第 5 章　深入了解 C#面向对象编程

本章重点介绍 C#开发语言所涉及的面向对象核心技术，包括继承机制，多态机制，操作符的重载，接口、委托、事件、索引器，异常处理，组件与程序集等内容。通过简单实例，让致力于学习该语言的读者深入学习 C#语言的面向对象编程技术。

- 理解 C#的继承性和多态性
- 掌握操作符重载的方法
- 熟练掌握接口的定义与使用
- 熟练掌握委托的使用
- 初步掌握事件的机制
- 熟练掌握索引器的定义与使用
- 理解异常处理和组件

5.1　C#继承机制


生活常识：

在这个世界上，一般人都有名字、身份证、父母等特征要素。但是，一个成功人士除了有名字、身份证、父母外，还有成功的业绩。所以成功人士也可以看作是一般人。这与 C#语言中继承的原理非常相似：一个基类具备基本特征，派生类除了具备基本特征外还具备特殊的特征。

C#继承机制：

（1）继承是面向对象技术最有特色、最重要、也是与传统编程方法最不相同的。

（2）继承表示了实体间的一种层次关系：

1）基类（父类），派生类（子类）；

2）派生类可以继承基类的特征和能力，如属性和方法；

3）派生类还可以添加新的特性或者是修改已有的特性以满足特定的要求，但不能删除基类的成员；

4）一个父类可以有多个子类，父类是所有子类公共特征的集合，子类则是父类的特殊化；

5）C#中每个子类只能有一个基类，即不允许多重继承。

（3）继承的好处：实现了代码重用。

（4）派生类可以继承基类中除构造函数和析构函数外的所有可访问的成员。

（5）访问修饰符 protected 的作用：子类可以访问，其他的类都不可以访问。

（6）继承是可传递的。

（7）基类与派生类之间的转换：基类与派生类间的转换分为隐式转换和显式转换。

1）隐式转换：派生类—>基类，下面看一个例子，如图 5-1 所示。

2）显式转换：基类—>派生类（有条件），反过来基类向派生对象转换过程就没这么顺利，首先如图 5-2 所示。

```
People p= new People();

Animal a=p;
```

图 5-1　派生类向基类转换举例

```
Animal a= new Animal();

People p=a;
```

图 5-2　基类向派生类转换举例一

这个程序没法编译通过。

接下来，再看改造后的情况，如图 5-3 所示。

做了强制类型转换后，程序可以编译通过，但是在运行过程中，会抛出异常错误。抛出异常：InvalidCastException。

下面进一步改造，如图 5-4 所示。

```
Animal a= new Animal();

People p=(People)a;
```

图 5-3　基类向派生类转换举例二

```
Animal a= new People();

People p=(People)a;
```

图 5-4　基类向派生类转换举例三

这样就可以了。

继承到底有什么好处呢？它是怎样在程序中体现的呢？首先看一个程序代码的举例示意图，如图 5-5 所示。

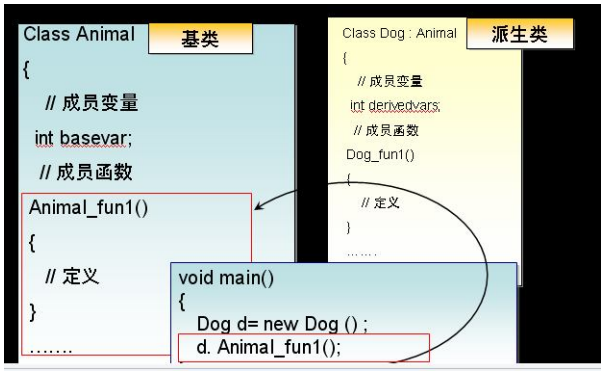

图 5-5　继承关系

派生类 Dog 继承基类 Animal 后，基类的一部分成员就可以被派生类使用，比如基类中的 Animal_fun1 这个方法，在派生类中无须再定义。d 对象是派生类 Dog 的对象，它就可以直接使用 d.Animal_fun1 这个方法。

继承的好处就是无须重新编写代码，维护方便。

我们再看一下继承的层次结构示例，如图 5-6 所示。

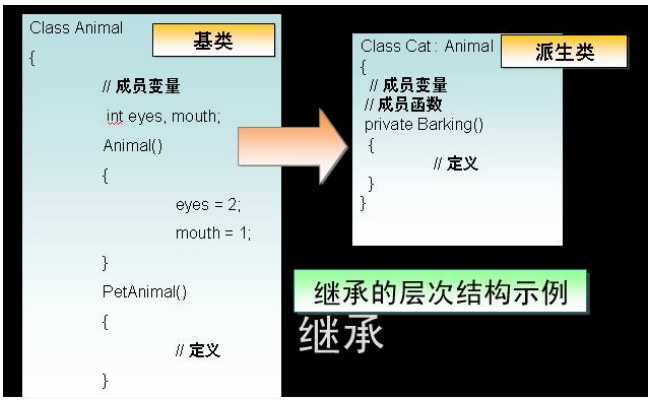

图 5-6　继承的层次结构

继承过程中，不允许实现多重继承，但允许多重接口实现。多重继承指的是一个类既继承了 A，又继承了 B，因为类 A 和类 B 在成员上可能存在着矛盾，所以不允许实现多继承。而接口我们在后面会讲到，它不需要实现它的成员，只要在继承的类里面去实现具体成员，所以允许多重接口实现，如图 5-7 所示。

图 5-7　禁止多重继承

下面再看两个程序例子。

案例学习：类的继承

（1）本次实验要求编写一个程序，程序中定义“动物”这个类，然后再定义一个“狗”类，狗类要继承动物这个类。动物类里要求有获取信息和显示信息的方法，狗类里有获取狗特征的方法。代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 继承
{
    class program
    {
```

```
        static void Main(string[] args)
        {
            Dog dog = new Dog();
            dog.GetInfo();
            dog.DispInfo();
            dog.GetMarks();
            Console.ReadLine();
        }
    }
    public class Animal
    {
        private string _style;
        private int _number;
        public void GetInfo()
        {
            Console.WriteLine("请输入动物的类型和数量:");
            _style = Console.ReadLine();
            _number = int.Parse(Console.ReadLine());
        }
        public void DispInfo()
        {
            Console.WriteLine("该动物的类型为{0}，数量为{1} ", _style, _number);
        }
    }
    public class Dog : Animal
    {
        private int _age;
        private string _weight;
        private string _color ;
        private string _hobby;
        public void GetData()
        {
            Console.WriteLine("请输入狗的年龄:");
            _age =int.Parse( Console.ReadLine());
            Console.WriteLine("请分别输入狗的体重，颜色和爱好:");
            _weight = Console.ReadLine();
            _color = Console.ReadLine();
            _hobby = Console.ReadLine();
            Console.WriteLine("该狗的特征为：{0},{1},{2}",_weight,_color,_ hobby);
        }
    }
}
```

输出结果如图 5-8 所示。

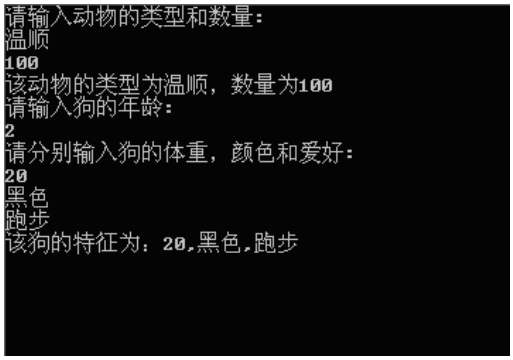

图 5-8　输出结果

程序中，dog 是派生类 Dog 的对象，它调用的方法 GetInfo 和 DispInfo 都是从基类 Animal 中继承过来的。因为在 Animal 类中已经实现了这两个方法，所以，无须在类 Dog 中实现这两个方法。而 GetData 方法在类 Animal 中没有，是 Dog 类中自己的成员，所以需要在 Dog 类中实现。

（2）本次实验要求编写一个程序，程序中定义“动物”这个类，然后再定义一个“狗”类，狗类要继承动物这个类。动物类里要求有获取信息和显示信息的方法，狗类里有获取狗特征的方法。在此基础上再定义一个“宠物狗”类，要求有一个区别，不同于其他一般看家狗的方法。代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 继承1
{
    class program
    {
        static void Main(string[] args)
        {
            PetDog dog = new PetDog();
            dog.GetInfo();
            dog.DispInfo();
            dog.GetData();
            dog.GetDiff();
            Console.ReadLine();
        }
    }
    public class Animal
    {
        private string _style;
```

```
        private int _number;
        public void GetInfo()
        {
            Console.WriteLine("请输入动物的类型和数量:");
            _style = Console.ReadLine();
            _number = int.Parse(Console.ReadLine());
        }
        public void DispInfo()
        {
            Console.WriteLine("该动物的类型为{0}，数量为{1} ", _style, _number);
        }
    }
    public class Dog : Animal
    {
        private int _age;
        private string _weight;
        private string _color ;
        private string _hobby;
        public void GetData()
        {
            Console.WriteLine("请输入狗的年龄:");
            _age =int.Parse( Console.ReadLine());
            Console.WriteLine("请分别输入狗的体重，颜色和爱好:");
            _weight = Console.ReadLine();
            _color = Console.ReadLine();
            _hobby = Console.ReadLine();
            Console.WriteLine("该狗的特征为: {0},{1},{2}",_weight,_color,_ hobby);
        }
    }
    public class PetDog : Dog
    {
        public void GetDiff()
        {
            Console.WriteLine("宠物狗比一般的看家狗更干净，更时尚！");
        }
    }
}
```

输出结果如图 5-9 所示。

在继承过程中，经常会遇到一个关键字 base。它的作用：

- 用于从派生类中访问基类成员。
- 可以使用 base 关键字调用基类的构造函数。

下面看一个例子，调用 base 构造函数，如图 5-10 所示。

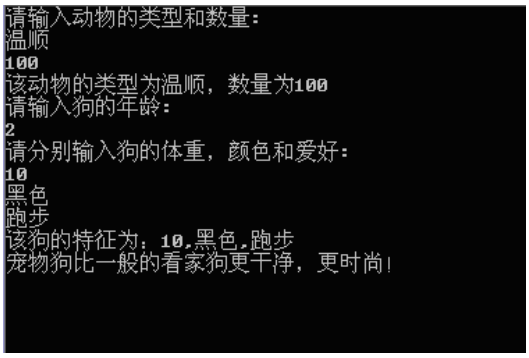

图 5-9　输出结果

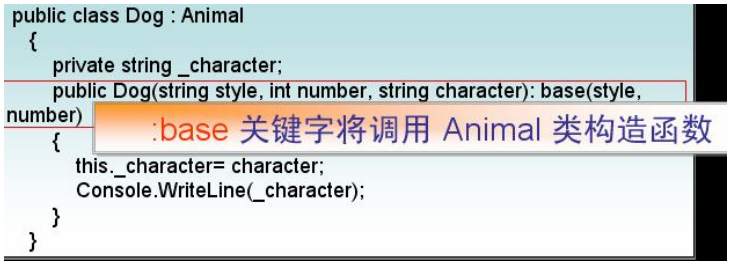

图 5-10　调用 base 构造函数示意

案例学习：base 的应用

本次实验要求编写一个程序，程序中定义“动物”这个类，然后再定义一个“狗”类，狗类要继承动物这个类。在狗对象实例化的同时，调用 base 构造函数为狗对象赋予类型、数量等信息。代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 继承应用
{
    class program
    {
        public static void Main(string[] args)
        {
            //构造 Student
            Dog d = new Dog("温顺型", 100, "跑步");
            Console.ReadLine();
        }
    }
    public class Animal
    {
        public string _style;
        public int _number;
```

```
        public Animal(string style, int number)
        {
            this._style = style;
            this._number = number;
            Console.WriteLine(_style);
            Console.WriteLine(_number);
        }
    }
    public class Dog : Animal
    {
        private string _character;
        public Dog(string style, int number, string character): base(style, number)
        {
            this._character= character;
            Console.WriteLine(_character);
        }
    }
}
```

输出结果如图 5-11 所示。

图 5-11　输出结构

这个程序在对 Dog 的对象实例化时，将参数 style 和 number 的值传递给通过 base 调用的 Animal 构造函数，这样完成了对类型、数量等信息的赋值工作。

5.2　C#多态机制

生活常识

课堂上，老师给出很多个函数，他们的作用都是比较大小。这些函数的名称都相同，不同的是参数的类型，个数和函数的返回值。过后，老师又给出一大堆数据，它们有整型、单精度、双精度等。而同学们要根据这些数据找出相应的函数，最终根据那个函数判断大小。这就是多态的原理。

（1）C#多态机制：

1）多态（Polymorphism）：多态的意思是事物具有不同形式的能力。例如，对不同的实例，某个操作可能会有不同的行为。这个行为依赖于所要操作数据的类型。

2）多态：用同样的一个语句，执行不同的操作。

3）多态机制使具有不同内部结构的对象可以共享相同的外部接口。

（2）如何实现多态？

1）C#中有两种实现多态的方法：

① 通过继承实现多态；

② 通过重载实现多态。

2）通过继承，可以用两类方法来实现多态：

① 重写基类的虚方法（虚方法重写）；

② 重写基类的抽象方法。

（3）对基类虚方法的重写涉及的问题：

1）基类和派生类中定义完全相同的两个方法：

① 方法名相同；

② 对应的参数相同；

③ 返回值相同。

2）语法规定：

① 基类的方法必须用 virtual 修饰符定义为虚方法；

② 派生类必须用 override 修饰符重新定义该方法。

3）与非虚方法的比较。

4）虚方法调用的特点：由对象变量所引用的对象来决定执行哪一个方法，而与对象变量本身的类型无关。

5）方法重写是实现多态的一种方法。

5.2.1 方法重写

案例学习：基类虚方法的重写的应用

下面的示例演示了基类虚方法的重写。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 虚方法
{
    class program
    {
        public static void Main(string[] args)
        {
            circle pan = new circle();
            pan.L();
            Console.WriteLine("圆的面积为{0}", pan.S());
            earth e = new earth();
            e.L();
            Console.ReadLine();
```

```
        }
    }
    public class circle
    {
        double pi = 3.14;
        double r = 0.0;
        virtual public void L()
        {
            Console.Write("请输入圆半径：");
            r = double.Parse(Console.ReadLine());
            Console.WriteLine("圆的周长为{0}", 2 * pi * r);
        }
        public double S()
        {
            Console.Write("请输入圆半径：");
            r = double.Parse(Console.ReadLine());
            return pi * r * r;
        }
    }
    public class earth : circle
    {
        string brand;
        public override void L()
        {
            Console.Write("地球仪的品牌是：");
            brand = Console.ReadLine();
            base.L();
            Console.WriteLine("{0}牌地球仪还不错", brand);
        }
    }
```

程序中，首先定义一个基类 circle 表示圆，类里有个成员是方法 L 表示圆的周长。又定义了一个类 earth 表示地球仪，也有个方法 L 表示地球仪的周长。由于在基类和派生类中 L 方法实现功能不一样，派生类 earth 比基类 circle 中的方法 L 多一个品牌功能。所以，我们在定义的时候将 circle 类中的 L 方法用 virtual 修饰，而在 earth 类中的 L 方法用 override 修饰，表示 cirlcle 类中的 L 方法是虚方法，earth 类中的方法 L 是对虚方法的重写。

按 F5 键调试、运行程序，输出结果如图 5-12 所示。

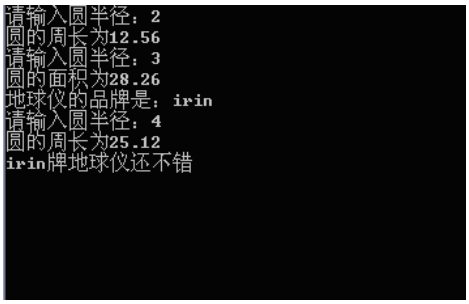

图 5-12　输出结果

案例学习：基类非虚方法的重写的应用

下面的示例演示了基类非虚方法的重写（方法的隐藏）。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 非虚方法
{
    class program
    {
        public static void Main(string[] args)
        {
            People p = new People();
            Student s = new Student();
            People pc = s;
            p.HideFun();
            s.HideFun();
            pc.HideFun();
            Console.ReadLine();
        }
    }
    class People
    {
        public void HideFun()
        {
            Console.WriteLine("人类的 HideFun 方法");
        }
    }
    class Student : People
    {
        public void HideFun()
        {
            Console.WriteLine("学生类的 HideFun 方法");
        }
    }
}
```

该程序中，基类和派生类都定义了方法 HideFun，如果用基类对象调用该方法则输出父类 People 的 HideFun 方法，如果是派生类对象调用该方法则输出子类 Student 的 HideFun 方法。也就是说，在继承过程中，派生类的方法将同名的基类方法隐藏了。

按 F5 键调试、运行程序，输出结果如图 5-13 所示。

图 5-13　输出结果

大家是否注意一个问题，在这个程序编译过程中，有个警告，“警告‘Samsung.Student.HideFun()’隐藏了继承的成员‘Samsung.People.HideFun()’。如果是有意隐藏，请使用关键字 new。”但是程序可以正常执行，也就是编译系统希望你用 new 这个关键字进行隐藏操作。

5.2.2　方法的隐藏

（1）派生类可以定义与基类具有相同签名的方法。

（2）new 关键字：派生类定义与基类具有相同签名的方法时，需要使用 new 关键字，否则编译器将给出警告。

（3）当用派生类的对象访问同名的方法时：

1）执行派生类的方法？对象变量.HideFun ()；

2）执行基类的方法？由对象变量的类型决定。对象变量的类型如表 5-1 所示。

表 5-1　隐藏方法执行表

调用方法	说明
p.HideFun()	p 是 People 类对象，执行 People 类的 HideFun 方法
s.HideFun()	s 是 Student 类对象，执行 Student 类的 HideFun 方法
pc.HideFun()	pc 是 People 类对象，执行 People 类的 HideFun 方法

问题讨论：

实际上 pc 引用的是 Student，因此本质是 Student。能否执行 Student 的 HideFun？如果不能，那我又怎样才能执行 Student 的 HideFun?

基类与派生类的方法关系如表 5-2 所示。

表 5-2　基类与派生类的方法关系表

类型	说明
扩充	是派生类新增的，基类没有
重载	派生类中有与基类同名的方法，但参数类型或个数不同
完全相同	派生类中定义了一个与基类相同的方法，即方法的原型完全相同
隐藏	可以声明与继承而来的同名的成员
重写	基类的方法，属性，索引器重新定义，而成员名和相应的参数都不变
调用	用 base 调用的基类方法

5.2.3　抽象类和抽象方法

抽象类和抽象方法，访问修饰符用 abstract。

语法：

```
abstract class ClassOne
```

```
{
    //类实现
}
```

含有抽象方法的类是抽象类，抽象类可以没有抽象方法。抽象类是派生类的基类，不能实例化。抽象方法在抽象类里面不能实现。在派生类中，抽象方法等抽象成员必须被重写并实现，如图 5-14 所示。

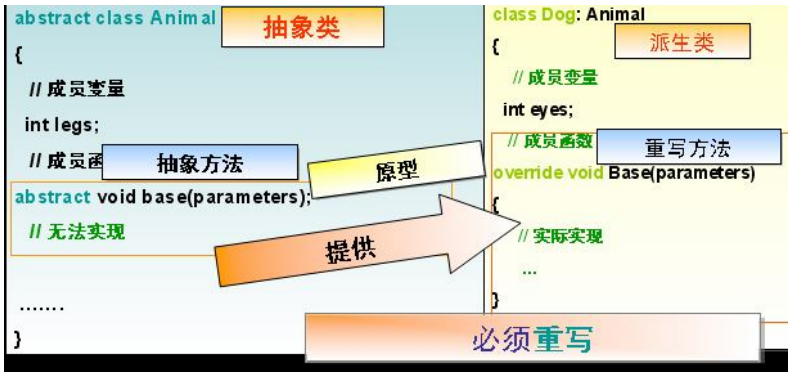

图 5-14　抽象类中的抽象方法在派生类中被重写并实现

多态性还分为运行时多态性和编译时多态性：

（1）运行时的多态性是通过继承和虚成员来实现的。运行时的多态性是指系统在编译时不确定选用哪个重载方法，而是直到程序运行时，才根据实际情况决定采用哪个重载方法。

（2）编译时的多态性具有运行速度快的特点，而运行时的多态性则具有极大的灵活性。

5.3　操作符重载

这个方法允许用户定义的类型如结构和类，为使它们的对象易于操作而使用重载操作符。

如何实现操作符重载？

（1）运算符重载实质上就是函数重载。

（2）运算符的函数表示法，如表 5-3 所示。

表 5-3　函数表示法表

运算符	函数表示法
op x	operator op(x)
x op	operator op(x)
x op y	operator op(x,y)

（3）语法规定：

1）允许重载的运算符：如表 5-4 所示。

2）必须是 public 和 static。

3）至少有一个参数是类自身。

可以被重载的操作符，如表 5-4 所示。

表 5-4　被重载的操作符表

操作符	描述
+, –, ! , ~, ++ , ––	这些一元操作符需要一个操作数，可以被重载
+,　-, * , /, %	这些二元操作符需要两个操作数，可以被重载
==, !=, <, >, <=, >=	比较操作符可以被重载
&&, ||	条件逻辑操作符不能被直接重载，但是它们使用& 和|它们可以求值，被重载
+=, –=, *=, /=, %=	赋值操作符不能被重载
=, ., ?:, ->, new, is, sizeof, typeof	这些操作符不能被重载

案例学习：运算符重载的应用

下面的示例演示了运算符重载的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 运算符重载
{
    class TwoD
    {
        private int x;
        public int X
        {
            get
            {
                return x;
            }
        }
        private int y;
        public int Y
        {
            get
            {
                return y;
            }
        }
        public TwoD() { x = y = 0; }
        public TwoD(int a, int b)
        {
            x = a;
            y = b;
        }
        public static TwoD operator +(TwoD op1, TwoD op2)
        {
            TwoD jieguo = new TwoD();
```

```
            jieguo.x = op1.x + op2.x;
            jieguo.y = op1.y + op2.y;
            return jieguo;
        }
        public static TwoD operator +(TwoD op1, int op2)
        {
            TwoD jieguo = new TwoD();
            jieguo.x = op1.x + op2;
            jieguo.y = op1.y + op2;
            return jieguo;
        }
        public override string ToString()
        {
            return string.Format("x坐标:{0},y坐标:{1}", X, Y);
        }
    }
}
```

在这个程序中，+号被重载，被重载的+号可以用来计算 TwoD 类型对象的相加，还可以计算 TwoD 类型对象与整数类型的相加操作。

下面再看一个程序例子。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 运算符重载
{
    class ThreeD : TwoD
    {
        private int z;
        public ThreeD() : base() { z = 0; }
        public ThreeD(int a, int b, int c)
            : base(a, b)
        {
            z = c;
        }
        public int Z
        {
            get
            {
                return z;
            }
        }
        public static ThreeD operator +(ThreeD op1, ThreeD op2)
        {
            ThreeD jieguo = new ThreeD(op1.X + op2.X, op1.Y + op2.Y, op1.z + op2.z);
            return jieguo;
        }
        public static ThreeD operator ++(ThreeD op1)
        {
            ThreeD jg = new ThreeD(op1.X + 1, op1.Y + 1, op1.z + 1);
            return jg;
        }
        public static bool operator ==(ThreeD op1, ThreeD op2)
```

```
        {
            if ((op1.X == op2.X) && (op1.Y == op2.Y) && (op1.z == op2.z))
                return true;
            else
                return false;
        }
        public static bool operator !=(ThreeD op1, ThreeD op2)
        {
            if ((op1.X != op2.X) || (op1.Y != op2.Y) || (op1.z != op2.z))
                return true;
            else
                return false;
        }
        public override string ToString()
        {
            return string.Format("{0},z 坐标:{1}", base.ToString(), Z);
        }
    }
}
```

在这个程序中，+、++、==、!=符号被重载。

（1）操作符重载为 C#操作符应用到用户定义的数据类型提供了额外的能力。

（2）仅预定义的 C#操作符可以被重载。

5.4　接口

对于接口的理解，首先我们看一个生活小例子，这是一个开关，它需要两个方法 ON 和 OFF，如图 5-15 所示。

图 5-15　生活中关于开关

另外还有其他开关，如图 5-16 所示。

图 5-16　生活中其他开关

其他开关也涉及 ON、OFF 两个方法，但是它们的实现与前面提到的开关不一样。所以我们在定义开关这种数据类型时，不能直接实现，必须根据具体开关来决定怎样实现 ON、OFF 方法。在这种情况下，我们可以把开关定义为接口，比如命名为 ISwitch。在接口里定义两个方法 ON 和 OFF，这两个方法只定义不实现。

案例学习：接口的应用

下面的示例演示了接口的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 接口应用
{
    class Program
    {
        static void Main(string[] args)
        {
            Button bt = new Button();
            bt.On();
            bt.Off();
            OnOffSwitch oos = new OnOffSwitch();
            oos.On();
            oos.Off();
            Console.ReadLine();
        }
    }
    public interface ISwitch
    {
        void On();
        void Off();
```

```
    }
    public class Button : ISwitch
    {
        public void On()
        {
            Console.WriteLine("顺时针转动！");
        }
        public void Off()
        {
            Console.WriteLine("逆时针转动！");
        }
    }
    public class OnOffSwitch : ISwitch
    {
        public void On()
        {
            Console.WriteLine("开启！");
        }
        public void Off()
        {
            Console.WriteLine("关闭！");
        }
    }
}
```

输出结果如图 5-17 所示。

图 5-17　输出结构

（1）实现接口基本步骤：

1）定义接口：接口定义了规则。

2）实现接口：类实现了接口的规则。

（2）那么接口是如何定义的呢？

- 接口是引用类型：

1）关键字 interface。

2）接口的成员有：属性、方法、事件和索引器。

3）接口中定义的成员只有声明，没有实现。

4）接口中的成员都隐式地具有 public 访问属性接口的定义，如图 5-18 所示。

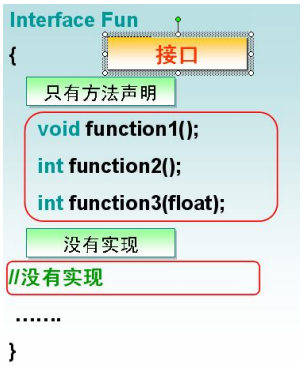

图 5-18　接口的定义

案例学习：定义接口的应用

下面的示例演示了定义接口的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 定义接口
{
    class Program
    {
        static void Main(string[] args)
        {
            MyPoint p = new MyPoint(1,2,3);
            Console.WriteLine("X={0},Y={1},Z={2}", p.X, p.Y,p.Z);
            Console.ReadLine();
        }
    }
    interface IPoint
    {
        int X
        {
            get;
            set;
        }
        int Y
        {
            get;
            set;
        }
        int Z
        {
            get;
            set;
        }
```

```
    }
    class MyPoint : IPoint
    {
        private int myX;
        private int myY;
        private int myz;
        public MyPoint(int x, int y,int z)
        {
            myX = x;
            myY = y;
            myz = z;
        }
        //实现 IPoint 接口中的属性
        public int X
        {
            get
            {
                return myX;
            }
            set
            {
                myX = value;
            }
        }
        public int Y
        {
            get
            {
                return myY;
            }
            set
            {
                myY = value;
            }
        }
        public int Z
        {
            get
            {
                return myz;
            }
            set
            {
                myz = value;
            }
        }
    }
}
```

输出结果如图 5-19 所示。

图 5-19　输出结果

这个程序定义了一个表示点的接口 IPoint。它有三个成员是 X、Y 和 Z，这三个成员是属性，但在接口里只完成定义，不能实现。它们的实现是在继承接口的类 MyPoint 中完成的。

用类实现接口：

（1）语法：与继承一样。

（2）规定：必须实现接口中声明的所有成员。

案例学习：用类实现接口的应用

下面的示例演示了定义用类实现接口的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 实现接口
{
    class Program
    {
        static void Main(string[] args)
        {
            Dog d = new Dog();
            d.AnimalFun();
            Student s = new Student();
            s.AnimalFun();
            s.PeopleFun();
            Console.ReadLine();
        }
    }
    interface IAnimal
    {
        void AnimalFun();
    }
    interface People : Animal
    {
        void PeopleFun();
    }
    class Dog : Animal
    {
        public void AnimalFun()
```

```
        {
            Console.WriteLine("Dog 类实现 Animal 接口的 AnimalFun()");
        }
    }
    class Student : People
    {
        public void AnimalFun()
        {
            Console.WriteLine("Student 类实现 Animal 接口的 AnimalFun()");
        }
        public void PeopleFun()
        {
            Console.WriteLine("Student 类实现 People 接口的 PeopleFun()");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-20 所示。

图 5-20　输出结果

接口与继承：

（1）接口的继承：

1）接口可以继承。

2）接口可以从多个基接口继承，类不允许多重继承。

（2）类的继承、实现接口的类：

1）相同的语法：类的继承，类实现接口。

2）类不能多重继承。

3）类可以实现多个接口，C#可以通过接口来实现多重继承。

4）规则：类的基列表中可以同时包含基类和接口，但基类应列在首位。

案例学习：接口与继承的应用

下面的示例演示了接口与继承的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 接口与继承
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            //实现接口 Animal 的类 Dog
            Animal a1 = new Dog();
            Animal a2 = (Animal)(new Dog());
            a1.AnimalFun();
            a2.AnimalFun();
            //实现接口 People 的类 Student，People 接口从 Animal 接口继承
            Animal a3 = new Student();
            Animal a4 = (Animal)(new Student());
            a3.AnimalFun();
            a4.AnimalFun();
            //编译错误
            //Student s1 = new Animal();
            //运行错误，抛出 InvalidCastException 异常
            //Student s2=(Student)(new Animal());
            Student s3 = new Student();
            People s4 = (People)(new Student());
            s3.PeopleFun();
            s3.AnimalFun();
            s4.PeopleFun();
            s4.AnimalFun();
            Console.ReadLine();
        }
    }
    interface Animal
    {
        void AnimalFun();
    }
    interface People : Animal
    {
        void PeopleFun();
    }
    class Dog : Animal
    {
        public void AnimalFun()
        {
            Console.WriteLine("Dog 类实现 Animal 接口的 AnimalFun()");
        }
    }
    class Student : People
    {
        public void AnimalFun()
        {
            Console.WriteLine("Student 类实现 Animal 接口的 AnimalFun()");
        }
        public void PeopleFun()
        {
            Console.WriteLine("Student 类实现 People 接口的 PeopleFun()");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-21 所示。

图 5-21　输出结果

接口的实例：

（1）什么是接口的实例：

1）接口的实例是指接口类型的变量（但不能用 new 实例化）。

2）声明语法：接口类型.接口实例名

（2）接口实例的赋值：

1）接口实例＝对象名——编译正确，可能发生运行错误。

2）接口实例＝(接口类型)对象名——编译正确，可能发生运行错误。

3）接口实例 B＝(接口类型 B)接口实例 A——编译正确，可能发生运行错误。

（3）接口实例的作用：通过接口实例访问接口的成员。

案例学习：接口实例的应用

下面的示例演示了接口实例的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 接口实例
{
    class Program
    {
        static void Main(string[] args)
        {
            //实现接口 Animal 的类 Dog
            Animal a = new Dog();
            //运行错误，抛出 InvalidCastException 异常
            // People p=(People)(a);
            //实现接口 People 的类 Student，People 接口从 Animal 接口继承
            Animal a1 = new Student();
            a1.AnimalFun();
            People p1 = (People)(a1);
            p1.PeopleFun();
            People p2 = new Student();
```

```
            p2.PeopleFun();
            Animal a2 = (Animal)p2;
            a2.AnimalFun();
            Console.ReadLine();
        }
    }
    interface Animal
    {
        void AnimalFun();
    }
    interface People : Animal
    {
        void PeopleFun();
    }
    class Dog : Animal
    {
        public void AnimalFun()
        {
            Console.WriteLine("Dog 类实现 Animal 接口的 AnimalFun()");
        }
    }
    class Student : People
    {
        public void AnimalFun()
        {
            Console.WriteLine("Student 类实现 Animal 接口的 AnimalFun()");
        }
        public void PeopleFun()
        {
            Console.WriteLine("Student 类实现 People 接口的 PeopleFun()");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-22 所示。

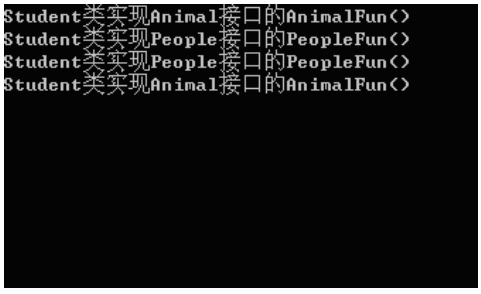

图 5-22　输出结果

接口的应用：

（1）用接口实现多重继承；

（2）用接口实现多态。

案例学习：用接口实现多重继承

对于多重继承的理解，我们可以看一个生活的例子：鸭子是一种动物，也是一种鸟，会游泳，同时又是一种食物。编写代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 接口实现
{
    class Program
    {
        static void Main(string[] args)
        {
            Duck duck = new Duck();
            duck.fly();
            duck.animal();
            duck.cook();
            duck.swim();
            Console.ReadLine();
        }
    }
    public interface Animal
    {
        void animal();
    }
    public interface Swim
    {
        void swim();
    }
    public interface Food
    {
        void cook();
    }

    public abstract class Bird
    {
        public abstract void fly();
    }
    public class Duck : Bird,Animal, Food, Swim
    {
        public override void fly()
        {
            Console.WriteLine("鸭子都会飞哦！");
        }
        public void animal()
        {
            Console.WriteLine("鸭子是禽类动物哦！");
```

```
        }
        public void cook()
        {
            Console.WriteLine("北京烤鸭是北京的特产哦！");
        }
        public void swim()
        {
            Console.WriteLine("鸭子能够在水里游泳！");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-23 所示。

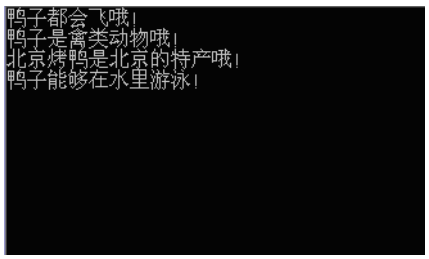

图 5-23　输出结果

案例学习：用接口实现多态

看一下接口多态是如何实现的？一个生活的例子：猫、狗、猴子可以跑。编写代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 接口实现1
{
    class Program
    {
        static void Main(string[] args)
        {
            Run[] run = new Run[3];
            run[0] = new Cat();
            run[1] = new Dog();
            run[2] = new Monkey();
            foreach (Run r in run)
            {
                r.run();
            }
            Console.ReadLine();
        }
    }
    public interface Run
```

```
    {
        void run();
    }
    public class Cat : Run
    {
        public void run()
        {
            Console.WriteLine("猫在屋檐上跑！");
        }
    }
    public class Dog : Run
    {
        public void run()
        {
            Console.WriteLine("小狗在花园里跑！");
        }
    }
    public class Monkey: Run
    {
        public void run()
        {
            Console.WriteLine("猴子在山上跑！");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-24 所示。

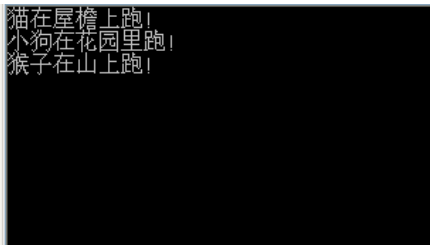

图 5-24　输出结果

案例学习：抽象类的应用

下面的示例演示了抽象类的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 抽象类
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Cat c = new Cat();
            c.animal();
            Duck d = new Duck();
            d.animal();
            d.fly();
            Console.ReadLine();
        }
    }
    abstract class Animal
    {
        public abstract void animal();
    }
    interface Bird
    {
        void fly();
    }
    class Cat : Animal
    {
        public override void animal()
        {
            Console.WriteLine("猫是一种可爱的动物！");
        }
    }
    class Duck: Animal, Bird
    {
        public override void animal()
        {
            Console.WriteLine("鸭子是禽类动物！");
        }
        public void fly()
        {
            Console.WriteLine("鸭子会飞哦！");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-25 所示。

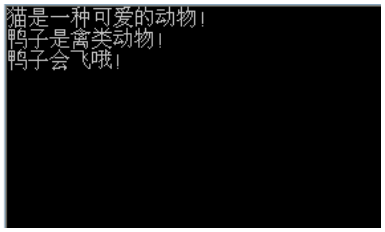

图 5-25　输出结果

抽象类与接口的比较如表 5-5 所示。

表 5-5　抽象类与接口的比较表

抽象类	接口
无	有属性、方法、事件和索引器
可以有只声明的方法，也可以有完整的方法	方法都只有声明
是对实体的抽象	是对行为的抽象，规定行为的准则
子类与抽象类，在概念上是一致的	实现接口的类与接口，在概念上是不同的

多重接口实现：

- C#不允许多重类继承。
- 但 C#允许多重接口实现。
- 这意味着一个类可以实现多个接口。

在 C#中，只要不发生命名冲突，就完全可以允许多重接口实现。那什么时候用显式接口实现呢？程序截图，如图 5-26 所示。

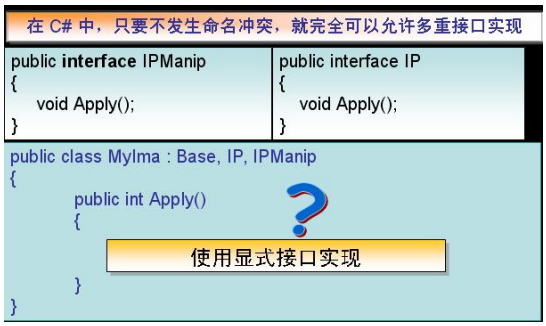

图 5-26　程序

在这个程序中有两个接口 IPcitManip 和 IPict，都有一个方法 ApplyAlpha，有个类 MyImages 继承了这两个接口，那么在实现的时候，到底是实现哪个接口的 ApplyAlpha 方法呢？这时就需要使用显示接口实现，如图 5-27 所示。

```
public class MyIma : Base, IP, IPManip
  {
    public int Delete()
    {
      Console.WriteLine("Delete！ ");
      return (5);
    }
    public void Apply()
    {
      Console.WriteLine("Apply！ ");
    }
    void IP.Display()
    {
      Console.WriteLine("Display的 IP 实现");
    }
    void IPManip.Display()
    {
      Console.WriteLine("Display 的 IPManip 实现");
    }
  }
```

显式接口实现

图 5-27　显式接口实现程序

案例学习：显式接口的应用

下面的示例演示了显式接口的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 显示接口
{
    class Program
    {
        static void Main(string[] args)
        {
            MyIma m = new MyIma();
            IP P = m; //IP 引用
            P.Display();
            IPManip Pi = m;//IPManip 引用
            Pi.Display();
            Console.ReadLine();
        }
    }
    public interface IP
    {
        int Delete();
        void Display();
    }
    public interface IPManip
    {
        void Display();
    }
    public interface Base
    {
        void Apply();
    }
    public class MyIma : Base, IP, IPManip
    {
        public int Delete()
        {
            Console.WriteLine("Delete! ");
            return (5);
        }
        public void Apply()
        {
            Console.WriteLine("Apply! ");
        }
        void IP.Display()
        {
            Console.WriteLine("Display 的 IP 实现");
```

```
        }
        void IPManip.Display()
        {
            Console.WriteLine("Display 的 IPManip 实现");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-28 所示。

图 5-28　输出结果

案例学习：非显式接口的应用

下面的示例演示了非显式接口的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 非显示接口
{
    class Program
    {
        static void Main(string[] args)
        {
            MyIma m = new MyIma();
            m.Display();
            int v = m.Delete();
            Console.WriteLine(v);
            m.ApplyA();
            m.ApplyB();
            Console.ReadLine();
        }
    }
    public interface IP
    {
        int Delete();
    }
    public interface IPManip
    {
        void Display();
```

```
        void ApplyA();
    }
    //继承多重接口
    public interface IPAll : IP, IPManip
    {
        void ApplyB();
    }
    public class MyIma : IPAll
    {
        public int Delete()
        {
            Console.WriteLine("Delete! ");
            return (5);
        }
        public void ApplyA()
        {
            Console.WriteLine("ApplyA! ");
        }
        public void ApplyB()
        {
            Console.WriteLine("ApplyB!");
        }
        public void Display()
        {
            Console.WriteLine("Display!");
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-29 所示。

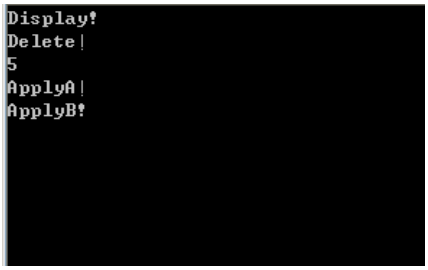

图 5-29　输出结果

5.5　委托

生活常识

古时候大多是人都是通过相亲，成为夫妻的。而在他们相亲的时候，都是通过媒婆介绍

认识的。这时候男女双方要求什么条件都是通过媒婆来传达的。而这个媒婆就类似于委托。

委托的概念：

（1）委托（Delegate）也是一种数据类型，它指的是某种类型的方法。

（2）可以定义委托变量（委托对象），但该变量接收的是一个函数的地址。

还可以理解为：委托是一个可以对方法进行引用的类。使用委托使程序员可以将方法引用封装在委托对象内。然后可以将该委托对象作为参数传递给引用该方法的方法，而不必在编译时知道将调用哪个方法。

（3）委托是从 System.Delegate 派生的类。

（4）使用委托的步骤：

1）定义一个新的委托。

2）声明委托变量并实例化。

3）使用委托变量：

① 通过委托变量调用方法；

② 委托变量可以作为参数传递。

定义一个新的委托，语法：

[访问修饰符]　delegate　返回类型　委托名(参数列表);

再看一个例子，如图 5-30 所示。

```
delegate void MyDelegate(float x);
```

图 5-30　委托举例

委托定义的位置可以放在类内，也可以放在类外。

委托的实例化，有两种方法，一个是用 new 实例化，一个是用赋值的办法，如图 5-31 和图 5-32 所示。

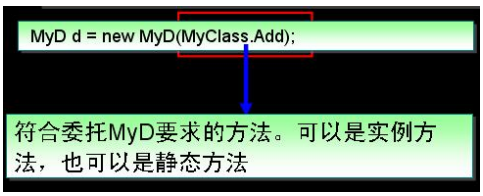

图 5-31　委托实例方法一

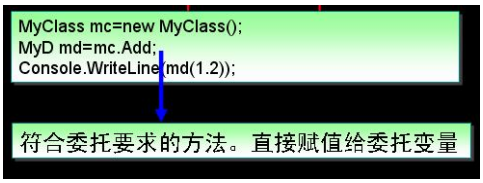

图 5-32　委托实例方法二

符合委托要求：返回值、方法签名要一致。

案例学习：通过委托变量调用方法的应用一

下面的示例演示了通过委托变量调用方法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 委托
{
    class Program
    {
        static void Main(string[] args)
        {
            MyD d = new MyD(MyClass.Add);
            d(2);
            Console.ReadLine();
        }
    }
    delegate void MyD(float x);
    class MyClass
    {
        public static void Add(float x)
        {
            float result = 2*x;
            Console.WriteLine("{0}的二倍等于:{1}", x, result);
            Console.ReadLine();
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-33 所示。

图 5-33　输出结果

案例学习：委托变量可以作为参数传递的应用

下面的示例演示了委托变量可以作为参数传递。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace 委托1
{
    class Program
    {
        static void Main(string[] args)
        {
            MyD d = new MyD(MyClass.Add);
            ExecuteMethod(d,5);
            Console.ReadLine();
        }
        static void ExecuteMethod(MyD d, float x)
        {
            d(x);
        }
    }
    delegate void MyD(float x);
    class MyClass
    {
        public static void Add(float x)
        {
            float result = 2 * x;
            Console.WriteLine("{0}的二倍等于:{1}", x, result);
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-34 所示。

图 5-34　输出结果

案例学习：通过委托变量调用方法的应用二

下面的示例演示了通过委托变量调用方法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 委托2
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            MyD d = new MyD(MyClass.Add);
            d += new MyD(MyClass.Double);
            d(5);
            Console.ReadLine();
        }
    }
    delegate void MyD(float x);
    class MyClass
    {
        public static void Add(float x)
        {
            Console.WriteLine("两个{0}的和等于:{1}", x, x + x);
        }
        public static void Double(float x)
        {
            Console.WriteLine("{0}的两倍等于:{1}", x, 2 * x);
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-35 所示。

图 5-35　输出结果

另外，还有一个组合委托的概念。

组合委托（即多点委托）：

（1）多点委托：

1）一个委托变量中可以包含多个方法，即可以一次执行多个方法；

2）多点委托是从 System.MulticastDelegate 派生的类。

（2）委托的组合与分解：

1）组合：+，+=运算

2）分解：-，-=运算

（3）多点委托的限制：多点委托只允许使用返回值为 void 的方法。

5.6　事件

为了理解事件，可以用一个学生听课的生活实例描述一个事件，如图 5-36 所示。

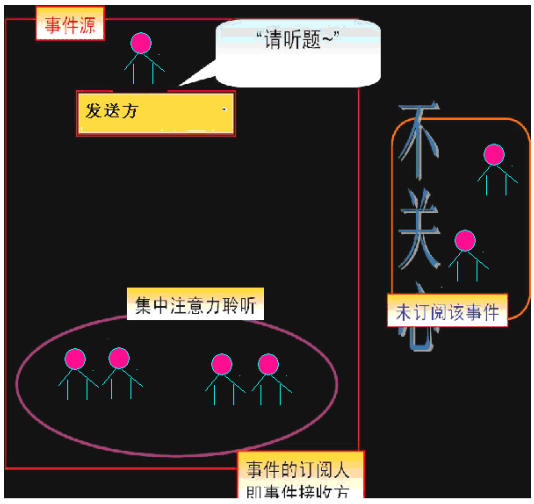

图 5-36　事件举例

我们把老师讲课看作是一个事件。讲课的老师是事件的发送方。学生分成两类：集中注意力聆听的学生和不关心课程的学生。集中注意力聆听的学生是事件的订阅人，即事件的接收方。而不关心课程的学生未订阅该事件。当老师发出“请听题～”的指令时，事件产生并发送，事件订阅人接收事件，未订阅事件者不接收事件。事件接收者接收事件后做相应的处理——开始听老师讲题。

那么，什么是事件？

（1）事件的组成：

1）事件是一种发布消息的机制；

2）事件的两个方面：发送方与接收方；

3）发送方负责发布消息；

4）接收方进行响应，即接收消息后进行必要的处理；

5）发送方与接收方通过事件订阅建立关联。

（2）C#中的事件：

1）C#中事件是类的成员；

2）C#中的事件是通过委托来实现的。

发送方与接收方通过事件订阅建立关联，事件是类用来通知对象需要执行某种操作的方式。

案例学习：事件应用

下面的示例演示了事件应用。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 事件
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Publisher pub = new Publisher();
            Receiver rece1 = new Receiver("阿花");
            Receiver rece2 = new Receiver("阿福");
            pub.Event += new MyD(rece1.Method);
            pub.Event += new MyD(rece2.Method);
            pub.FireEvent("面试时间到了！");
            Console.ReadLine();
        }
    }
    public delegate void MyD(string mess);
    //事件发送者
    public class Publisher
    {
        public event MyD Event;
        public void FireEvent(string notice)
        {
            if (Event != null) Event(notice);
        }
    }
    //事件接收者
    public class Receiver
    {
        string _name;
        public Receiver(string name)
        {
            _name = name;
        }
        public void Method(string s)
        {
            Console.WriteLine("{0}知道:{1}", _name, s);
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-37 所示。

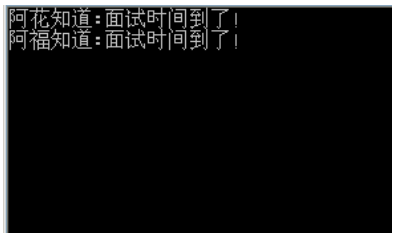

图 5-37　输出结果

如何实现事件，如表 5-6 所示。

表 5-6　如何实现事件步骤表

步骤	说明	位置
定义委托	见委托的定义	
定义事件	用 event 关键字定义事件，事件是一个委托变量	发送方
触发事件	通知所有订阅该事件的对象	发送方
订阅事件	接收对象要向发送对象订阅事件，接收方与发送方建立关联	
定义响应，事件的方法	对事件进行响应的方法（回调函数），该方法必须符合事件（委托）的要求	接收方

定义事件的语法：

[访问修饰符]　　event　　委托名　　事件名;

下面看一个例子，如图 5-38 所示。

```
public delegate void MyD(string mess); //定义委托
//事件发送者
   public class Publishe
   {
      public event MyD Event; //定义事件
      public void FireEvent(string notice)
      {
         if (Event != null) Event(notice);
      }
   }
```

图 5-38　事件定义

触发事件：通知其他对象，发生了某个事件，如图 5-39 所示。

图 5-39　触发事件

好处：事件发送者不需要事先知道都有哪些订阅者。

事件通信机制的优点：淡化了事件发送和事件接收两个对象之间的关系，使得两个类之间不须建立关联关系就可以进行通信（无连接）。

对象之间一般是通过调用方法来实现对象之间通信的，调用方法时需要用对象名做限定，如何知道对象名？可以在对象之间建立关联关系，即对象 A 作为对象 B 的属性，这样对象 B 就可以调用对象 A 的方法，从而对象 B 可以向对象 A 发送消息但对象之间建立的关联关系，增加了对象之间的耦合。

定义响应事件的方法如图 5-40 所示。

```
//事件接收者
  public class Receiver
  {
     string _name;
     public Receiver(string name)
     {
        _name = name;
     }
     public void Method(string s) //响应事件的方法（回调函数）
     {
        Console.WriteLine("{0}知道:{1}", _name, s);
     }
  }
```

图 5-40　定义响应事件方法

订阅事件如图 5-41 所示。

```
Publisher pub = new Publisher();
Receiver rece1 = new Receiver("阿花");
Receiver rece2 = new Receiver("阿福");
pub.Event += new MyD(rece1.Method);
pub.Event += new MyD(rece2.Method);
pub.FireEvent("面试时间到了！");
Console.ReadLine();
```

事件发生

图 5-41　订阅事件

事件发生如图 5-42 所示。

```
Publisher pub = new Publisher();
Receiver rece1 = new Receiver("阿花");
Receiver rece2 = new Receiver("阿福");
pub.Event += new MyD(rece1.Method);
pub.Event += new MyD(rece2.Method);
pub.FireEvent("面试时间到了！");
Console.ReadLine();
```

事件发生

```
public class Publishe
{
 public event MyD Event; //定义事件
 public void FireEvent(string notice)
{
if (Event != null) Event(notice);
}
}
```

触发事件

利用委托调用，通知已订阅该事件的所有对象

图 5-42　事件发生

事件发送方：定义事件、触发事件。至于事件触发后做什么，即如何响应，在定义事件发送方时并不知道。

Windows 窗体的事件机制，就是利用事件驱动的方式进行工作的。例如：

Button 的 Click 事件：

（1）发送方：Button 控件

（2）接收方：某个窗体

为事件编写处理程序（回调函数）

- 订阅事件：*.designer.cs

（1）双击控件

（2）属性窗口中的事件列表

5.7　索引器

索引器是访问/修改类中数据的一种方法：

（1）像数组一样使用下标访问/修改类中的数据；

（2）索引器下标可以是 int，也可以是 string。

语法：

```
[访问修饰符]  数据类型  this[数据类型  标识符]
{
    get{……}
    set{……}
}
```

案例学习：索引器的应用一

下面的示例演示了索引器的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 索引器
{
   class Program
   {
      static void Main(string[] args)
      {
         Point p = new Point();
         p[0] = 1;
         p[1] = 2;
         p[2] = 3;
```

```
            Console.WriteLine("Point.x={0}", p[0]);
            Console.WriteLine("Point.y={0}", p[1]);
            Console.WriteLine("Point.z={0}", p[2]);
            Console.ReadLine();
        }
    }
    class Point
    {
        private float x, y, z;
        //定义索引器，使得可以使用数组的方式访问 Vector 的数据
        public float this[int i]
        {
            get
            {
                switch (i)
                {
                    case 0:
                        return x;
                    case 1:
                        return y;
                    case 2:
                        return z;
                    //若访问下标超出范围，抛出异常
                    default:
                        throw new IndexOutOfRangeException("下标超出范围");
                }
            }
            set
            {
                switch (i)
                {
                    case 0:
                        x = value; break;
                    case 1:
                        y = value; break;
                    case 2:
                        z = value; break;
                    //若访问下标超出范围，抛出异常
                    default:
                        throw new IndexOutOfRangeException("下标超出范围");
                }
            }
        }
```

```
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-43 所示。

图 5-43　输出结果

案例学习：索引器的应用二

下面的示例演示了索引器的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 索引器 1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 创建一个容量为 3 的相册
            Framework friends = new Framework(3);
            // 创建 3 张照片
            Picture first = new Picture("幼年相片");
            Picture second = new Picture("成年相片");
            Picture third = new Picture("老年相片");
            // 向相册加载照片
            friends[0] = first;
            friends[1] = second;
            friends[2] = third;
            // 按索引检索
            Picture p1 = friends[2];
            Console.WriteLine(p1.Title);
            // 按名称检索
            Picture p2 = friends["幼年相片"];
            Console.WriteLine(p2.Title);
            Console.ReadLine();
        }
    }
    class Picture
```

```
    {
        string _title;
        public Picture(string title)
        {
            this._title = title;
        }
        public string Title
        {
            get
            {
                return _title;
            }
        }
    }
    class Framework
    {
        // 该数组用于存放照片
        Picture[] picture;
        public Framework(int capacity)
        {
            picture = new Picture[capacity];
        }
        //用整型的序号作为下标的索引器
        public Picture this[int index]
        {
            get
            {
                // 验证索引范围
                if (index < 0 || index >= picture.Length)
                {
                    Console.WriteLine("索引无效");
                    // 使用 null 指示失败
                    return null;
                }
                // 对于有效索引，返回请求的照片
                return picture[index];
            }
            set
            {
                if (index < 0 || index >= picture.Length)
                {
                    Console.WriteLine("索引无效");
                    return;
                }
                picture[index] = value;
            }
        }
        public Picture this[string title]
        {
            get
            {
                // 遍历数组中的所有照片
                foreach (Picture p in picture)
                {
```

```
                    // 将照片中的标题与索引器参数进行比较
                    if (p.Title == title)
                        return p;
                }
                Console.WriteLine("未找到");
                // 使用 null 指示失败
                return null;
            }
        }
    }
}
```

在这个程序例子里，先看一下带有 int 参数的索引器定义，如图 5-44 所示。

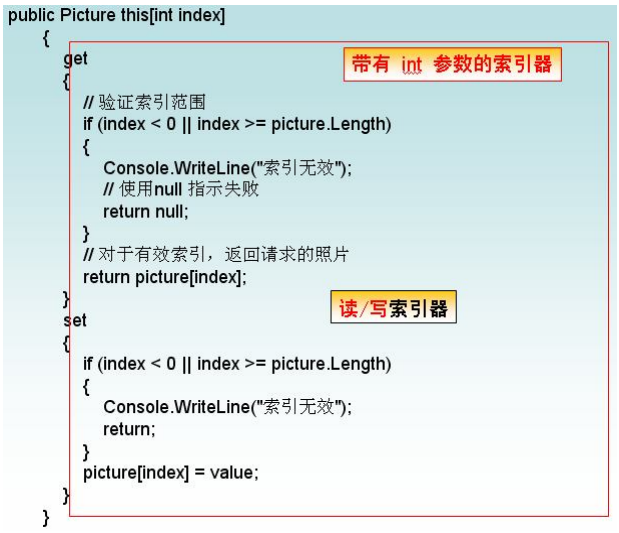

图 5-44　带有 int 参数的索引器定义

再看一下带有 string 参数索引器，如图 5-45 所示。

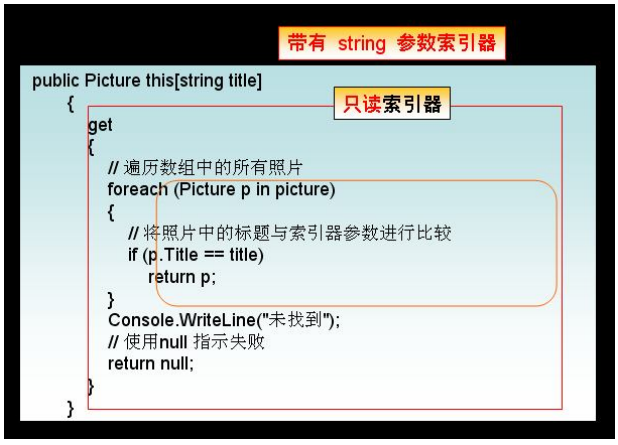

图 5-45　带有 string 参数的索引器定义

索引器的使用，如图 5-46 所示。

```
// 向相册加载照片
friends[0] = first;
friends[1] = second;
friends[2] = third;
// 按索引检索
Picture p1 = friends[2];
Console.WriteLine(p1.Title);
// 按名称检索
Picture p2 = friends["幼年相片"];
Console.WriteLine(p2.Title);
Console.ReadLine();
```

图 5-46　索引器的使用

按 F5 键调试、运行程序，输出结果如图 5-47 所示。

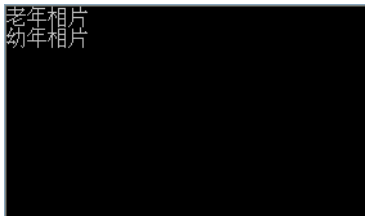

图 5-47　输出结果

5.8　异常处理

（1）异常：是指程序运行时发生的错误。

（2）报告错误（异常）：运行时的错误要及时地通知给程序进行必要的处理。

（3）异常对象与抛出异常：CLR 将自动收集运行时的错误信息，封装成对象（异常对象）来报告错误。这种报告错误的方法称为抛出异常。

（4）异常处理：C#用 try…catch 语句来捕捉系统抛出的异常对象，根据错误的内容进行相应的处理。

两种类型的异常：

（1）系统异常：

1）对常见的错误定义了相应的异常类；

2）发生运行时错误，由 CLR 自动创建异常对象并抛出。

（2）应用程序异常：

1）根据需要定义异常类（从 ApplicationException 继承）；

2）程序员根据需要在应用程序的代码中，创建异常对象并抛出 System.Exception 类。

在 C#中，异常用类来表示，所有异常类都必须从内部异常类 Exception 派生而来，而 Exception 是 System 名字空间的一部分。因此所有异常都是 Exception 的子类。

异常对象的结构，如图 5-48 所示。

异常的表示，即如何描述错误信息？用类来描述错误信息，即异常类。通过类的使用窗口，认识各种类型的异常对象。Exception 是所有异常的基类，ApplicationException 是应用程序异常的基类，它从 Exception 继承，但并没有增加新的功能，其目的是为了区分系统异常。

异常对象通过属性来描述错误信息，Exception 类的常用属性，如表 5-7 所示。

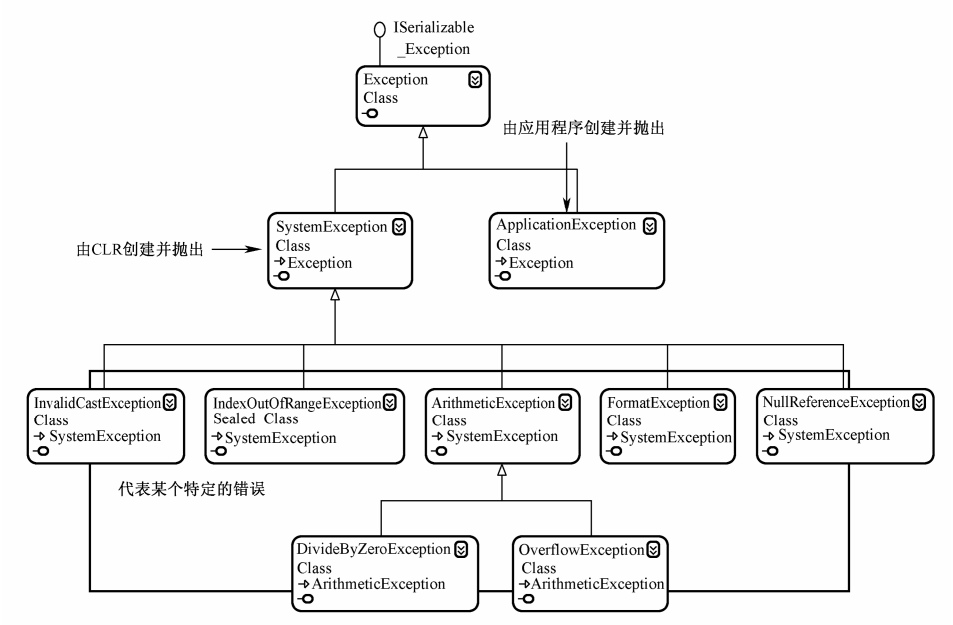

图 5-48　异常类的层次结构

表 5-7　Exception 类的常用属性表

属性	说明
Message	描述当前异常的错误信息（只读）
Data	存储用户定义的其他异常信息（2.0 新增，键/值对的集合类型）
HelpLink	获取或设置关联的帮助文件链接
Source	导致错误的对象名称或产生异常的程序集名称
TargetSite	产生当前异常的方法名称

系统抛出的异常，程序是如何获得的呢？也就是说，异常对象怎样产生的呢？我们看一个组合语句——try-catch-finally。

当用异常对象报告错误后，只有 try-catch 中的 catch 语句块能够捕获异常对象并进行处理。try-catch 的语法，如图 5-49 所示。

```
try
{
//捕获运行时错误代码
}

catch (Exception ee)
{
//响应处理错误代码
}
finally
{
//清理工作代码
}
```

图 5-49　try-catch 的语法

异常处理的执行流程，如图 5-50 所示。

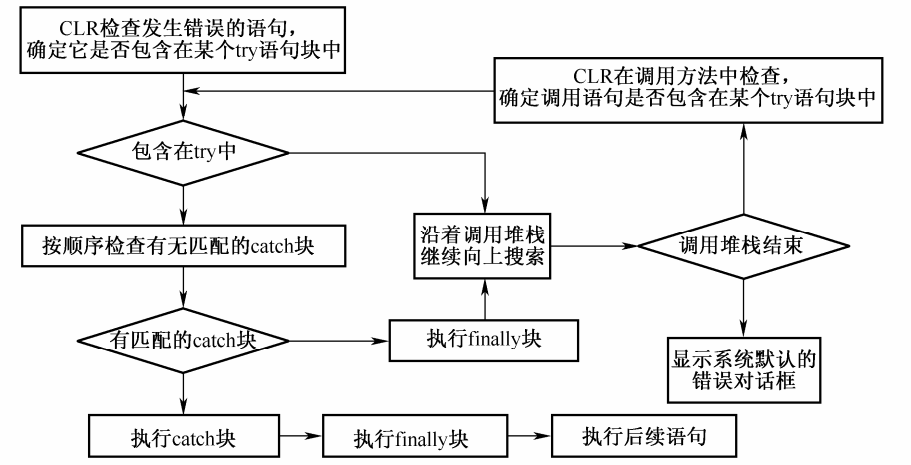

图 5-50　异常处理的执行流程

（1）在 try 中，发生错误的语句行之后的语句不会执行；

（2）在 try-catch 中只能执行一个 catch 块；

（3）如果没有匹配的 catch 对错误进行处理，异常对象将沿调用堆栈向上传播。

首先看第一种语法组合——try-catch 语句。

try-catch 语句的语法细则（一），如表 5-8 所示。

表 5-8　各语句块的作用表

语句块	作用
try	把可能发生错误的语句放在 try 语句块中进行保护
catch	捕获 try 语句块中抛出的异常对象并进行处理，可以有多个 catch 语句块
finally	清理和释放 try 块中的资源，finally 语句块是可选的

try-catch 语句的语法细则（二），如表 5-9 所示。

表 5-9　各语句块的执行表

语句块	执　　行
Try	把可能发生错误的语句放在 try 语句块中进行保护，如果某行语句发生错误，其后的语句将不会执行
Catch	只执行与异常对象匹配的第一个 catch 语句块
Finally	无论什么情况，都会执行

try-catch 语句的语法细则（三）——catch 块

（1）异常筛选器：

1）指定 catch 块要捕获的异常类型；

2）对于没有异常筛选器的 catch 块，可以捕捉任何类型的异常。

（2）匹配 catch 块：

1）异常筛选器中指定的异常类型与异常对象的类型相同；

2）异常筛选器中指定的异常类型与异常对象基类的类型相同。

（3）对于多个 catch 块，异常筛选器的排列顺序：

1）排列规则为派生类在前，基类在后；

2）异常筛选器的排列顺序是编译器的检查范围。

将具有最具体的（派生程度最高的）异常类的 catch 块放在最前面。

对于 try 和 catch 块的理解，如图 5-51 所示。

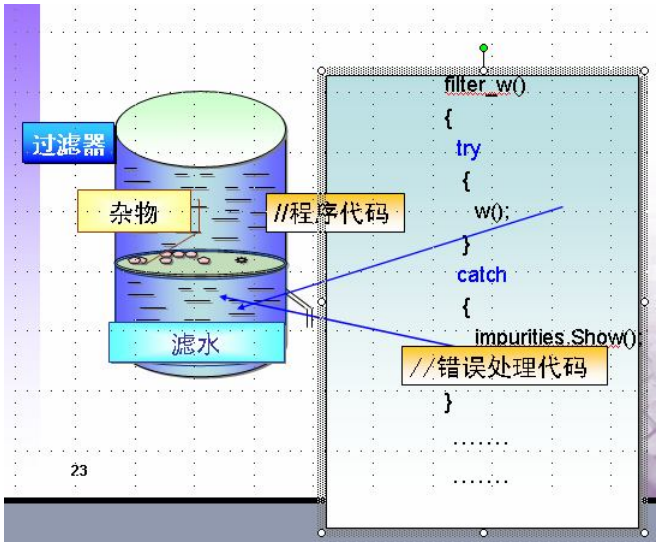

图 5-51　try 和 catch 块的理解

catch 后面还可以定义各类异常对象，不同异常类的对象可以获取不同类型的异常信息，如图 5-52 所示。

图 5-52　IOException 处理的异常类型

而 System.Exception 类可处理系统中的任何一种异常，如图 5-53 所示。

异常的获得有两种途径，一种是系统抛出，另一种是由程序主动抛出，对于程序主动抛出这种情况，如下段程序所示：

```
if (input < 1 && input > 50)
{
     throw new InvalidNumberInput("请输入 1 和 50 之间的数字");
}
```

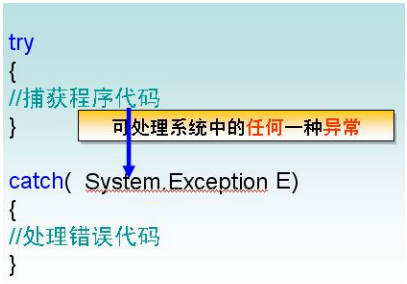

图 5-53　Exception 处理的异常类型

这段程序里，throw 可用来引发自定义异常 InvalidNumberInput，也可以抛出其他异常对象。

我们看一个使用 try-catch 语句的程序例子，代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 异常处理
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int input;
                InvalidNumberInput i = new InvalidNumberInput();
                input = int.Parse(Console.ReadLine());
                i.shuru(input);
            }
            catch (InvalidNumberInput ee)
            {
                Console.WriteLine(ee.Message);
            }
            Console.ReadLine();
        }
    }
    public class InvalidNumberInput : System.ApplicationException
    {
        string mess;
        public InvalidNumberInput() : base() { }
        public InvalidNumberInput(string message)
            : base(message)
        {
            mess = message;
```

```
        }
        public void shuru(int input)
        {
            if (input < 1 && input > 50)
            {
                throw new InvalidNumberInput
                ("请输入 1 和 50 之间的数字");
            }
        }
    }
}
```

该程序中的 InvalidNumberInput 是一个自定义的异常类，用来处理输入数字不符合要求的情况。它继承于应用程序异常类 ApplicationException。

下面再看一下 try-catch 语句使用 finally 的情况，如图 5-54 所示。

图 5-54　使用 finally 的情况

还有一种多 catch 块的使用情况，如图 5-55 所示。

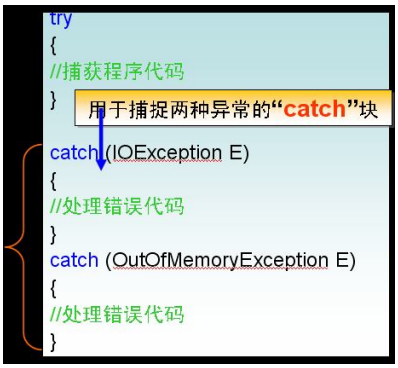

图 5-55　多 catch 块的使用情况

案例学习：多重 catch 块的应用

下面的示例演示了多重 catch 块的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 异常处理1
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int dividend = int.Parse(Console.ReadLine());
                int divisor = int.Parse(Console.ReadLine());
                MyCustomException my = new MyCustomException();
                my.chushu(divisor);
                int result = dividend / divisor;
                Console.WriteLine("两个数相除，结果为：{0}",result);
            }
            catch (MyCustomException ee)
            {
                Console.WriteLine(ee.Message);
            }
            Console.ReadLine();
        }
    }
    public class MyCustomException : System.ApplicationException
    {
        string mess;
        public MyCustomException() : base() { }
        public MyCustomException(string message)
            : base(message)
        {
            mess = message;
        }
        public void chushu(int divisor)
        {
            if (divisor == 0)
            {
                throw new MyCustomException("除数不能为零");
            }
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-56 所示。

图 5-56　输出结果

案例学习：自定义异常的应用

下面的示例演示了自定义异常的用法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace 异常处理2
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                string name = Convert.ToString(Console.ReadLine());
                string email = Convert.ToString(Console.ReadLine());
                EmailException e = new EmailException();
                e.Save(name, email);
            }
            catch (EmailException ee)
            {
                Console.WriteLine(ee.Message);
            }
            Console.ReadLine();
        }
    }
    public class EmailException : ApplicationException
    {
        public string _message;
        //重写构造函数
        public EmailException()
            : base()
        {
            _message = null;
        }
        public EmailException(string message)
            : base()
        {
            _message = message.ToString();
        }
```

```
        public EmailException(string message,
            Exception myNew)
            : base(message, myNew)
        {
            _message = message.ToString();
        }
        //Message 属性的重载
        public override string Message
        {
            get
            {
                return "Email 格式错误。";
            }
        }
        public bool Save(string name, string email)
        {
            string[] subStr= email.Split('@');
            //如果输入的 Email 不是被"@"字符分割成两段，则抛出 Email 错误异常
            if (subStr.Length != 2)
            {
                throw new EmailException();
            }
            else
            {
                int index = subStr[1].IndexOf(".");
                if (index <= 0)
                {
                    throw new EmailException();
                }
                if (subStr[1][subStr[1].Length - 1] == '.')
                {
                    throw new EmailException();
                }
            }
            return true;
        }
    }
}
```

按 F5 键调试、运行程序，输出结果如图 5-57 所示。

图 5-57　输出结果

5.9　组件与程序集

软件行业中的术语“组件”常用于指可重用的、以标准化方式向客户端公开一个或多个接口的对象。一个组件可作为一个类实现，也可作为一组类实现；主要要求是完善定义基本的公共接口。

组件的分类：

（1）可视化组件（Visual Component）：可视化组件在运行期间用户是可以看到的，也称控件（control）。

（2）非可视化组件（Nonvisual Component）：非可视化组件是指用户在运行期间是看不到的。

在.NET Framework 编程中最常用的一些组件是添加到 Windows 窗体中的可视控件，如 Button 控件（Windows 窗体）、ComboBox 控件（Windows 窗体）等。非可视组件包括 Timer Control、SerialPort 和 ServiceController 以及其他组件。

程序集是任何.NET Framework 应用程序的基本构造块。例如，在生成简单的 C#应用程序时，Visual Studio 创建一个单个可移植可执行（PE）文件形式的程序集，明确地说就是一个 EXE 或 DLL。

程序集包含描述它们自己的内部版本号和它们包含的所有数据和对象类型的详细信息的元数据。

程序集仅在需要时才加载。如果不使用程序集，则不会加载。这意味着程序集可能是在大型项目中管理资源的有效途径。

程序集可以包含一个或多个模块。

5.10　本章小结

- 继承是获得现有类的功能的过程。
- 创建新类所根据的基础类称为基类或父类，新建的类则称为派生类或子类。
- base 关键字用于从派生类中访问基类成员。
- override 关键字用于修改方法、属性或索引器。new 访问修饰符用于显式隐藏继承自基类的成员。
- 抽象类是指至少包含一个抽象成员（尚未实现的方法）的类。抽象类不能实例化。
- 重写方法就是修改基类中方法的实现。virtual 关键字用于修改方法的声明。
- 接口只包含方法、属性、索引器（有参属性）、事件四种成员。方法的实现是在实现接口的类中完成的。
- 组件是管理代码编写、在通用语言运行时构建的，组件向开发人员提供了一个全新的混合开发环境。

课后习题

一、编程题

1．创建一个研究生类，派生自学生类。学生具有属性：学号和姓名，研究生除了有学号和姓名以外，还具有工资属性。

2．使用运算符重载技术实现复数的相加。

二、选择题

1．分析下列程序中类 MyClass 的定义

```
class BaseClass
{
public int i;
}
class MyClass:BaseClass
{
public new int i;
}
```

则下列语句在 Console 上的输出为（　　）。

```
MyClass y = new MyClass();
BaseClass x = y;
x.i = 100;
Console.WriteLine("{0}, {1}",x.i,y.i);
```

（提示：注意类 MyClass 中的 new 关键字）

A．0, 0　　B．100, 100　　C．0, 100　　D．100, 0

2．在定义类时，如果希望类的某个方法能够在派生类中进一步进行改进，以处理不同的派生类的需要，则应将该方法声明成（　　）。

A．sealed 方法　　B．public 方法　　C．Virtual 方法　　D．override 方法

3．类 MyClass 中有下列方法定义：

```
public void testParams(params int[] arr)
{
Console.Write ("使用 Params 参数！");
}
public void testParams(int x,int y)
{
Console.Write ("使用两个整型参数！");
}
```

请问上述方法重载有无二义性？若没有，则下列语句的输出为（　　）。

```
MyClass x = new MyClass();
x.testParams(0);
x.testParams(0,1);
x.testParams(0,1,2);
```

A．有语义二义性

B．使用 Params 参数！使用两个整型参数！使用 Params 参数

C．使用 Params 参数！使用 Params 参数！使用 Params 参数

D．使用 Params 参数！使用两个整型参数！使用两个整型参数

4．C#程序中，可使用 try-catch 机制来处理程序出现的（　　）错误。

A．语法　　B．运行

C．逻辑　　D．拼写

5．C#中，在方法 MyFunc 内部的 try-catch 语句中，如果在 try 代码块中发生异常，并且在当前的所有 catch 块中都没有找到合适的 catch 块，则（　　）。

A．.NET 运行时忽略该异常

B．.NET 运行时马上强制退出该程序

C．.NET 运行时继续在 MyFunc 的调用堆栈中查找提供该异常处理的过程

D．.NET 抛出一个新的“异常处理未找到”的异常

6．下列（　　）不是接口成员。

A．方法　　B．属性

C．事件　　D．字段