

(3) 分别使用 Find、FindLast 和 FindAll 三个方法在数组中查找小于 10 的元素，并输

出查找结果。详细代码如下：

```
// Find 方法只返回匹配的第一个元素
// 数组中第一个小于 10 的元素是 3,故返回 3
int result1 = Array.Find(arr, new Predicate<int>(FindCallback));
Console.WriteLine("Find 方法查找小于 10 的元素: {0}", result1);

// FindLast 方法返回匹配元素的最后一项
// 数组中最后一个小于 10 的元素是 9,故返回 9
int result2 = Array.FindLast(arr, new Predicate<int>(FindCallback));
Console.WriteLine("FindLast 方法查找小于 10 的元素: {0}", result2);

// FindAll 方法返回所有匹配的元素
// 数组中 3、6、9 都小于 10
// 因此,返回一个由 3、6、9 三个元素组成的数组
int[] result3 = Array.FindAll(arr, new Predicate<int>(FindCallback));
Console.WriteLine("\nFindAll 方法查找所有小于 10 的元素: ");
foreach (int x in result3)
{
      Console.Write(x + " ");
}
```

数组中小于 10 的有三个元素：3、6、9。Find 方法只返回第一个符合条件的元素，所以返回 3；FindLast 方法返回符合条件的最后一个元素，所以返回 9；FindAll 方法返回所有符合条件的元素，因此返回 3、6、9。

(4) 以下的代码定义 FindCallback 方法，用于 Predicate<T>委托。如果元素小于 10 则返回 true，否则返回 false。

```
private static bool FindCallback(int val)
{
      if (val < 10)
            return true;
      return false;
}
```

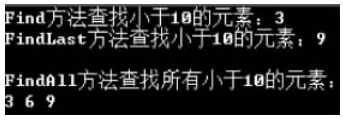


图 3-52 查找结果

示例的运行结果如图 3-52 所示。

完整的示例源代码请参考\第 3 章\Example_29。

3.14.8 灵活使用 ArrayList 类

通过前面的学习，读者对于数组已经不陌生了，而且读者也了解到，在同一个数组实例中，只能放置类型相同的元素。本节将介绍一个可以在其中放置不同类型的类似数组结构的类——ArrayList(位于 System.Collections 命名空间)。

ArrayList 类不仅可以添加不同类型的元素，而且容量会随着新元素的添加自动增长，

也可以通过 Capacity 属性修改 ArrayList 实例的容量。

ArrayList 类虽然可以放置不同的类型，但也会在一定程度上影响性能，因此最好不要向 ArrayList 中添加过大的元素。

ArrayList 类的使用并不复杂，下面将通过一个示例来做演示。启动 Visual Studio 开发环境，并新建一个控制台应用程序项目。核心代码如下：

```
ArrayList list = new ArrayList();
// 添加 int 类型对象
list.Add(100);
// 添加 double 类型对象
list.Add(20.977d);
// 添加 string 类型对象
list.Add("hello");
// 添加 long 类型对象
list.Add(9800000L);

// 可以通过索引取得元素
// 要进行类型转换
// 取出来的元素的类型要与放入时对应
Console.WriteLine("[0] - {0}", (int)list[0]);
Console.WriteLine("[3] - {0}", (long)list[3]);
// 输出元素总个数
Console.WriteLine("元素个数: {0}", list.Count);
// 删除最后一个元素
list.RemoveAt(list.Count - 1);
// 删除后再次输出元素个数
Console.WriteLine("删除后元素个数: {0}", list.Count);
```

首先创建一个 ArrayList 实例，随后向 ArrayList 实例依次添加一个 int 类型的元素、一个 double 类型的元素、一个 string 类型的元素、一个 long 类型的元素。

接着读出第一个和第四个元素，并输出到屏幕。然后，调用 RemoveAt 方法删除最后一个元素，索引为 list.Count-1，因为索引是从 0 开始的，所以最后一个元素的索引应为元素总数减去 1。删除元素前后向屏幕输出 ArrayList 实例的元素个数，以方便进行对比。

如果要删除 ArrayList 中指定的元素，而不是通过索引操作，可以使用 Remove 方法，该方法调用时向参数传递指定元素的具体的值，而 RemoveAt 方法在调用时，是通过传递要删除的元素的索引作为参数的。

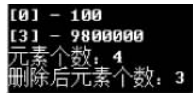


图 3-53　应用程序的输出结果

示例的运行结果如图 3-53 所示。

完整的示例代码请参考\第 3 章\Example_30。