第 3 章　面向对象设计基础

第二章介绍了 C#的基本语法，以及使用方法。C#同 Java、C++一样是面向对象的编程语言，同时 C#更强化了面向对象的概念。本章将介绍面向对象的基础知识并介绍使用 C#编写面向对象的应用程序，在 C#中，面向对象的开发能够给系统设计、编码、维护提供更多的便利。

3.1　什么是面向对象

面向对象是应用程序开发中一个非常重要的技巧和概念，面向对象并不是什么高深的技术也不是负责的学习体系，面向对象主要是一种设计的思路。使用面向对象进行应用程序开发能够非常好的将现实中的物体进行抽象，这样就在一定程度上丰富了应用程序的结构，不仅如此，面向对象还包括继承、多态等特性以便能够快速构架应用程序。

3.1.1　传统的面向过程

在传统应用程序开发领域（如 C 语言的开发），或者是早期的基于 B/S 领域的 Web 应用程序开发（如 ASP）都使用的是传统的面向过程的开发语言。而 C++/JAVA/C#等开发都是使用的面向对象的开发语言。面向对象的开发语言更接近人类理解自然的语言，对开发人员来说更加通俗易懂，同时也对“对象”进行了较好的抽象。面向过程的一段 C 语言代码如下所示。

```
main()
{
sum(x,y);
get(x,y);
print(x);
print(y);
}
```

上述代码截取了 C 语言中的一个代码段，从上述代码中可以看出，C 语言中基本没有对象的概念。当执行一个主方法 Main 时，按照程序逻辑调用不同的函数，来达到运算的目的。传统的面向过程的开发必须规定每一个步骤，或者明确每一种函数，并在程序运行中调用不同的函数来实现运算的目的。面向过程的思想决定了其没有派生、覆盖、继承等特性，所以每当创建一个新的“对象”时，就有可能需要编写更多的代码，这在一定程度上造成了代码的编写和维护的困难。

3.1.2　面向对象的概念

在面向过程的开发当中，开发人员发现在调用函数的时候，很难分清楚函数本身是属于哪个文件的，在代码的阅读上面，不同的开发人员会发现很难读懂其他的开发人员的代码。虽然注释和良好的命名都是必要的，但是还是给开发人员之间的交互造成了极大的困扰。为了解决这个问题，于是有了面向对象的概念。面向对象的一段 C#代码如下所示。

```
class Program                                                    //主程序类
```

```
{
    public int sum(int x, int y)                                    //sum 方法
    {
        return x + y;                                               //返回值
    }
    static void Main(string[] args)                                 //静态方法
    {
        int x = 1, y = 2;                                           //声明整型变量
        Program sum = new Program();                                //创建对象
        Console.WriteLine(sum.sum(x, y));                           //输出方法返回值
    }
}
```

上述代码中，sum 是一个 Program 对象，sum 有一个方法 sum 进行加法运算。读者可能会疑惑，相对于面向过程，面向对象的代码量好像变多了，而且执行的方法过程并没有太大的区别。其实面向对象解决了代码难以划分结构、代码可读性不高的问题，让程序变得更加容易组织和阅读。例如在.NET 中的 Convert.ToInt32 静态方法，当阅读到此代码的时候，开发人员能够比较清楚的知道这个方法做了什么操。而在调用面向过程中的函数的时候，如果函数的名称是 CTi32，而又分不清该函数在上下文中起到的作用时，会比较难以理解这个方法究竟做了什么。

3.1.3 面向组件的概念

面向组件其实是面向对象的另一种加强。在面向对象中，常常需要引用命名空间或者引用头文件（如 C++）来说明一个函数所在的对象与另一个同样名称的函数所在的对象是不同的。在传统的应用程序开发过程中，当客户更改了若干需求，往往只需要修改一个很小的功能，就要改动很多的代码，因此就出现了软件危机。

于是人们使用了分层的概念，将代码封装在一个类，然后对类进行组织协调，通过编译器对类或类库进行编译，形成 DLL 组件。在应用程序中使用 DLL，从而提高了代码的重用性。分层的概念也是设计模式的开端。面向组件的概念在平时就已经被读者广泛使用了，例如.NET 中的某些类库，还有 COM 组件等。面向组件的概念对开发人员在设计的思想上要求更高，这些要求不仅仅局限于编码。

3.2 面向对象的 C#实现

C#是面向对象的编程语言。在面向对象开发当中，不可避免的要创建一个类，创建类后还需要创建该类的属性和方法来描述对象，然后再创建这个类的对象进行实例化。创建后的对象能够通过类中的属性和方法完成相应的操作。

3.2.1 定义

什么是对象？世间万物皆对象，在生活中，可能是一只猫、一只狗，或者是饼干、一张订单、银行卡等等都是对象。对象描述了一个物体的特征，抽象一个实际的物体或一个方法。

类定义了对象的特征，对对象进行了描述。这些特征包括对象的属性、对象的方法、对象的权限，以及如何访问该对象。在生活中就有很多例子，例如人类属于世界上的动物，并属于哺乳动物，同样，猫、狗、鸟也属于哺乳动物。所以，哺乳动物能够描述人类、猫、狗、鸟的一些基本特性。但是，人类

会说话；猫会爬树；鸟会飞，不同的动物之间，实现的功能又不尽相同。所以这些对象之间也是有区别的。

程序开发人员能够为哺乳动物定义基本的相同的特性，如重量、体色、有没有毛之类，同样也能定义哺乳动物是否能够行走和飞行或者进行其他的操作。当然，如果定义了人类能够飞行，这是非常不符合逻辑的，类的设计同时也是需要符合逻辑。

注意：虽然在类设计中，能够自行设计相应的类和方法，以及属性，例如人类能够飞行，但是这个类的设计是不复合逻辑的，也是没有必要的。所以在类设计中，只要尽量完整的描述与现实中相同的属性即可。

3.2.2 创建一个类和其方法

上一节中了解了什么是面向对象，初学者可能还是对什么是面向对象、什么是对象、为什么要使用面向对象等概念很模糊。这里可以通过创建一个类可以深入了解面向对象的概念。在 Visual Studio 2008 中，系统提供了类的创建向导，如图 3-1 所示。

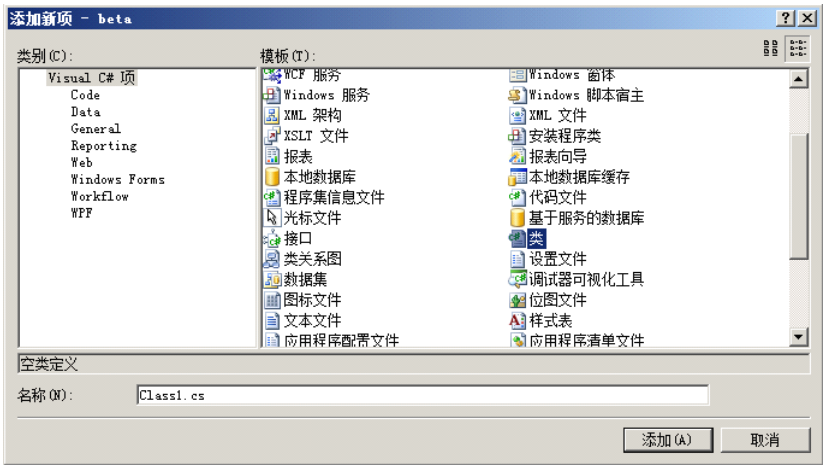

图 3-1　创建一个类

创建一个类的步骤如下所示。

（1）在 Visual Studio 2008 中打开一个项目。

（2）右击该项目，在下拉菜单中选择【添加】选项，在【添加】下拉菜单中选择【新建项】选项。

（3）在弹出对话框中选择【类】选项并，如图 3-1 所示，并给类一个名称。

（3）单击【添加】按钮，类就添加到项目中了。

技巧：也可以在导航菜单栏中的【文件】菜单栏中选择【新建文件】创建一个类文件。

在创建了类文件之后，就能够编写代码创建类描述对象，为了能够描述哺乳动物，这里创建一个 Animal 类，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;                                          //声明命名空间
using System.Text;                                          //不同的命名空间
namespace MyClass                                           //声明命名空间
{
    class Animal                                            //定义类
    {
```

```
    }
}
```

使用 C#创建一个类，命名空间会包含在类文件中。默认命名空间通常与创建的项目名称相同，示例代码如下所示。

```
namespace MyClass                                                    //当前程序的命名空间
```

类被创建之后，可以向类中添加方法、属性等，以便更加清晰的描述该对象，示例代码如下所示。

```
namespace MyClass
{
    public class Animal                                              //创建对象
    {
        string color;                                                //对象包含的字段
        public string get()                                          //对象的方法
        {
            return color;                                            //执行的方法
        }
    }
}
```

在主函数中，可以调用 Animal 类并创建该类的对象并执行相应的方法，这样就能够很好的描述一个对象并执行相应对象的动作，示例代码如下所示。

```
        static void Main(string[] args)
        {
            Animal an = new Animal();                                //创建对象
            an.get();                                                //执行方法
        }
```

上述代码首先创建了一个 Animal 对象 an，在创建对象后执行了对象的 get 方法进行该对象的颜色的获取，从而得到了一个动物 an 的颜色。

3.2.3 类成员

在 C#中，类包含若干个组成成员，这些组成成员包括字段、属性、方法、事件等，这些组成成员能够彼此协调用于对象的深入描述。

1. 字段

“字段”是包含在类中的对象的值，字段使类可以封装数据，字段的存储可以满足类设计中所需要描述。例如上一节中 Animal 类中的字段 color，就是用来描述动物的颜色。当然，Animal 的特性不只颜色，可以声明多个字段描述 Animal 类的对象，示例代码如下所示。

```
    class Animal
    {
        public string color;                                         //声明颜色字段
        public bool haveFeather;                                     //声明是否含有羽毛字段
        public int age;                                              //年龄字段
    }
```

上述代码中，对 Animal 类声明了另外两个字段，用来描述是否有羽毛和年龄。当需要访问该类的字段的时候，需要声明对象，并使用点“.”操作符实现，Visual Studio 2008 中对“.”操作符有智能提示功能，示例代码如下所示。

```
            Animal bird = new Animal();                              //创建对象
            bird.haveFeather = true;                                 //鸟有羽毛
```

```
bird.color = "black";                                    //这是一只黑色的鸟
```

2．属性

C#中，属性是类中可以像类的字段一样访问的方法。属性可以为字段提供保护，避免字段在用户创建的对象不知情的情况下下被更改。属性机制非常灵活，提供了读取、编写或计算私有字段的值，可以像公共数据成员一样使用属性。

在 C#中，它们被称为“访问器”，为 C#应用程序中类的成员的访问提供安全性保障。当一个字段的权限为私有（private）时，不能通过对象的“.”操作来访问，但是可以通过“访问器”来访问，示例代码如下所示。

```
public class Animal
{
    private int _age;                                    //定义私有变量
    public int Age { get { return _age; } set { _age = value; } }    //赋值属性
}
```

上述代码中为 Animal 类声明了一个属性 Age，在主程序中，同样可以通过“.”操作符来访问属性，示例代码如下所示。

```
Animal bird = new Animal();                              //创建对象
bird.Age = 1;                                            //Age 访问了_age
```

在 Visual Studio 2008 中，属性的声明被简化，不再需要冗长的声明，示例代码如下所示。

```
public class Animal                                      //创建类
{
    public int Age { get; set; }                         //简便的属性编写
}
```

注意：虽然在 VS2008 中，简化了代码，但是实现的过程依旧没有改变。

3．方法

方法用来执行类的操作，方法是一段小的代码块。在 C#中，方法接收输入的数据参数，并通过参数执行函数体，返回所需的函数值，方法的语法如下所示。

```
私有级别 返回类型 方法名称(参数 1，参数 2)
{
    方法代码块。
}
```

方法在类中声明。对方法的声明，需要指定访问级别、返回值、方法名称以及任何必要的参数。参数在方法名称后的括号中，多个参数用逗号分割，空括号表示无参数，示例代码如下所示。

```
public string output()                                   //一个无参数传递的方法
{
    return "没有任何参数";                               //返回字符串值
}
public string out_put(string output)                     //一个有参数传递的方法
{
    return output;                                       //返回参数的值
}
```

上述代码中，创建了两个方法，一个是无参数传递方法 output 和一个参数传递的方法 out_put，在主函数中可以调用该方法，调用代码如下所示。

```
Animal bird = new Animal();                              //创建对象
bird.out_put();                                          //使用无参数的方法
string str = "我是一只鸟";                               //创建字符串用于参数传递
```

```
        bird.out_put(str);                                          //使用有参数的方法
```

如上述代码所示，主函数调用了一个方法 out_put，并传递了参数“我是一只鸟”。在使用类中的方法前，将“我是一只鸟”赋值给变量 str，传递给 out_put 函数。在上述代码中，“我是一只鸟”或者 str 都可以作为参数。

在应用程序开发中，方法和方法之间也可以互相传递参数，一个方法可以作为另一个方法的参数，方法的参数还可以作为另一个方法的返回值，示例代码如下所示。

```
public string output()                                              //一个无参数传递的方法
{
    return "没有任何参数";                                          //返回字符串
}
public string out_put()                                             //使用其他方法返回值的方法
{
    string str = output();                                          //使用另一个方法的返回值
    return str;                                                     //返回方法的返回值
}
```

如上述代码所示，out_put 使用了 output 方法，output 返回一个字符串“没有任何参数”。在 out_put 方法中，使用了 output 方法，并将 output 方法的返回值赋给 str 局部变量，并返回局部变量。在方法的编写中，方法和方法之间可以使用同一个变量而互不影响，因为方法内部的变量是局部变量，示例代码如下所示。

```
public string output()                                              //一个无参数传递的方法
{
    string str = "没有任何参数";                                    //声明局部变量 str
    return str;                                                     //使用局部变量 str
}
public string out_put()                                             //一个无参数传递的方法
{
    string str = "还是没有任何参数";                                //声明局部变量 str
    return str;                                                     //使用局部变量 str
}
```

如上述代码所示，output 和 out_put 方法都没有任何参数，但是却使用了同一个变量 str。str 是局部变量，str 的作用范围都在该变量声明的方法内，称作“作用域”。创建了一个方法，就必须指定该方法是否有返回值。如果有返回值，则必须指定返回值的类型，示例代码如下所示。

```
public int sum(int number1, int number2)                            //必须返回 int 类型的值
{
    return number1 + number2;                                       //返回一个 int 类型的值
}
public void newsum(int number1, int number2)                        //void 表示无返回值
{
    int sum = number1 + number2;                                    //没有返回值则不能返回值
}
```

上述代码中，声明了两个方法，分别为 sum 和 newsum。sum 方法中，声明了该方法是共有的返回值为 int 的方法，而 newsum 方法声明了是共有的无返回值方法。

4．事件

事件是一个对象向其他对象提供有关事件发生的通知的一种方式。在 C#中，事件是使用委托来定义和触发的。类或对象可以通过事件向其他类或对象通知发生的相关事情。发送或引发事件的类称为“发行者”，接收或处理事件的类称为“订阅者”。例如在 Web Form 中双击按钮的过程，就是一个事件，

控件并不对过程做描述，只是负责通知一个事件是否发生。事件具有以下特点：

- 事件通常使用委托事件处理程序进行声明。
- 事件始终通知对象消息并指示需要执行某种操作的一种方式。
- 发行者确定何时引发事件，订阅者确定执行何种操作来响应该事件。
- 一个事件可以有多个订阅者。一个订阅者可处理来自多个发行者的多个事件。
- 没有订阅者的事件永远不会被调用。
- 事件通常用于通知用户操作，例如，图形用户界面中的按钮单击或菜单选择操作。
- 如果一个事件有多个订阅者，当引发该事件时，会同步调用多个事件处理程序，也可以使用异步处理多个事件。

声明委托和事件的示例代码如下所示。

```
public delegate void AnimalEventHandler();                    //声明委托
public class Animal                                           //创建类
{
    public event AnimalEventHandler OnFly;                    //声明事件
    public void Fly()                                         //创建类的方法
    {
        OnFly();                                              //使用事件
    }
}
```

上述代码定义了一个委托，并针对相关委托声明了一个方法。关于委托和事件，会在后面的章节中讲到，上述代码在主函数调用代码如下所示。

```
        Animal bird = new Animal();                           //创建对象
        bird.OnFly += new AnimalEventHandler(TestAnimal);     //注册事件
        bird.Fly();                                           //使用方法
```

3.2.4 构造函数和析构函数

构造函数和析构函数是面向对象中一个非常特别的函数，构造函数在对象初始化时被执行而析构函数在对象被销毁时被执行。开发人员无需显式的进行函数的编写，在C#应用程序开发中能够为开发人员提供默认的构造函数和析构函数。

1．构造函数

在变量中，常常需要初始化变量，而在类的声明中，也需要构造和初始化类。在类中，构造函数是在第一次创建对象时调用的方法。在任何时候，只要创建了一个对象，编译器就会默认的调用构造函数并执行构造函数。构造函数与类名相同，并且一个类可以有一个或多个构造函数，示例代码如下所示。

```
public class Animal                                           //创建一个类
{
    public string AnimalName;                                 //创建 AnimalName 名称字段
    public Animal()                                           //使用构造函数
    {
        AnimalName = "动物";                                  //赋值私有变量
    }
}
```

上述代码声明了一个 Animal 类，并使用构造函数，构造函数与类同名。当声明一个对象时，系统默认使用此构造函数进行对象的构造。另外，构造函数也可以传递参数，示例代码如下所示。

```
public class Animal                                           //创建一个类
```

```
{
    public string AnimalName;                                   //创建 AnimalName 名称字段
    public Animal()                                             //无参数的构造函数
    {
        AnimalName = "动物";                                    //赋值私有变量
    }
    public Animal(string name)                                  //有参数的构造函数
    {
        AnimalName = name;                                      //私有变量获取传递的参数
    }
}
```

构造函数可以传递参数，当声明一个对象时，若不指定构造函数，系统会默认使用无参数的构造函数。在初始化时，若指定构造函数，系统会按照指定的构造函数构造对象，示例代码如下所示。

```
        Animal dog = new Animal("狗");                          //通过构造函数创建对象
        Console.WriteLine(dog.AnimalName);                      //输出对象的属性
```

在初始化中，直接将参数初始化并传递给构造函数则会在初始化中为对象中的字段初始化。构造函数方便了开发人员设置默认值、限制实例化来编写灵活并且便于阅读的代码。

2．析构函数

析构函数是将当前对象从内存中移除时运行和调用的方法，析构函数的使用通常是为了确保需要释放的对象资源都得到了适当的处理。析构函数的函数名和类名基本相同，在方法前还需要“~”符号来声明该方法是类的析构函数。对于析构函数，有以下几个特点。

- 只能对类定义析构函数，结构不支持析构函数。
- 一个类只能有一个析构函数。
- 无法继承或重载析构函数。
- 无法调用析构函数，在对象注销时，系统会自动调用。
- 析构函数即没有修饰符也不能为它传递参数。

创建析构函数示例代码如下所示。

```
public class Animal                                             //创建类
{
    public string AnimalName;                                   //创建 AnimalName 名称字段
    public Animal()                                             //使用构造函数
    {
        AnimalName = "动物";                                    //赋值共有字段
    }
    ~Animal()                                                   //使用析构函数
    {
        AnimalName = String.Empty;                              //将字符串清空
    }
}
```

上述代码中，当 Animal 类的对象被销毁时，同时也将 AnimalName 设置为 String.Empty。构造函数隐式的对对象的基类调用 Finalize，所以开发人员无法控制在何时调用析构函数。在 C++中，当对象被销毁时，系统会调用析构函数并释放对象，而在 C#中，垃圾回收机制会自动清理对象资源。在确保了对象没有被任何应用程序使用后，C#的垃圾回收机制会自动清除不再使用的对象的资源。对于开发人员而言，虽然可以显式的使用 Collect 进行垃圾回收机制，但是会影响应用程序的性能。

3.3 对象的生命周期

在上一节中声明了类并说明了类成员，这些类成员包括字段、方法、事件、构造函数以及析构函数。类是对象的设计图（也称为模板），类用于描述对象。在创建对象后，对象就开始了其生命周期，只有在生命周期内的对象才能够被使用，否则无法使用相应的对象。

3.3.1 类成员的访问

类声明的方法是以 class 关键字开头，后面紧接着类名字，并以“{”“}”大括号包裹住类成员，示例代码如下所示。

```
访问权限 class 类名称
{
    类成员
}
```

例如在 3.2.2 中创建了一个 Animal 类，其中类名称就是 Animal。在实例化一个对象之后，在主程序或其他代码段中，需要对实例化的对象进行访问，即需要对类成员的访问。访问类成员的方法就是在对象后使用“.”号，并通过 Visual Studio 2008 智能提示选择相应的类成员，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;                                          //使用 LINQ 命名空间
using System.Text;                                          //使用文本命名空间
namespace MyClass
{
    public class Animal                                     //创建一个类
    {
        string type;                                        //声明了类成员 string type
        public void SetType(string type)                    //声明了类方法
        {
            this.type = type;                               //字段赋值
        }
    }
    class Program                                           //主程序类
    {
        static void Main(string[] args)                     //程序入口方法
        {
            Animal bird = new Animal();                     //创建了一个 bird 对象
            bird.SetType("bird");                           //引用了一个对象的成员
        }
    }
}
```

上述代码中，创建了一个公共类 Animal，并创建了类成员字段 type 和方法 SetType。在主函数中，创建了一个 Animal 对象 bird，并通过“.”号访问了类成员 SetType 方法。在访问类或类成员时，可以通过关键字来限制类或类成员的访问权限，以便只有该类的派生类才能访问或者使用，同时也能够限制类成员的权限，提高类成员的安全性。这些关键字包括 public、private、protected 和 internal。

1．public 共有权限

public 字段具有最高访问级别，任何它的对象或者其他的类都能对 public 关键字所修饰的类或类成员进行访问，示例代码如下所示。

```
public class Animal                                          //共有的类
{
    public string type;                                      //共有的字段
    public void SetType(string type)                         //共有的方法
    {
        this.type = type;                                    //赋值共有字段
    }
}
```

2．private 私有权限

private 字段具有最低的访问级别，它能够保证类和类成员的安全，却限制了其他类或对象对它的访问。私有成员只有在声明他们的类之后才能访问，示例代码如下所示。

```
public class Animal
    {
    private int age;                                         //私有成员
    string type;                                             //默认的私有成员
    public void SetType(string type)
    {
        this.type = type;                                    //赋值私有成员
    }
}
```

3．protected 保护权限

protected 字段具有保护类中字段的功能，能够保证类和类成员的安全性，也能够限制其他类或对象对它的访问。但是与 private 不同的是，protected 能够在类和类的的派生类中使用，比 private 具有更高的访问级别，又比 public 拥有更低的访问级别，保证了类的安全性，示例代码如下所示。

```
public class Animal
{
    protected string str;                                    //受保护的成员
}
```

4．internal 程序集保护权限

internal 字段修饰的类或类成员只有在同一程序集的文件中内部类型或成员才可以访问，示例代码如下所示。

```
public class Animal
{
    internal string str;                                     //受保护的程序集内的成员
}
```

这种程序集的文件中内部类型或成员才可以访问的修饰符通常是基于组件开发的，因为它能够使一组组件能够以私有方式进行合作，保证了组件的安全性。通常情况下，ASP.NET 中页面控件都是通过内部组件方式进行合作。另一方面，这些访问权限修饰符还能够组合使用，例如 protected internal 就可以进行组合使用，组合使用所修饰的对象只有该类和该类派生的类的成员才可以访问。

3.3.2 类的类型

每一个类的对象都是独立的对象，对象与对象之间有共同的属性，但是对象与对象之间不存在联系，虽然很多情况下类也可以引用类，示例代码如下所示。

```
public class Animal                                          //创建类
{
    public string type;                                      //创建字符串型共有变量
}
class Program                                                //主程序类
{
    static void Main(string[] args)                          //程序入口方法
    {
        Animal bird = new Animal();                          //bird 对象
        bird.type = "bird";                                  //初始化字段
        Animal cat = new Animal();                           //cat 对象
        cat.type = "cat";                                    //初始化字段
    }
}
```

上述代码创建了两个对象，一个对象为 bird，另一个为 cat 对象。在初始化类成员时，为不同的对象的类成员赋了不同值，虽然这些类成员的名称相同，但是“.”号说明了该成员所在的对象是不同的。另外，由于类是引用类型，所以类的对象之间可以互相赋值，示例代码如下所示。

```
        Animal newbird = new Animal();                       //创建对象
        newbird = bird;                                      //对象之间互相赋值
```

上述代码将对象 newbird 初始化后并通过 bird 赋值，所以对象 newbird 中的 type 的值等于对象 bird 中的 type 值，因为 newbird 和 bird 都是引用的同一个对象。

3.3.3 .NET 的垃圾回收机制

当创建一个对象，.NET 对该对象初始化并在内存相应位置存储，当一个对象执行析构函数时，该对象被销毁并释放相关资源。在 C++中，使用析构函数能够让开发人员显式的释放资源。而在.NET 中，由于使用了垃圾回收机制（GC）从而导致开发人员无法控制析构函数是何时被运行的。

垃圾回收机制监视对象的生存周期，当一个对象没有被任何应用程序引用时，垃圾回收器就释放对象所占的内存以及资源。在基于.NET Framework 编程时，开发人员无需像 C++中显式的释放对象的资源也无需关心对象所占用的内存，因为.NET Framework 的垃圾回收器能够监视对象并在相应的时候释放对象的资源。

垃圾回收机器没有固定的工作模式。它的工作间隔是不可预期的，一般情况下，当应用程序占用的内存不足的时候会启用垃圾回收器释放未被引用的对象的资源。在应用程序使用复杂并昂贵的外部资源的时候，.NET 机制提供接口能够让开发人员实现垃圾回收，以及资源释放机制，通过实现来自 IDisposable 接口的 Dispost 方法可以完成显式的资源释放。

在 ASP.NET 或者 Win Form 开发中，显式的使用 Dispost 方法能够提高应用程序的性能。同样，析构函数也是一种清理资源的方法，在对象析构时，可以用 Dispose 对对象的资源，以及连接信息进行清空从而对对象进行释放。

3.4 使用命名空间

在应用程序开发过程中，类和类成员的名称是丰富的，为了描述一个具体的对象，需要对类成员进行设计。在设计类和类成员过程中，不可避免的类成员中的方法或者类的名称会出现相同的情况，这样就会使类的使用变得复杂，代码的混乱造成可读性降低，使用命名空间可以解决此类难题。

3.4.1 为什么要用命名空间

正如引言中所述，在设计类和类成员的过程中，不可避免的，类或类成员使用的名称都是相同的，这样就让开发更加复杂，代码可读性降低。使用命名空间能够解决此类问题，示例代码如下所示。

```
namespace Programmer1                                    //程序员 1 的命名空间
{
    public class Animal                                  // Programmer1 的 Animal 类
    {
        public string type;                              //声明字段
    }
}
namespace Programmer2                                    //程序员 2 的命名空间
{
    public class Animal                                  // Programmer1 的 Animal 类
    {
        public string type;                              //声明字段
    }
}
```

上述代码中，创建了同样的两个类 Animal 以及两个类成员 type。在主函数中，开发人员很难区分到底是使用哪一个类进行对象的创建和初始化，因为通常情况下，每个程序员可能只负责该程序员的模块或者代码，当整合的时候，代码就会变得难以调用或难以维护。

正如某个学校有两个班级，每个班级里都有一个叫小明的人。如果在早操大会上点名小明，那么这两个小明都不知道点的是谁，如果指定一下，说是一班的小明，那么二班的小明就不会认为是自己了。命名空间就起到了这个区分的作用，在主函数中，可以显式的对类进行访问，示例代码如下所示。

```
namespace Programmer1                                    //程序员 1 的命名空间
{
    public class Animal                                  // Programmer1 的 Animal 类
    {
        public string type;                              //声明字段
    }
}
namespace Programmer2                                    //程序员 2 的命名空间
{
    public class Animal                                  // Programmer2 的 Animal 类
    {
        public string type;                              //声明字段
    }
}
namespace MyClass                                        //主程序的命名空间
{
```

```
    class Program                                                    //主程序类
    {
        static void Main(string[] args)                              //主程序入口方法
        {
            Programmer1.Animal bird = new Programmer1.Animal();//说明是程序员 1 的命名空间下的 Animal
            bird.type = "bird";                                      //初始化字段
        }
    }
}
```

上述代码中很好的解决了类名称相同的情况下开发和维护的困难。在执行代码 Programmer1.Animal bird = new Programmer1.Animal()时，编译器就能够知道 Animal 类是属于命名空间 Programmer1 的，也就不会和命名空间 Programmer2 的 Animal 类混淆。

3.4.2 创建命名空间

程序开发中，创建和良好的使用命名空间，对开发和维护都是有利的，命名空间语法格式如下所示 。

```
namespace  命名空间
{
    类以及类成员
}
```

namespace 声明了一个命名空间，名称取命名空间的名称，在由“{”“}”大括号内引用的类成员来创建类。同样也可以创建两层或多层命名空间，示例代码如下所示。

```
namespace Programmer1                                    //单层命名空间
{
    public class Animal                                  // Programmer1 的 Animal 类
    {
        public string type;                              //声明字段
    }
}
namespace Programmer2                                    //双层命名空间
{
    namespace Programmer3                                // Programmer2 的命名空间
    {
        public class Animal                              // Programmer3 的 Animal 类
        {
            public string type;                          //声明字段
        }
    }
}
```

同样，命名空间成员也通过“.”号访问，例如访问双层命名空间的类的字段时，可以通过 Programmer2.Programmer3.Animal.type 进行访问。

3.4.3 分层设计中使用命名空间

从上一节中可以看出，命名空间的使用可以对相同名称的类进行较好的规范。但是，在同一层代码块中，似乎很少使用命名空间来规范。而在分层设计中，命名空间的使用是非常必要的，虽然初学者不

需要详细的了解分层设计，但是分层设计在软件开发过程当中是非常必要的，使用 Visual Studio 2008 能够轻松的创建分层构架软件。

在解决方案资源管理器中选择当前解决方案，右击【解决方案】项目，在下拉菜单中选择【添加】选项，在【添加】的下拉菜单中选择【新建项目】选项。若无法在解决方案管理器中看见解决方案，则可以在菜单栏的【工具】选项中选择【选项】菜单并在弹出窗口中找到【项目和解决方案】窗口，在窗口中选中【总是显示解决方案】复选框即可配置解决方案管理器，如图 3-2 所示。

为了能够在应用程序中进行分层开发，需要创建类库用于类的规划，创建一个“类库”项目，如图 3-3 所示。

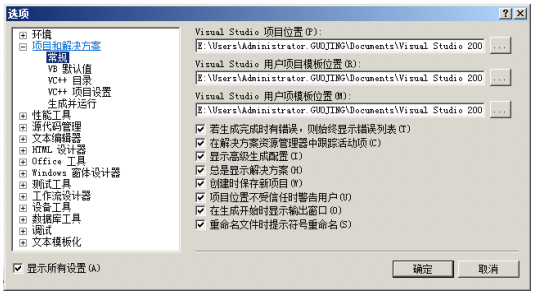

图 3-2　显示解决方案

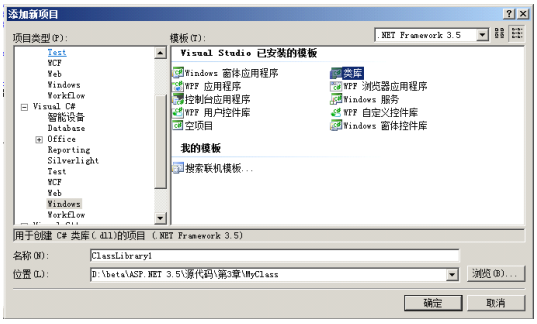

图 3-3　创建类库

输入项目名称，创建了类库后，命名空间就是创建的项目名称，系统会自动创建一个类 Class1，读者可自行修改类名称，并创建字段和方法，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                          //使用系统命名空间
namespace MyNamsSpace                                       //当前程序命名空间
{
    public class Class1                                     //当前类名称
    {
        public string output()                              //输出字符串方法
        {
            return "另一个项目的命名空间";                  //返回值
        }
    }
}
```

在原有的项目中，必须声明使用用户创建的命名空间，才可以访问命名空间类的类并创建对象。正如.NET 中为我们提供的系统命名空间一样，也是通过使用 using 来声明的。首先需要添加引用，右击项目，在下拉菜单中选择【添加引用】选项，在添加引用窗口选项卡上选择【项目】标签并引用相应类库，如图 3-4 所示。

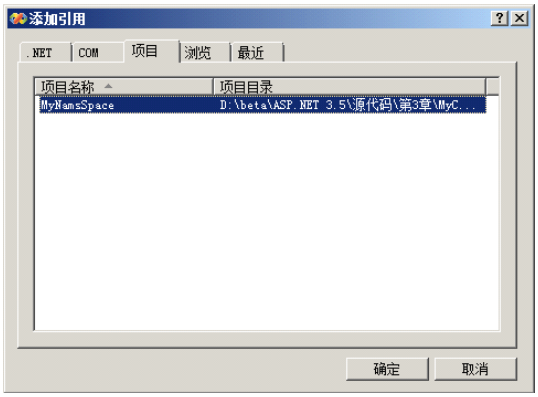

图 3-4　添加引用

单击【确定】按钮后，在代码头部书写 using 命名空间则可以使用类库项目的命名空间并访问方法和成员，代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                  //系统命名空间的使用
using MyNamsSpace;                                  //自定义命名空间的使用
namespace MyClass
{
    class Program
    {
        static void Main(string[] args)
        {
            Class1 myclass = new Class1();          //创建对象，该类完整名称是 MyNameSpace.MyClass
        }
    }
}
```

上述代码中引用了 MyNameSpace 命名空间，并访问了命名空间中的类，通过该类创建了一个对象。分层设计是软件设计中常用的设计方法，同设计模式一样，分层设计也是为了规范软件的开发和维护、降低软件开发成本、将软件模块化。但是过多的使用命名空间和分层设计也会造成层次过多无法维护的相反效果。

注意：当使用另一个命名空间的方法时，如果在本程序段中没有同名的方法，可以直接使用方法名称，而如果当前程序中包含了同名的方法，则需要指定命名空间名。

3.5　类的方法

创建了类，就需要创建类的字段，初始化字段。同样，创建了类之后也需要创建类的方法，来访问或者对字段进行操作。在类的对象的初始化后，对象能够使用方法进行对象的操作从而能够更加完整的描述一个对象（事务）。

3.5.1 编写方法

方法是指定名称的一组语句，每个方法都有一个方法名和一个方法体。方法名用来指定方法的名称，方法体用来描述该方法使用何种算法和结构来完成计算。对方法名的声明推荐具有一定的含义，例如GetUserPassword 的大意就是获取用户密码，这样其他的开发人员也能够读懂该函数的作用，提高了编码效率。方法的声明语法如下所示。

```
描述 权限 返回值类型 方法名称(参数)
{
方法体
}
```

上述伪代码说明了方法的声明语法，开发人员能够参照以上伪代码进行方法的编写，示例代码如下所示。

```
static public string GetUserName()                          //返回值类型为 string
{
    string username = "guojing";                            //定义字符串变量
    return username;                                        //返回 string 类型
}
```

编写一个方法必须按照方法声明的语法来编写，在编写方法时需要指定描述、权限、返回值类型等，这些必要的条件作用如下所示。

- 描述：用来描述方法，例如是否是静态方法等。
- 权限：权限用来描述方法的访问性，确定外部对象是否能引用或直接访问此方法或类成员。
- 返回值类型：返回值类型用来描述方法体中所返回的值的类型。
- 方法名称：描述方法的命名称。
- 参数：参数用来向方法传递参数，可以为 0 个或多个参数，多个参数用逗号分割。
- 方法体：用来描述方法所要执行的操作。

如果创建了一个方法，并且说明了该方法必须有返回值，则该方法的必须有返回值并且返回值的类型与声明的方法的返回值类型相同。若方法中的返回值和修饰符中返回值的修饰不相同，则编译器会报错，若方法无返回值，可使用 void 关键字修饰方法，示例代码如下所示。

```
static public void   GetUserName()                          //无返回值的方法
{
    string username = "guojing";                            //初始化字段
}
```

3.5.2 给方法传递参数

上一节中，了解了方法可以传递参数。通过参数的传递，开发人员能够知道方法的作用、方法的意义并通过传递参数对未知源代码的方法进行使用而无需阅读源代码。带参数传递的方法代码编写如下。

```
static public string   GetUserName(string name)            //有返回值有参数的方法
{
    string username = name;                                 //初始化字段
    return username;                                        //返回一个返回值
}
```

上述代码创建了一个包含参数传递的方法。当传递了一个参数，方法接收了变量值的拷贝，然后使用拷贝的变量值进行操作。

1．传递值

方法可以接收传递的值，并获取参数，示例代码如下所示。

```
class Program
{
    public void   GetUserName(string name)                    //包括参数传递的方法
    {
        string username = name;                                //内部变量赋值
    }
    static void Main(string[] args)
    {
        Program pro = new Program();                           //创建一个新对象
        pro.GetUserName("guojing");                            //传递了一个值
    }
}
```

上述代码创建了一个类，并在类中创建了一个 GetUserName 方法，包含一个参数传递。在创建了一个对象后，可以直接通过传递值来传递参数。

2．传递对象

方法也可以接收对象，对象也可以看作是一个变量进行参数传递，示例代码如下所示。

```
class Program
{
    public string name;                                        //声明共有变量
    public void   GetUserName(Program pro)                     //声明共有方法
    {
        string username = pro.name;                            //初始化字段
    }
    static void Main(string[] args)
    {
        Program pro = new Program();
        pro.name=”guojing”                                     //初始化字段
        pro.GetUserName(pro);                                  //传递了一个对象
    }
}
```

上述代码包含一个参数传递，该参数是一个对象。

3．this 关键字

this 关键字能够访问类成员，当参数名和类成员中字段名称相同时，可以使用 this 关键字，示例代码如下所示。

```
class Program
{
    public string name;                                        //声明共有变量
    public void   GetUserName(string name)
    {
        this.name = name;                                      //使用 this 关键字赋值私有变量
    }
    static void Main(string[] args)
    {
        Program pro = new Program();                           //创建一个新对象
        pro.GetUserName("guojing");                            //使用方法进行参数传递
```

```
    }
}
```

上述代码中，方法传递的参数名称为 name，同样类内有一个 name 为名称的字段，通过 this 关键字可以区分，this.name 就是类中的 name 字段。

3.5.3 通过引用来传递参数

通常情况下，方法只能返回一个值，但是在实际的开发当中，实际的情况可能需要返回多个值的方法。所以，传递一个变量的引用给一个方法即可实现。在平常使用的过程中，传递的参数在方法中是以参数的值的拷贝来运算的。当使用引用时，方法使用的是参数的实际数值，对参数的改变也会改变参数的实际值。

1．使用 ref 关键字

在描述一辆车的时候，可以创建一个 Car 类来描述车这个对象，示例代码如下所示。

```
class Car
{
    string carNumber="1001";
    string carType = "Car";
    public string GetCarNum(string num,string type)                    //创建方法
    {
        num = carNumber;                                               //获取私有变量的值
        type = carType;                                                //获取私有变量的值
        return num;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();                                         //创建一个新对象
        string number="0";                                             //初始化对象
        string type = String.Empty;                                    //String.Empty 初始化空字符串
        myCar.GetCarNum(number, type);                                 //改变字符串的值
        Console.WriteLine(number);                                     //输出
        Console.WriteLine(type);
        Console.ReadKey();
    }
}
```

上述代码运行结果中，输出的 number 依旧是 0，而 type 依旧是空字符串。上述代码并没有为多个变量参数进行更改，所以需要使用 ref 关键字来对多个对象进行更改，示例代码如下所示。

```
class Car
{
    string carNumber="1001";                                           //声明字符串变量
    string carType = "Car";                                            //声明字符串变量
    public string GetCarNum(ref string num,ref string type)            //使用 ref 关键字
    {
        num = carNumber;                                               //获取私有变量的值
        type = carType;                                                //获取私有变量的值
```

```
            return num;                                                    //返回字符串变量
        }
    }
    class Program
    {
        static void Main(string[] args)                                    //程序的主入口方法
        {
            Car myCar = new Car();                                         //创建一个新对象
            string number="0";                                             //将 number 赋值为 0
            string type = String.Empty;                                    //将 type 赋值为空
            myCar.GetCarNum(ref number,ref type);                          //使用 ref 关键字
            Console.WriteLine(number);                                     //输出字段
            Console.WriteLine(type);                                       //输出字段
            Console.ReadKey();                                             //等待用户按键
        }
    }
```

上述代码使用了 ref 关键字，运行后，number 和 type 的值都被改变了。在声明的方法参数前，必须使用 ref 关键字，非常重要的一点是，在使用该方法时，同样需要使用 ref 关键字修饰参数。

技巧：在这里 carNumber 并不是一个 int 型变量，原因是虽然汽车编号大部分是以数字显示的，但是其实是字符串，一辆车的编号为 1001 并不代表这辆车是第 1001 辆车或其他。

2．赋值明确

在 C#中，传递给方法的变量必须在传递前明确赋值，这样可以避免在执行方法时变量未赋值导致的不可预知的错误。因为有可能一个变量的值为 null，而 null 值不能转换到.NET 中的某些类型，例如上一小结的代码中，若不对 number 赋值为“0”，则编译器会报错。

3.5.4 方法的重载

在程序开发过程中，很多情况下，需要执行相同的函数体。例如在获取用户信息时，有的时候需要对用户的 ID 进行查询再获取用户信息，有的时候需要通过对用户的用户名来获取用户信息，可以用以下代码来实现。

```
    public void GetUserInfor(string name)                    //一个方法
    {
        //通过用户名获取用户信息
    }
    public void GetUserInforById(int id)                     //重载方法
    {
        //通过 ID 获取用户信息
    }
```

当一个功能需要加强的时候，就要写更多的方法。当这些方法的方法体大致上是相同的时，或者这当些方法体都能够表示这个方法名的意义时，就可以使用重载来增强代码的可读性。只要传递的参数不同，就可以重载函数，示例代码如下所示。

```
    public void GetUserInfor(string name)                    //原方法
    {
        //通过用户名获取用户信息
    }
    public void GetUserInfor(int id)                         //方法名相同但是参数不同
```

```
    {
        //通过 ID 获取用户信息
    }
```

上述代码创建了两个相同方法名的方法，但是传递的参数不同，编译器也可以通过编译。当调用方法时，指定具体的调用参数，则编译器会调用相应的方法，示例代码如下所示。

```
UserInformation user = new UserInformation();                    //创建对象
user.GetUserInfor("guojing");                                    //调用第一个方法
user.GetUserInfor(115);                                          //调用重载方法
```

当声明对象后，对象使用方法，根据传递的参数不同，编译器会自动识别，如传递的参数为字符串变量 guojing 时，则会使用 GetUserInfor(string name)方法。同样，如传递的参数为整型变量 115 时，则会使用 GetUserInfor(int di)方法。在类设计中，可以将一些具有相同功能的方法设计为重载方法。使用重载的情况如下所示：

- 设计一些相同的方法时，如果只是参数不同，则使用重载。
- 如果为程序添加一个新功能，重载一个方法是不错的选择。
- 如要实现类中相似的功能，则可以考虑使用重载。

3.6 封装

在 C#中，封装就是将类成员中的字段、方法以及属性事件、委托等放在一个公共的结构中。按照一个公共的方法把数据和操作这些数据的方法进行组装（封装），同时为对象指定操作和属性，从而创建了新的数据类型提供给用户使用，而保证了私密的内容不会被用户察觉。

3.6.1 为什么要封装

在应用程序开发，特别是面向组件开发的过程中，常常会将类成员中的字段、方法、属性及事件封装在一个类或命名空间内。当把数据和方法封装后，就可以指定方法和属性，对于外部使用者而言，他们无需知道类是怎样设计的，只需要关心如何实现信息，让用户成为类的使用对象。正如在.NET 中的 System.Web 命名空间或者类型转换 Convert.ToInt32 一样，开发人员知道这个命名空间或者这个类的静态方法是做什么的，却不知道这个方法内部的代码。

这些方法的内部代码对开发人员是封闭的，保证内部代码的隐私和安全，开发人员只需要使用方法或者覆盖方法来达到自己编程的目的。例如一台电脑有显示器、主机等等外设，而相对于显示器而言，显示器是一个类，同样，机箱也是一个类，同样这些来还包括鼠标，键盘等。而显示器内部和机箱内部对一般的用户是不可见的，因为用户不知道怎样拆装显示器和机箱，但是用户知道怎样将插座相连，组成一个完整的计算机。封装能够让用户更加关注“计算机”本身，而不去关注“计算机”内部是怎样实现的。

这种抽象方便了封装人员便于改变内部实现细节，在改变了内部细节之后，用户通常情况下不会感到有任何的差异。

注意：例如.NET 中，在.NET 从 1.1 到 2.0 然后到 3.X，很多的细节被改变，因为类或方法有了改变。但是大部分情况下，很多的细节改变了，而使用的方法没有改变，特别是从 2.0 到 3.5。

3.6.2 类的设计

在设计一个类时，应该尽量的隐藏细节，只暴露那些开发人员或者类使用者需要知道的操作和数据。这样就方便了代码的维护，实现了在使用者无需改变与类成员交互方式的前提下，对类的实现细节进行更改。

在 C#中，没有对类成员或方法的公共属性进行设置，则默认的权限是 private（私有的）。而在类的设计中，尽量将使用的类成员设置为私有的权限，因为这样保证了代码的安全性，也让使用者更加关心怎样与类成员进行交互。而可以对一些共有的属性或者与方法交互的一些字段设置成为 public（共有的），除非认为字段是有用的，否则一般不予暴露。这种方式不仅保护了类成员，同样也让维护变得简单，示例代码如下所示。

```
class Car                                                    //创建一个类
{
    public void GetCar()                                     //创建类的方法
    {
        int i = 1;                                           //声明整型变量
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();                               //使用类创建对象
        myCar.GetCar();                                      //使用了对象的一个方法
    }
}
```

上述代码创建了一个汽车（car）类，在主方法中使用类，同样，当对 Car 类的类成员做修改时，不会影响到用户的使用，示例代码如下所示。

```
class Car                                                    //创建一个类
{
    public void GetCar()                                     //创建类的方法
    {
        int i = 1;                                           //声明变量
        int j = 0;                                           //更改了类内部的实现细节
    }
}
class Program
{
    static void Main(string[] args)
    {
        Car myCar = new Car();                               //创建对象
        myCar.GetCar();                                      //使用方法依旧没有改变
    }
}
```

同样，在对类进行了封装之后，用户不知道类是怎样实现的，但是同样能够创建类或对象，这如前面的章节中命名空间一样。通常情况下开发人员会建立一个类库，使用者需要引用类库，并调用类库中的代码，而无需知道类库中的代码是怎样实现的，示例代码如下所示。

```
class Program                                                //应用程序主类
```

```
{
    static void Main(string[] args)                    //应用程序入口方法
    {
        Car myCar = new Car();                         //使用 Car 类，而使用者并不知道细节
        myCar.GetCar();                                //使用方法
    }
}
```

3.7 属性

在 3.2.3 类成员一节，简单的讲解了属性作为类成员的使用方法。属性是为了把对象的实现细节与使用者所能看见的元素隔离，同时通过定义类成员的作用域，从而控制关于对象数据的访问。虽然在前面的章节中，可以通过 private、public 等关键字限制类成员的访问权限，但是通过属性可以管理其他对象对类中类成员的访问。属性是一种类成员，提供了对对象或类中元素的访问方法。

3.7.1 语法

定义属性的语法通常使用 public、private 来修饰访问权限，同样包括类型、属性名、关键字以及由“{”“}”大括号包含的代码段，示例代码如下所示。

```
    public string Str { get; set; }                    //编写属性
```

get 语句和 set 语句成为访问器。在 Visual Studio 2005 中，get 访问器的返回类型必须与属性类型相同，或者可以隐式的转换为属性类型。set 访问器等价于具有一个叫 value 的隐式参数的方法。而在 Visual Studio 2008 中，可以使用更加简单的代码，如上述代码所示。

1．类成员实现

通过设计类成员，可以实现属性的原理，也能够更加方便的理解属性的实现，示例代码如下所示。

```
class Car                                              //编写类
{
    private string _str;                               //私有的类成员，字段
    public string SetStr(string str)                   //共有的方法，设置类成员
    {
        _str = str;                                    //设置相应属性
        return _str;                                   //返回相应属性
    }
    public string GetStr(string str)                   //共有的方法，返回私有类成员值
    {
        return _str;                                   //返回相应属性
    }
}
```

上述代码创建了一个汽车类，用来描述一辆汽车，同时也创建了一个私有的类成员_str，用来描述车的属性。当设置为私有时，对象不能通过“.”号来访问_str，于是需要通过类成员中的方法来实现访问。SetStr 是一个为_str 赋值的一个方法，代码非常简单，而 GetStr 是获取_str 值的方法，而这两个方法都公共方法，可以通过外部引用。

2．属性的使用方法

从上述代码中可以看出，属性的使用无非就是为私有的变量的访问权限做了调整，通过属性来访问用户本不应该访问的私有变量。可以将上述类更改，使代码更加简洁和可读，示例代码如下所示。

```
class Car                                                    //编写类
{
    public string str { get; set; }                          //使用属性声明字符串变量
}
class Program                                                //应用程序主类
{
    static void Main(string[] args)                          //应用程序入口方法
    {
        Car myCar = new Car();                               //创建新变量
        myCar.str = "Car";                                   //访问类成员并赋值
    }
}
```

上述代码使用了属性，让代码更加可读。可以将属性的方法对用户隐藏，提高了代码的安全性，也增加了封装的特性。

注意：在 2.0 中，get 需要返回一个值，并且与 get 访问器的返回类型必须相同。而 set 访问器中，需要写变量名称的值等于 value 的隐式参数方法，但是在 3.5 中去掉了这样的声明方式，直接写 get;set; 即可实现。

3.7.2 只读/只写属性

只读属性和只写属性是 C#属性中的一个特殊属性，只读属性和只写属性都只包含相应的访问器用于不同的属性的访问。只读属性和只写属性在一定程度上保证了属性的安全性和类的密封性，如果在应用程序开发中为了保证某个字段的安全性或可访问性，可以使用只读或只写属性。

1．只读属性

在属性的声明中，允许声明只包含一个 get 访问器的属性。在这种情况下，声明的类成员只允许读，并在程序上下文中使用，示例代码如下所示。

```
class Car                                                    //编写类
{
    public string str { get {return "this is a str";}}      //只读属性
}
```

上述代码中只使用了 get 访问器，在使用中，类成员变量 str 只能读而不能改。

2．只写属性

同样，在属性的声明中，同样允许声明只包含一个 set 访问器的属性。在这种情况下，声明的类成员只允许写，但是却不能读，在程序的上下文中不允许使用，示例代码如下所示。

```
class Car                                                    //编写类
{
    private string _str;                                     //声明私有变量
    public string str { set { _str = value; } }              //只写属性
}
```

上述代码中只使用了 set 访问器，在使用中，类成员变量 str 只能写而不能读。

3.8 继承

在类的设计中，经常需要管理和开发相似功能。在设计这些类的时候，就可以使用继承的原则。在面向对象的应用程序开发中，允许创建一个抽象的类而不实现其具体方法，而需要通过继承、派生来实现方法。这样不仅优化了代码，提高了代码的可读性，而且在开发过程中，也让开发人员有比较明确的编码思想，使开发人员与开发人员之间更加容易协调。

3.8.1 继承的基本概念

在应用程序开发过程中，需要完成功能相近但是实现不同的类来抽象对象的时候，就需要用到继承。例如，要开发一个应用程序或者网站来统计全球动物的种类和基本信息的时候，就需要创建一些类，比如人类、猫、狗等等。相比之下，人类和猫和狗有相似之处，如它们都是哺乳动物，说的更高一点，就都是生物。而人，猫和狗却是不同的对象，还必须要区分三者之间的关系，错误的类设计代码如下所示。

```
public class Animal                                          //编写类
{
    public string type;                                      //动物的种类
    public string color;                                     //动物的颜色
    public string sound;                                     //动物叫的声音
}
class Program                                                //应用程序主类
{
    static void Main(string[] args)                          //应用程序入口方法
    {
        Animal cat = new Animal();                           //创建对象
        cat.type = "cat";                                    //猫科动物
        cat.sound = "miaomiao";                              //猫的叫声为 miaomiao
        Animal person = new Animal();                        //创建对象
        person.type = "person";                              //人类
    }
}
```

上述代码中，并没有语法错误，但是有逻辑误差或者说是开发困难。因为，如果要形容人类的国家，那么就得在 Animal 类中增加一个 country，但是猫却没有国家之分，这样就让整个类变得非常的混乱且庞大臃肿，使用派生就能解决这样的问题。派生类的优点如下所示。

- 提高重用性：派生提高了代码的重用性，不至于在创建一个新对象时再重新写一个新的类。
- 提高结构性：派生让程序有了结构，在程序开发过程，每一个派生类均继承上一个类的方法，且每个派生类除了可以使用公共的字段以外，可以专门为派生类增加字段和方法而不去影响到其他的派生类。

上述代码中可以看出，在设计类的时候，对于同一个类的重复使用，可能会使用户非常的迷惑。例如猫是派生自动物的，但是用户会希望猫的类名是 Cat，而同样，当创建一个人的类的时候，通常情况下用 Animal 描述是不太适合的，虽然人和猫都是属于动物，但是这样的表述往往让人迷惑不解，而派生类可以更好的实现。

3.8.2 创建派生类

通过使用派生类，可以让程序的代码的意义更加的明确和容易阅读，可以通过改造派生类去实现更多的方法，而不用修改基类，以免影响到其他的派生类。在.NET中，当创建一个应用程序或者是Web Form应用程序时，其实都已经默认派生自一个系统提供的基类。而派生可以允许派生用户自定义的类，示例代码如下所示。

```
public class Animal                                    //创建基类
{
    public string type;                                //创建基类成员
    public string color;                               //创建基类成员
    public string sound;                               //创建基类成员
}
public class People:Animal                             //创建派生类
{
    public string country;                             //创建派生类成员
}
```

使用“:”运算符说明该类是派生自一个基类，上述代码创建了一个新的类Person来描述人类这种高级动物，“:”运算符说明了People类派生自Animal。

创建了派生类，说明了该派生类继承了基类的共有或保护的方法和属性，在派生类中可以无需在声明变量。例如上述代码中，为People类增加了country字段来描述人类的国家。而人类同样有声音、肤色等，这些基类已经提供，就可以不需要再重新声明了，可以直接通过“.”运算符使用，因为基类的字段或方法已经被“继承”了，示例代码如下所示。

```
public class Animal                                    //创建基类
{
    public string type;                                //创建基类成员
    public string color;                               //创建基类成员
    public string sound;                               //创建基类成员
}
public class People:Animal                             //创建派生类
{
    public string country;                             //创建派生类成员
}
class Program
{
    static void Main(string[] args)
    {
        People person = new People();                  //创建派生类对象
        person.country = "China";                      //初始化派生类字段
        person.color = "yellow";                       //使用基类字段
    }
}
```

上述代码中，People类并没有声明字段color，但是People类是继承自Animal类的，而Animal类包括字段color，并且该字段是共有的，所以在People类中也可以使用color。

注意：当基类的字段或者方法等访问修饰符为public或protected时，继承的派生类可以使用基类的字段或方法。但是当基类的字符按或方法等访问修饰符为private时，继承的派生类不能使用基类的private字段或方法。

3.8.3 对象的创建

当创建一个派生类的对象，派生类的对象可以使用基类中共有（public）或保护（private）的类成员。之所以派生类的对象能够使用基类的类成员，是因为在创建派生类的对象的时候，首先会执行基类的构造函数，然后再执行派生类的构造函数，最后一个对象才会被创建，示例代码如下所示。

```
public class Animal                                          //编写类
{
    public Animal()                                          //编写构造函数
    {
        Console.WriteLine("Animal 被构造");                  //Animal 构造函数
    }
}
public class People:Animal                                   //编写派生类
{
    public string country;                                   //声明变量
    public People()                                          //编写构造函数
    {
        Console.WriteLine("People 被构造");                  //People 构造函数
    }
}
class Program                                                //应用程序主类
{
    static void Main(string[] args)                          //创建主方法
    {
        People person = new People();                        //创建一个 Person 对象
        Console.ReadKey();                                   //等待用户按键
    }
}
```

运行结果如图 3-5 所示。

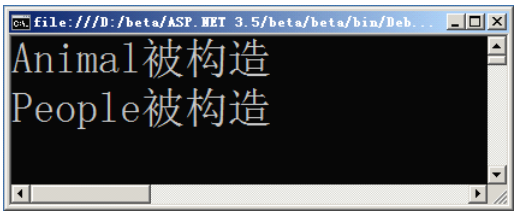

图 3-5　派生类的构造

从上述代码中可以看出，当创建了 Person 类的对象的时候，虽然没有创建 Animal 类的对象，但是还是构造了 Animal 类。Animal 的构造函数执行后，才开始执行 Person 的构造函数。当执行了 Person 构造函数，势必会执行 Person 基类的构造函数，如果一个派生类的基类有多个构造函数，而开发人员想指定构造函数时，必须使用 base 关键字，Animal 类示例代码如下所示。

```
public class Animal                                          //编写类
{
    public Animal()                                          //编写构造函数
    {
        Console.WriteLine("Animal 被构造");                  //构造函数
    }
```

```
        public Animal(DateTime time)                                    //另一个构造函数
        {
            Console.WriteLine("Animal 在" + DateTime.Now.ToString() + "被构造");//另一个代码块
        }
    }
```

上述代码中，Animal 类具有两个构造函数，而要让派生类使用基类的某个构造函数，就必须指定派生类使用的基类的构造函数的方法，示例代码如下所示。

```
    public class People:Animal                                          //编写派生类
    {
        public string country;                                          //定义字符串型字段
        public People(DateTime time):base(time)                         //指定构造函数
        {
            Console.WriteLine("People 被构造");                         //编写构造函数
        }
    }
```

在指定了构造函数后，在主函数中也必须修改相应的对象的创建代码，示例代码如下所示。

```
        static void Main(string[] args)                                 //主入口方法
        {
            People person = new People(DateTime.Now);                   //使用指定的构造函数
            Console.ReadKey();                                          //等待用户按键
        }
```

运行结果如图 3-6 所示。

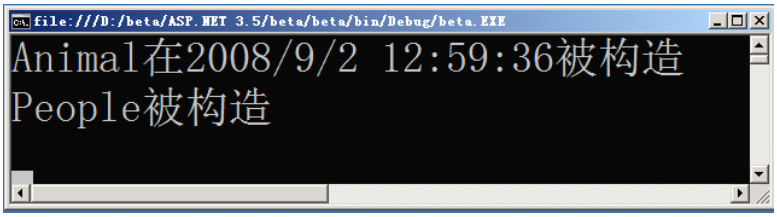

图 3-6　指定使用基类的构造函数

3.8.4　使用抽象类

在类被创建的时候，派生类的构造函数在执行前，会首先执行基类的构造函数。当声明或者设计一个类的结构时，基类往往是不完善的，也不应该把基类的类成员实例化。直接从 Animal 类创建对象是不正确的，因为基类的作用是为了整合派生类中公共的相同或相似的属性或字段，而对基类的成员赋值或者创建基类的对象，会使类的结构变得混乱。

例如在上面的例子中，Person 类从 Animal 类派生，当开发人员希望创建一个对象的时候，无论是从代码还是命名都能够更好的理解。虽然很多场景会不会使用人类的国家参数，但是从 Person 类创建对象还是比从 Animal 类创建对象更加好理解，示例代码如下所示。

```
            People abc = new People();                                  //创建对象
            Animal abcd = new Animal();                                 //创建对象
```

从上述代码中，阅读代码时，能够非常清楚的了解 abc 是一个人，而不是一只猫或者一只狗。而 abcd 却让人匪夷所思，只知道此对象是动物，而不知道具体是什么。虽然上述的例子中，abc 没有任何意义，在这里只是使用了开发中的某个场景，当代码过长的时候，命名也会变得困难，也会很难找到合适的命名，但是还是推荐使用有意义的名称来定义变量，例如 person。这里只是作为例子来说明创建对象时，

好的类设计带来的好处。

创建对象时，如果对基类使用 abstract 关键字，那么编译器会阻止基类的直接实例化，从而可以强制的让开发人员使用正确的类让类层次结构正确并容易阅读。当用户创建 Animal bird=new Animal()时，编译器会报错并提示错误，示例代码如下所示。

```
public abstract class Animal                                    //创建抽象类
{
    public Animal()                                             //创建构造函数
    {
        Console.WriteLine("Animal 被构造");                     //编写构造函数
    }
}
```

当使用上述代码进行类的对象的创建时，系统会提示错误，因为该类是一个抽象类，在抽象类的基类中，系统是禁止基类的直接实例化的。

3.8.5 使用密封类

与抽象类相反的是，C#支持创建密封类，密封类是一种永远不能做基类的类。其他的类不能从此类派生，从而保证了密封类的密封性和安全性，使用 sealed 关键字能够创建密封类，示例代码如下所示。

```
public sealed class Animal                                      //创建密封类
{
    public Animal()                                             //创建构造函数
    {
        Console.WriteLine("Animal 被构造");                     //编写构造函数
    }
}
```

当 Person 再次从 Animal 派生时，编译器会提示出现错误，Person 类无法从密封类 Animal 派生。

注意：设计类的时候，通常情况下是不需要将类设置为密封类的，因为密封类会让类的扩展性非常的差，这个类也无法再次扩展和派生。但是，出于某种目的，当程序块只需要完成某些特定的功能或者在商业上为了保密，则可以使用密封类对类进行密封，保证类的可靠性。

3.9 多态

面向对象应用程序开发中，与传统的面向对象不同的是，面向对象具有很多的特性让开发变得简单和方便，代码便于阅读和维护，多态也是其中的重要的特性。多态可以分为两种，分别为动态多态和静态多态。上面章节中讲到的重载是多态的一种，重载是一种静态多态。

3.9.1 抽象方法

抽象方法是一个没有对类成员中方法进行具体实现的一种方法，抽象方法的实现必须让派生类实现。虽然抽象类中的所有方法不一定全是抽象方法，但是包含抽象方法的类被称作抽象类。抽象类可以使用 abstract 修饰符修饰类名称，抽象类中抽象方法的实现语法如下所示。

```
public abstract class Animal                                    //编写类
{
```

```
    public abstract string Sound();                                    //创建抽象方法
}
```

上述代码中创建了抽象类 abstract class Animal，同时创建了抽象方法 Sound()。抽象方法不允许有方法体，同样不允许包含括号，只允许声明抽象方法。在派生类中，必须实现基类中的抽象方法，示例代码如下所示。

```
public class People:Animal                                             //派生自 Animal 类
{
    public string country;
    public override string Sound()                                     //实现抽象方法
    {
        return "language";                                             //返回值
    }
}
```

在派生类中，为了实现抽象的方法，就必须使用 override 关键字，来表示此方法是对基类的抽象方法的实现。使用抽象方法的好处在于，一位开发人员（可以是开发小组的组长或者软件构架设计师）可以创建一个或多个基类，来划分模块，或者按照功能划分和设计类，而小组的其他成员可以通过派生类来对基类进行实现，而设计基类的人员无需对类中方法的细节进行关心。同样，当修改代码时，也无需对基类进行修改，直接对派生类修改，防止当多个派生类派生于同一个基类时，出现不可预料的错误。

注意：Sound 方法是一个抽象方法，在其他派生类中，如猫、狗，都有自己独特的叫声，而人类使用的是语言。不同的种别实现的方法千差万别，在基类中规定将属性规定死是非常不明智的做法。

3.9.2 覆盖

当基类创建了一个方法来描述类对象的时候，派生类的同一方法必须实现不同的细节，例如动物类中，初始化了方法中的声音的方法为“miaomiao”，那么在派生类中，可能声音的方法不是“miaomiao”，那么就可以对基类的方法进行覆盖，示例代码如下所示。

```
public class Animal                                                    //创建基类
{
    public string Sound()                                              //基类中的 Sound 方法
    {
        return "miaomiao";                                             //返回值
    }
}
public class People:Animal                                             //创建派生类
{
    public string country;                                             //创建私有成员
    public string Sound()                                              //覆盖派生类中的 Sound 方法
    {
        return "language";                                             //返回值
    }
}
```

派生类中使用了 Sound 方法，而基类中同样使用了此方法。当编译器编译代码的时候，会将派生类中的方法覆盖基类的方法，并在派生类中对象运行的时候，执行派生类中的方法，而不会执行基类中的方法。

注意：虽然派生类可以覆盖基类的方法，但是在设计类的时候，推荐使用抽象类或者接口来实现基类，因为这样便于阅读和维护，也提高了代码的安全性。

3.9.3 虚方法的抽象类

在类的设计中，可以使用 abstract 关键字修饰类为抽象类，那么在基类中就不需要实现抽象类，而必须在派生类中实现基类的方法。但是，如果在基类中，有多个方法来形容一个动物的特性，如飞行的特性可以用来形容鸟，而世界上很多动物都会飞行，但是人类是无法飞行的，所以在人类这个派生类中实现飞行的方法是没有必要，也是降低的代码的可读性的。这里就可以通过 virtual 关键字实现虚方法，让派生类能够选择是否实现该方法，示例代码如下所示。

```
public class Animal                                          //创建类
{
    public virtual string Fly()                              //虚方法，飞行方法
    {
        return "Most Of The Animal Can Fly";                 //返回值
    }
}
public class People:Animal                                   //创建派生类
{
    public string country;                                   //没有实现虚方法也可以
}
public class Bird : Animal                                   //创建派生类
{
    public string FLY()                                      //鸟儿能飞行，实现一个虚方法
    {
        return "It Can Fly";                                 //返回值
    }
}
```

上述代码中，人类不能飞行，所以没有必要实现 Fly()方法，而鸟儿可以飞行，为了更好的描述一个对象，可以实现 Fly()方法让对象能够飞行。

技巧：在类设计中，对于多数的派生类使用的方法，可以考虑将方法放置在基类中，当一个方法，例如飞行，有一些派生类不需要使用，如人类，则可以将此方法设置为虚方法。而当一个方法，例如吃东西，是每个派生类都必须使用的，则最好将此方法设计为抽象方法，强制每个开发人员必须实现该方法。

3.9.4 抽象属性

同方法相同的是，属性也可以使用抽象属性。同样，基类的派生类也必须实现基类的抽象属性的方法，示例代码如下所示。

```
public class Animal                                          //创建基类
{
    public string Sound{get;set;}                            //创建 sound 属性
}
public class People:Animal                                   //创建派生类
{
    public string country;
    public string Sound { get; set; }                        //覆盖基类属性
}
```

当抽象属性为只读或只写属性时，可以移除相应的访问器简化代码并提高相应字段的安全性。

3.10 委托和事件

委托让初学者觉得非常的疑惑和困难，委托其实就是一种引用方法的类型。委托通常与事件一起使用。通常情况下，如果为委托分配了方法，委托和声明的方法具有完全相同的功能，在 ASP.NET 应用程序的开发中，服务器控件就使用了委托和事件的思想进行了开发。

3.10.1 委托

委托的方法和其他所有的方法一样，具有参数以及返回值，委托用关键字 delegate 修饰，示例代码如下所示。

```
public delegate string MySound(string sound);                    //声明一个委托
```

委托是一种安全的类型，并将方法安全的封装。在 C/C++应用程序开发中，C#中的委托和 C++的函数的指针类型类似，而稍有不同的是，C#中的委托是面向对象、类型安全的。委托的类型由委托的名称定义，如上述代码所示，代码中声明了 MySound 委托。

在委托对象被创建的时候，通常情况下提供委托包装的方法或匿名方法。实例化委托后，委托将把对它进行的方法调用传递给方法，委托允许将方法作为参数进行传递并定义回调的方法，调用方传递给委托的参数被传递给方法，相应的委托的方法的返回值返回给调用方，示例代码如下所示。

```
class Program                                                    //应用程序主类
{
    public delegate void MyDel(string message);                  //声明一个委托
    public static void DelegateMethod(string message)            //声明调用委托
    {
        Console.WriteLine("message is: " + message);             //输出字串
    }
    static void Main(string[] args)                              //应用程序入口方法
    {
        MyDel del = DelegateMethod;                              //注意这里初始化
        del("Delegate Method");                                  //使用委托
        Console.ReadKey();                                       //等待用户按键
    }
}
```

当创建委托并实例化委托对象时，委托把对它进行的方法调用传递方法。上述代码中，将 DelegateMethod 方法传递，在实例化后，当使用委托时，调用方（del）传递给委托的参数（Delegate Method）被传递给了方法（DelegateMethod），实现了方法，并调用、执行方法，运行结果如图 3-7 所示。

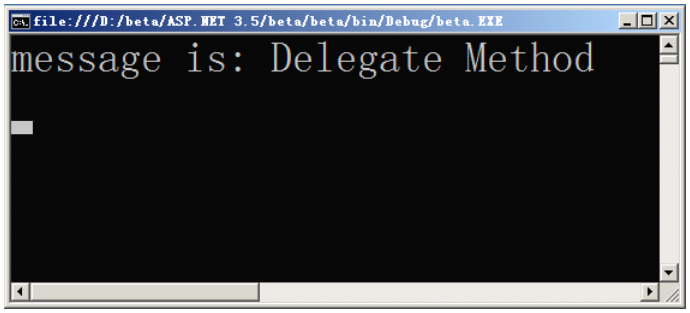

图 3-7 委托的定义和使用

3.10.2 声明事件

在 3.2.3 小结中，讲解了事件的基本概念，事件具有以下特点：

- 事件通常使用委托事件处理程序进行声明。
- 事件始终通知对象消息并指示需要执行某种操作的一种方式。
- 发行者确定何时引发事件，订阅者确定执行何种操作来响应该事件。
- 一个事件可以有多个订阅者。一个订阅者可处理来自多个发行者的多个事件。
- 没有订阅者的事件永远不会被调用。
- 事件通常用于通知用户操作，例如，图形用户界面中的按钮单击或菜单选择操作。
- 如果一个事件有多个订阅者，当引发该事件时，会同步调用多个事件处理程序，也可以使用异步处理多个事件。

事件和方法一样，通常事件和方法一起使用，事件和委托一样具有签名，但是事件的签名通过委托类型来定义，示例方法代码如下所示。

```
public delegate void MyDel(object sender, EventArgs e);                    //声明一个委托
public class Event                                                          //编写事件类
{
    public event MyDel EventTest;                                           //声明一个事件
    public void EventTestMethod()                                           //编写事件方法
    {
        Console.WriteLine("事件被使用");                                    //输出字串
    }
}
```

上述代码中，声明了一个委托，声明委托后，在 Event 类中声明了一个事件，这个事件绑定到委托 MyDel。在事件的签名中，第一个参数为引用事件源的对象，第二个参数为一个传送与事件相关的数据的类。在 C#中，规范的代码编写能够让代码更具可读性。

3.10.3 引发事件

如果要引发事件，类可以调用委托，并传递所有与事件有关的参数。上面的章节中讲到，事件通常和委托一起使用，并且通过给委托发送信息，来引发事件。如果该事件没有任何处理程序，则此事件为空，所以在引发事件之前，必须先确定该事件不为空，否则会抛出 NullReferenceException 异常，引发事件代码如下所示。

```
public delegate void MyDel(object sender, EventArgs e);                    //创建委托
public class Event                                                          //编写事件类
{
    public event MyDel EventTest;                                   //声明一个事件
    public void EventTestMethod()                                   //声明一个事件所执行的方法
    {
        MyDel OnLoad = EventTest;                                   //声明事件的方法
        if (OnLoad != null)                                         //判断事件是否为空
        {
            OnLoad(this, EventArgs());                              //不为空则使用委托
        }
    }
}
```

每一个事件都可以分配多个程序来接收该事件，这样，事件自动调用每个接收器，当有多个接收器时，引发事件只需要调用一次事件。

3.10.4 订阅事件

事件可以像一个方法一样，若要接收某个事件的类，可以创建一个方法来接收该事件，接收事件的类像类事件自身添加方法的委托，这个被称作“订阅事件”，可以说，和平时上网中的 RSS 订阅整个过程很像。值得注意的是，接收器必须具有与事件自身相同的签名的方法，然后该方法才能采取适当的操作来响应事件，示例代码如下所示。

```
public class EventReceiver                                          //创建一个接收器
{
    public void EventTestReceiver(object sender, EventArgs e)      //方法的签名必须相同
    {
        Console.WriteLine("从" + sender.ToString() + "引发了一个事件"); //执行方法体
    }
}
```

每一个事件可以分配多个程序来接收该事件，也就是说可以有多个接收器。多个接收器由源按照顺序调用。如果一个接收出现异常，则没有接收的接收器会接收事件。如果要订阅事件，接收器必须创建一个与事件具有同种类型的委托，并使用事件处理委托的目标，这也就是为什么事件通常情况下会与委托一起使用。示例代码如下所示。

```
public void EventTestSubscribe(Event eve)
{
    MyDel del = new MyDel(EventTestReceiver);                       //声明委托
    eve.EventTest += del;                                           //增加事件
}
```

上述代码中，通过“+=”运算符订阅了一个事件，同样，也可以使用“-=”号取消订阅。示例代码如下所示。

```
public void EventTestSubscribe(Event eve)
{
    MyDel del = new MyDel(EventTestReceiver);                       //声明委托
    eve.EventTest -= del;                                           //取消事件
}
```

3.10.5 委托和事件

上面几节中分开讲解委托和事件，对于初学者而言，委托和事件是很难学习的知识，但是当学习过委托和事件之后，会发现委托和事件非常的简单。在 ASP.NET 开发当中，很多控件都使用了委托和事件。例如当单击一个按钮控件时，按钮会发送信息指示“引发了一个按钮事件”，然后发送给相应的接收器，接收器接收了发来的信息从而引发相应的操作。在了解委托和事件的基本概念后，下列代码说明了怎样一步步的使用委托和事件。

为了实现广播喇叭功能（类似 QQ 的聊天窗口的系统信息），应用程序中不仅有用户的聊天窗口，也包括系统发送窗口。系统可以给用户的聊天窗口发送系统信息，在应用程序中，不仅需要广播用户的信息，同样系统也能够广播系统信息。为了实现这一功能，首先，需要创建一个委托，示例代码如下所示。

```
        public delegate void BetaDel string str);                                    //创建一个委托
```

在创建了委托后，就要为写方法，示例代码如下所示。

```
        public delegate void BetaDel(string str);
        public static void Output(string str)                                        //用户发送信息方法
        {
            Console.WriteLine("用户发送给你一个消息");                                  //输出用户提示信息
        }
        public static void SystemOutput(string str)                                  //系统发送信息方法
        {
            Console.WriteLine("系统发送给你一个消息");                                  //输出用户提示信息
        }
        public static void OutputChoose(string str,BetaDel del)                      //使用委托变量
        {
            del(str);
        }
```

注意：在上述代码中，del 是一个委托变量，del(str)会按照方法的签名在委托的方法表中执行。上述代码，与 del(string str)签名相同的方法有 Output 和 SystemOutput，他们的方法签名相同。

在主函数中，可以通过委托来使用方法，示例代码如下所示。

```
        static void Main(string[] args)
        {
            OutputChoose("你好", Output);                                            //通过传递方法名称来使用方法
            Console.ReadKey();
        }
```

上述代码中，使用了 OutputChoose 方法。值得注意的是，在 OutputChoose 方法中，其中的一个参数是方法名称。因为通过委托，可以将方法名称作为参数进行传递，从而执行了相应的方法。值得注意的是，在上述代码中，委托等方法全部都声明在一个类中，因为这样能够方便理解，但是这样就不具备面向对象的特点，面向对象的特性就是封装，封装能让代码具有结构性，于是可以使用事件。创建一个类，类名称叫 OutputChoose，示例代码如下所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;                                          //使用系统命名空间
namespace beta
{
    class OutputChoose
    {
        public string message="你有新短消息,请注意查收";     //声明短消息字串
        public delegate void BetaDel(string str);           //定义委托注册事件
        public event BetaDel MyEvent;                       //声明事件
        public void OnLoad()                                //编写 OnLoad 方法注册事件
        {
            if (MyEvent != null)
            {
                MyEvent(message);                           //当存在事件时，调用所有注册对象的方法
            }
        }
    }
}
```

上述代码将前面代码中的方法进行了封装作为委托。然后添加一个用户消息类，类名为UserMessage，示例代码如下所示。

```
using System.Linq;
using System.Text;                                          //使用文本处理命名空间
namespace beta                                              //声明当前程序命名空间
{
    class UserMessage
    {
        public void Output(string str)                      //输出方法
        {
            Console.WriteLine("用户发送给你一个消息:" + str);    //简单的输出
        }
    }
}
```

再添加一个系统消息类，类名称为 SystemMessage，示例代码如下所示。

```
using System.Linq;
using System.Text;                                          //使用文本处理命名空间
namespace beta                                              //声明当前程序命名空间
{
    class SystemMessage
    {
        public void SystemOutput(string str)                //系统获取输出方法
        {
            Console.WriteLine("系统发送给你一个消息:" + str);//显式系统发送的消息
        }
    }
}
```

在主函数中，可以触发事件来，示例代码如下所示。

```
        static void Main(string[] args)
        {
            OutputChoose opc = new OutputChoose();          //声明一个类的对象
            SystemMessage msg = new SystemMessage();
            opc.MyEvent += msg.SystemOutput;                //注册方法
            opc.OnLoad();                                   //开始自动调用所有注册的方法
            Console.ReadKey();
        }
```

上述代码中，OnLoad()触发了之前注册的事件，并执行事件，运行结果如图 3-8 所示。

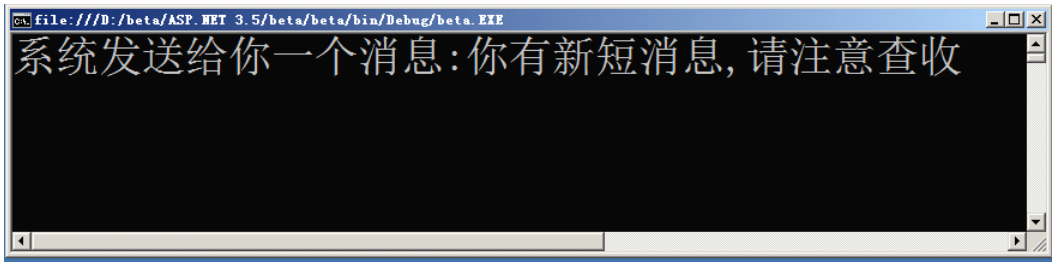

图 3-8　委托和事件的综合用例

运行结果显示，当创建了一个对象，对象可以注册声明事件，因为该对象没有实现该事件的方法的具体实现，但是在事件中增加了方法，类似于在该类中增加了一个方法，而在该类的编码实现中，定义

了一个 OnLoad 方法来调用所有注册对象的方法。

3.11 类命名

.NET 框架系统中类的命名总是包含着各种含义，无论是命名空间还是类甚至是变量。良好的命名规范这能够让使用它的人非常容易理解并方便阅读和使用。在系统开发中，对于程序开发人员而言，也推荐统一并按照一定的规范来命名，这用同样为了方便阅读和维护。

3.11.1 命名空间的命名

在.NET 框架中，包含很多系统的命名空间，示例代码如下所示。

```
using System;                                    //System 命名空间
using System.Collections.Generic;                // System.Collections.Generic 命名空间
using System.Linq;                               // System.Linq 命名空间
using System.Text;                               // System.Text 命名空间
```

上述代码中，开发人员能够非常方便的了解使用了什么命名空间，这个命名空间大概都能做什么操作，例如 System.Linq 就是为了支持 3.5 中 LINQ 的特性而提供的命名空间。当开发人员创建命名空间时，命名空间的名称应该避免与公司名称或其他品牌的名称相同，例如为了做 Office 开发扩展，可能会命名为 Microsoft.Office，但是此命名空间已经在.NET 框架中被使用了，所以编译器会报错。

技巧：尽量使用开发人员开发组或公司的名称作为命名空间，因为开发组或公司的名称能够表示这个程序或组件是来自哪里。不仅如此，开发组或公司的名称也能在一定程度上避免了重复，例如 HuaWei.TcpIp.Class 既能表示华为的 TCP/IP 研发小组，又可以很大程度上的避免重复。

3.11.2 类的命名原则

同样，类的命名也是有原则的，其原则基本上同命名空间一样，类名尽量使用 Pascal 大写，减少类名称的所写的使用量，并且不推荐使用前缀和下划线。正确的类命名如下所示。

```
class SystemMessage                              //良好的命名
{
}
```

上述代码命名了 SystemMessage 类，用来表示系统发出的消息，SystemMessage 类的名称能够让开发人员清晰的了解该类的基本用途。

3.11.3 接口的命名原则

接口的命名原则与类基本相同，不同的是，在接口的命名中，接口名前应加上大写字母“I”来表示这是一个接口而不是一个类。示例代码如下所示。

```
public interface ISystemMessage                  //良好的接口命名
{
    void SystemOutput(string str);
}
public interface IUserMessage                    //良好的接口命名
{
```

```
        void Output(string str);
    }
```

3.11.4 属性的命名原则

属性的命名中，通常需要在属性名后加上 Attribute，来表示自定义属性类，说明该类用来封装属性，示例代码如下所示。

```
public class PersonInformationAttribute                    //属性命名
{
}
```

3.11.5 枚举的命名原则

枚举的命名同样使用 Pascal 大写。枚举值的名称同样需要使用 Pascal 大写，并同样避免枚举名称中所写的使用量。枚举的命名不需要加入任何的后缀，示例代码如下所示。

```
public enum FileType
{
    Txt,                                                   //定义枚举成员 Txt
    Mp3,                                                   //定义枚举成员 Mp3
    Mp4,                                                   //定义枚举成员 Mp4
    Doc,                                                   //定义枚举成员 Doc
    Pdf,                                                   //定义枚举成员 Pdf
    Html,                                                  //定义枚举成员 Html
    Htm,                                                   //定义枚举成员 Htm
}
```

如果需要使用数字值，这可以使用 Flags 对属性进行自定义，示例代码如下所示。

```
[Flags]                                                    //使用 Flags 属性标记
public enum FileType
{
    Txt,                                                   //定义枚举成员 Txt
    Mp3,                                                   //定义枚举成员 Mp3
    Mp4,                                                   //定义枚举成员 Mp4
    Doc,                                                   //定义枚举成员 Doc
    Pdf,                                                   //定义枚举成员 Pdf
    Html,                                                  //定义枚举成员 Html
    Htm,                                                   //定义枚举成员 Htm
}
```

在 Win32 API 中，这种类型是非常常见的，但是不同的是，Win32 API 中的枚举成员都是大写的，但是在.NET 中还是推荐使用 Pascal 大写。

3.11.6 只读字段的命名原则

只读字段在应用程序的开发中通常作为不可改变的变量而使用，只读字段一旦进行了声明，在代码中就无法对只读字段进行更改。只读字段通常用于声明那些在现实中规定好的变量，例如 π 的值、一千克的精确重量等等。

只读字段的命名同一般的变量的声明方法相同，但是只读字段推荐使用 Pascal 大写命名规则进行只

读字段的变量命名。

3.11.7 参数名

参数名应该具有描述性，即一个参数能够描述当前参数的意义，例如 computerUser 能够描述一台电脑的使用者，而不应该命名为其他没有意义的字符，例如 abc，并推荐使用 camel 大写方式命名。参数的命名应该根据参数的意义来命名，而不是根据参数的种类，并且不提倡使用保留参数、下划线等。示例代码如下所示。

```
public void Set(string computerUser)                    //参数声明
{}
```

3.11.8 委托命名原则

对于委托，使用 EventHandler 作为后缀命名委托处理程序，示例代码如下所示。

```
public delegate void BetaDelHandler(string str);        //委托命名
```

对于委托的参数，也有相应的命名规范，在参数中，使用名为 sender 和 e 两个参数，sender 参数代表提出委托的对象。sender 参数是一个类型的对象，为 Object 类的对象，e 代表与事件相关的状态，示例代码如下所示。

```
public delegate void BetaDelHandler(Object sender,EventArgs e);    //委托的参数
```

3.12 小议设计模式

在应用程序开发中，良好的命名规范是为了规范代码，让代码更加容易维护和阅读。同样，设计模式是一种软件设计方法，也是为了开发出来的应用程序能够更好的使用和维护。在应用程序设计中，维护成本在软件设计占 70%或更高，所以良好的软件结构能够为后期开发和维护提供便利。

3.12.1 什么是设计模式

模式，就是一种规范，在新华字典上关于模式的解释如下。

- 法式，规范，标准：模范。模式。楷模。模型。模本。模压。
- 仿效：模仿（亦作“摹仿”）。模拟（亦作“摹拟”）。模写。

那么，所谓的设计模式，就是软件设计的一种范例。虽然定义看上去非常简单，看是实际上初学者是非常难以理解的，设计模式是一项长期的学习，需要不停的使用和学习才能掌握。

3.12.2 为什么要使用设计模式

设计模式是一种编码的范例。在软件开发过程中，不同的人对类的设计不同，命名的方法也不同，在类的结构上的设计也不同，通常会导致代码混乱，难以阅读和难以维护，以至于上个世纪出现了所谓的“软件危机”。于是人们在软件开发的经验中找到一种模式，所谓的模式，就是一种规律、一种规范，就例如现在的教育模式，大家都会从小学开始，慢慢的读到大学，是一种经典的模式。在软件的开发过程中，也存在这样的模式，就是通常情况下所说的设计模式。

设计模式让人们在软件设计中有据可循。例如在开发一套网站管理系统，首先系统设计师需要对网站进行需求分析，在需求分析之后，就需要规定接口，来说明开发中的类的规范。在规范制定好以后，开发人员需要按照规范来开发软件，并按照统一的类结构或风格编写代码。

设计模式基本上规定了类的结构，规定了类是怎样派生，以及怎样引用。常用的有 28 种设计模式，经常使用的成熟的设计模式在 9 种左右。设计模式的学习是一个长期的过程，初学者在理论上很难理解设计模式，是因为如果没有一定的编码基础，是很难理解为何要使用设计模式的，因为很短的代码段，基本上难以涉及到多人开发，以及市场的考验的经验。

3.12.3 改装现有类

使用设计模式，就需要对现有的代码不停的重构。例如不规范的命名，以及在实际项目过程中遇到的类的结构设计错误等，都需要改装，改装现有类并不是纯属为了类的“好看”，而是为了在实际的运用过程中，能够胜任新的系统要求，并且能够比原有的结构有很好的扩展能力，方便维护企业开发成本。例如，在设计一个制造汽车的过程，设计了一个汽车类，示例代码如下所示。

```
class car                                                    //设计汽车类
{
    string CarNumber;                                        //设计汽车编号
    string CarType;                                          //设计汽车类型
    public string ShowCar()                                  //显式汽车方法
    {
        return "this is a car";
    }
}
class Program                                                //主程序类
{
    static void Main(string[] args)                          //入口方法
    {
        car car = new car();                                 //创建一个新对象
        car.ShowCar();                                       //使用对象的方法
    }
}
```

上述代码中，在同一个文件内声明了一个汽车类 class car，在 Main 方法中，创建了一个 car 的对象，并使用对象 car 的 showcar 方法。从上述代码中可以看出，这其中有几个非常不好的习惯，归纳如下所示。

- 命名没有按照规范，看起来没有层次。
- 命名中的字段没有按照规范。
- 命名中的字段推荐使用属性的方法。
- 类与使用对象的方法放在同一个文件中。

上面的习惯中，最后一条习惯可能读者会有疑问，会在想为何不能放在同一个文件中。其实，在语法上是没有错误的，不好的是，每当汽车类需要更改，开发人员就要重新修改 car 类并重新编译主方法，这样的效率非常的低。就好像当你做了一个网站，更新一个网页就必须更新整个网站一样，这样成本是非常高，而且不利于维护的。最好的做法是将类封装在类库中，存储在“后台”，在“前台”中创建类的对象，而更新细节的时候，只需要更新类库即可。

在设计模式的学习和使用过程中，是没有最好的设计模式的，而另一方面，虽然没有最好的设计模

式但是有最适合的设计模式。设计模式的熟练使用，通常要求开发人员有较熟练的编码以及较好的编码习惯，并且在项目开发上有一定的经验。

3.13 小结

在这一章中，介绍了 C#面向对象的特性。面向对象在构建强大的应用程序中，起着重要的作用。同样，面向对象的设计思想为维护做了铺垫，为了让读者能够加深面向对象的概念，本章还讲解了基本的设计模式的概念。

- 什么是面向对象：介绍了面向对象的概念。
- 面向对象的 C#实现：使用 C#介绍面向对象。
- 对象的生命周期：讲解了构造函数、析构函数、垃圾回收机制等对象的生命周期的概念。
- 使用命名空间：讲解了使用命名空间，以及创建自定义命名空间。
- 类的方法：讲解了什么是类、类成员、字段、方法等基本知识。
- 封装：讲解了封装以及为何要封装。
- 属性：详细讲解了 C#中属性的概念，声明和使用方法。
- 继承：讲解了面向对象中的继承的概念，并用 C#实现。
- 多态：讲解了面向对象中的多态的概念，并用 C#实现。
- 委托和事件：讲解了委托和事件的概念，深入讲解了怎样使用委托。
- 类命名：讲解了命名规范，以及为何要规范命名。
- 小议设计模式：加深读者对面向对象的概念，从实际出发，讲解设计模式的基本概念，以及为何要使用设计模式进行软件开发。

本章讲解了 C#中面向概念的特性，面向概念不仅局限于 C#，在其他面向对象的编程语言中，基本的面向对象的概念都是差不多的，而 JAVA/C#可以说是纯面向对象的编程语言，在.NET 3.5 中，C#还加强了面向对象的特性。