第3章 值和方法

程序处理的是内存中的数据，这些数据的表示形式分为常量和变量；方法用于对数据进行各种操作，方法代码中出现的变量可分为局部变量、字段和参数。本章详细介绍C#语言中的常量、变量、字段、方法和委托的概念，并说明对类型成员的访问限制机制。

3.1 常量和变量

3.1.1 常量

常量指的是固定的值。例如，10是整数类型的常量，0.5是实数类型的常量，true是布尔类型的常量，等等。这些常量值可以直接放入代码中。例如，下面的语句分别输出了整数常量、字符常量和字符串常量的值。

```
Console.WriteLine(10);
Console.WriteLine('A');
Console.WriteLine("Apple");
```

字符串类型是一种特殊的引用类型；除了它之外，其他引用类型的常量只有一个，那就是空值null。

3.1.2 变量

变量是程序中的数据存储单元，每一个变量都是其类型的一个对象，其值在运行时可以被改变。变量必须先定义后使用。例如，下面的代码先定义了一个整数变量$x1$，而后通过常量10为其赋值：

```
int x1;
x1 = 10;
```

C#语言允许将上面两行代码合并为一行。例如：

```
int x1 = 10;
```

进一步，多个同类型变量的声明和赋值也可以合并为一行代码，变量之间以逗号分隔：

```
int x1 = 10, x2 = 5;
```

此外，变量一经定义，其类型就不能改变。例如，下面的代码不能通过编译。

```
int x3 = 10;
x3 = new Double();                          //错误：整数类型的变量不能变为实数类型
```

提示：C#语言要求变量的名称必须以字母开头，并由字母、数字和下划线这3种字符构成，而且不能与C#中的关键字或库函数的名称(如using、int、Main等)相同。

C#从3.0开始引入了一个关键字var，它用于“隐式”变量声明，即在声明变量时不用明确指定其类型。例如：

```
var x1 = 10;
var x2 = 0.5;
```

对于上述代码，编译器会自动推断$x1$为int类型，$x2$为double类型，之后对它们的使用与标准声明的变量并没有什么不同。如果类型名较长，那么使用隐式变量声明能够有效简化代码，而且很多时候还能减少对其他命名空间的引用。但一般情况下，过度使用var会影响程序的可理解性。

再看下面的代码段，其中出现了3个double类型的变量。

```
double x;
double y = 0.3;
double z = y + 0.2;
```

其中，变量y和z是分别通过常量和表达式来进行赋值。不过变量x只是声明，还没有真正被分配内存；如果将最后一行代码中的y换成x，那么程序不能通过编译，错误原因是“使用了未赋值的局部变量x”。

对于类的变量，要注意“未赋值”和“空值null”之间的区别。例如，下面的代码是错误的，因为字符串类型的变量s还没有被赋值。

```
string s;
Console.WriteLine(s);                       //错误：使用了未赋值的变量
```

而如果换成如下的代码就是正确的，运行时控制台会输出一个空行：

```
string s = null;
Console.WriteLine(s);
```

对于引用类型的变量而言，如果未赋值就访问其成员，那么代码不能通过编译；而如果在变量值为null时访问其成员，代码能够通过编译，但在程序运行时会发生异常。例如，在下面的程序中，Main方法的最后一行代码在输出对象c1时会出错，原因是“未将对象引用设置到对象的实例”。

```
//程序清单 P3_1.cs
using System;
namespace P3_1
{
    class Program
    {
        static void Main()
```

```
        {
            Contact c1 = new Contact() { Name = "王小明", Phone = "321000" };
            PrintContact(c1);
            c1 = null;
            PrintContact(c1);
        }

        static void PrintContact(Contact c)
        {
            Console.WriteLine("{0},{1}", c.Name, c.GetGenderChar());
            Console.WriteLine("Phn: {0}", c.Phone);
            Console.WriteLine("Adr: {0}", c.Address);
        }
    }

    class Contact
    {
        public string Name;
        public bool Gender;
        public string Phone;
        public string Address;

        public char GetGenderChar()
        {
            if (Gender) return '男';
            else return '女';
        }
    }
}
```

要改正这一错误,可以将方法 PrintContact 的代码改为如下内容:

```
static void PrintContact(Contact c)
{
    if (c == null)
        Console.WriteLine("联系人对象为空");
    else
    {
        Console.WriteLine("{0},{1}", c.Name, c.GetGenderChar());
        Console.WriteLine("Phn: {0}", c.Phone);
        Console.WriteLine("Adr: {0}", c.Address);
    }
}
```

重新编译运行程序 P3_1,可以看到输出结果为:

```
王小明, 女
Phn: 321000
Adr:
联系人对象为空
```

上面讨论的变量均指在方法的代码中定义的变量，其作用范围只限于当前方法：方法执行到创建变量的代码时就分配内存，而方法执行完毕后就从内存中清除，因此它们也叫做局部变量。一个方法的执行代码中不允许出现两个同名的变量；但不同方法中的变量不会存在命名冲突。下一节将讨论字段型变量，它们是对象的数据成员。

3.2 字段

3.2.1 实例字段

在类或结构中定义的一般数据成员叫做实例字段。在创建类或结构的对象时，字段也就拥有了自己的值。类的定义中可以为字段指定初始值。例如：

```
class Rectangle
{
    public int Width = 1;
    public int Height = 1;
}
```

那么，使用表达式 new Rectangle()创建对象，其两个字段的初始值均为 1。但要注意这种方式只能用于类，而不能用于结构。另一种初始化字段的方式是使用对象创建表达式或构造函数，其中，构造函数的用法将在第 6 章中介绍。

如果未采用任何一种方式为字段指定初始值，那么在创建对象后，字段将取其类型的默认值，这一点和局部变量是不同的。对于所有的整数(包括字符和枚举)类型和实数类型，其默认值为 0；布尔类型的默认值为 false；而所有的引用类型的默认值为 null。例如，对于程序 P3_1 中的 Contact 类，直接创建对象时，Gender 字段值默认为 false，其他三个 String 类型的字段值则为 null。为验证这一点，可在程序 P3_1 的 Main 方法最后加上如下代码：

```
c1 = new Contact();
PrintContact(c1);
```

那么，上述最后一行代码对应的输出结果为：

```
，女
Phn:
Adr:
```

此外，和局部变量不同，字段在定义时必须明确其类型，而不能使用 var 进行隐式定义。

3.2.2 静态字段

实例字段归类型的对象所有。如果在字段定义时再增加一个 static 修饰符，那么，表示字段是静态的，它归类型本身所有，而与类型的具体对象无关。

例如，下面的 Account 类就定义了一个静态字段 Cur 以及一个实例字段 Money：

```
class Account
{
    public static string Cur = "人民币";            //静态字段
    public decimal Money = 1000;                    //实例字段
}
```

和实例成员不同,访问静态成员时,圆点连接符的前面不再是某个具体的对象名,而是类型的名称。下面的 Query 方法代码演示了对两种字段的不同访问方式。

```
class Program
{
    static void Query(Account a)
    {
        Console.WriteLine("余额{0}{1}", a.Money, Account.Cur);
    }
}
```

不过,如果是在所属类型的成员方法中访问,无论是实例字段,还是静态字段都可以只使用字段名,而不需要使用对象名或类型名作为前缀。例如:

```
class Account
{
    public static string Cur = "人民币";            //静态字段
    public decimal Money;

    public void Query()
    {
        Console.WriteLine("余额{0}{1}", Money, Cur);
    }
}
```

实例字段的生命周期随对象的创建而开始,随对象的销毁而结束。静态字段的生命周期则从程序中第一次使用到其所属类型开始,直到该类型的所有对象都被销毁才结束。对于类的实例字段,每创建一个类的对象,都在内存中开辟了一块区域用于存储该字段;而类的静态字段为这个类的所有对象所共享,无论该类创建了多少个实例,一个静态字段在内存中只有一份存储。参见下面的程序示例。

```
//程序清单 P3_2.cs
using System;
namespace P3_2
{
    class Program
    {
        static void Main()
        {
            Account acc1 = new Account() { ID = "001" };
            Account acc2 = new Account() { ID = "002", Money = 1500 };
            BankCard card1 = new BankCard() { ID = "A001" };
            card1.Pay(500);
            acc1.Query();
```

```
            acc2.Query();
            card1.Query();
            Account.Cur = "港币";
            acc1.Query();
            acc2.Query();
            card1.Query();
        }
    }

    class Account
    {
        public static string Cur = "人民币";        //静态字段
        public string ID;
        public decimal Money = 1000;

        public void Query()
        {
            Console.WriteLine("账户{0} 余额{1}{2}", ID, Money, Cur);
        }
    }

    class BankCard: Account
    {
        public bool Pay(decimal price)
        {
            if (Money >= price)
            {
                Money = Money - price;
                return true;
            }
            else
                return false;
        }
    }
}
```

在 Main 方法代码中，通过 Pay 方法修改了 card1 对象的实例字段 Money 后，其他对象的 Money 值不会受影响；而在修改了 Account 类的静态字段 Cur 后，通过所有 Account 对象（包括其派生对象）访问到的字段值都发生了变化。程序 P3_2 的输出结果如下：

```
账户 001 余额 1000 人民币
账户 002 余额 1500 人民币
账户 A001 余额 500 人民币
账户 001 余额 1000 港币
账户 002 余额 1500 港币
账户 A001 余额 500 港币
```

3.2.3 常数和只读字段

如果字段是用于保存一些不发生变化的数值（例如数学上的圆周率 π、自然对数常数 e

等),那么还可以使用关键字 const 或 readonly 来修饰字段,这样就能够禁止对字段的修改。

先看 const 所修饰的常数字段,它必须在定义的同时进行赋值,且所赋的值必须是一个常量(包括简单值和空值 null)。例如,在表示"圆"的类型 Circle 中可以定义一个常数字段来记录圆周率。

```
class Circle
{
    public const double Pi = 3.14159;              //常数字段
}
```

const 所修饰的字段默认就是静态的(但不能再增加 static 修饰符),属于类型本身所有。因此,对常数字段的访问应通过类型进行,但不允许修改其值。

```
Console.WriteLine(Circle.Pi);
Circle.Pi = 3.5;                                   //错误: 不允许修改常数字段
```

从理论上说,常数字段可以是任何类型;但对于除字符串之外的其他引用类型,其常数字段在初始化时都只能被赋值为 null,这在实际程序中没有什么意义。因此,const 修饰的常数字段一般都是简单值类型或字符串类型。例如,下面的代码都是错误的。

```
public const object o = new object();              //错误: 常数字段必须被赋值为常量
public const int[] a = new int[] { 1, 2, 3 };      //错误: 不能定义数组型常数字段
```

readonly 所修饰的字段称为只读字段,但它默认是实例字段(当然也可使用 static 关键字将其修饰为静态只读字段)。只读字段可在字段定义时或在构造函数中进行初始化,但不能在对象创建表达式中赋值。例如,对于单一货币的账户类型,其币种信息可以作为只读字段存储。

```
class Account
{
    public readonly string Cur = "人民币";         //静态字段
    public string ID;
    public decimal Money = 1000;

    public void Query()
    {
        Console.WriteLine("账户{0} 余额{1}{2}", ID, Money, Cur);
    }
}
```

只读字段既可以是值类型,也可以是引用类型。例如,下面的代码都是合法的。

```
private readonly object o = new object();
public readonly int[] array = new int[] { 1, 2, 3 };
```

对于只读字段而言,除了其所属类或结构的构造函数外,不允许在其他任何地方修改其值。

那么,如果将只读字段也定义为静态的,它和常数字段有什么区别呢?答案是,对于常数字段,其值是在编译时确定的,在编译后的程序中是一个"常数",程序每次运行的值都是一样的;而静态只读字段的值是在运行时才确定,因此,程序每次运行的值不一定相同。参

见下面的程序示例。

```
//程序清单 P3_3.cs
using System;
namespace P3_3
{
    class Program
    {
        static void Main()
        {
            RandNum r1 = new RandNum();
            Console.WriteLine("{0}, {1}, {2}", RandNum.X, RandNum.Y, r1.Z);
            RandNum r2 = new RandNum();
            Console.WriteLine("{0}, {1}, {2}", RandNum.X, RandNum.Y, r2.Z);
        }
    }

    class RandNum
    {
        static Random rand = new Random();
        public const int X = 100;
        public static readonly int Y = rand.Next(100);
        public readonly int Z = rand.Next(100);
    }
}
```

上述代码中用到了一个 Random 类，它在 System 程序集中定义，主要用于生成各种伪随机数，其方法 Next(n)将随机返回 0 到 n 之间的一个数。程序 P3_3 的 RandNum 类使用它为只读字段 Y 和 Z 随机赋值，那么，在程序中创建不同的 RandNum 对象，对象的常数字段 X 和静态只读字段 Y 的值总是相同的，而只读字段 Z 的值则不尽相同；但如果多次运行程序，X 的值不会变化，Y 的值就会发生变化。图 3.1 演示了程序 P3_3 多次运行的输出结果。

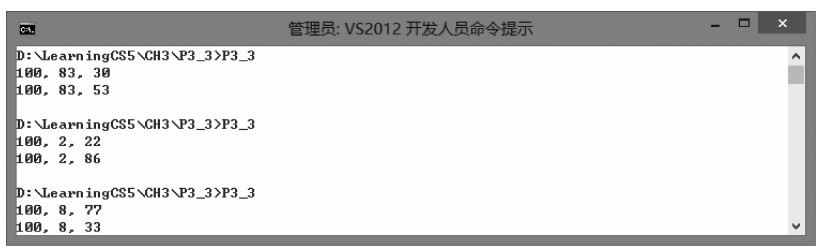

图 3.1　程序 P3_3 的运行结果

3.3　方法

3.3.1　方法的定义和调用

1. 方法的定义

方法代表了对象所能提供的操作或服务。方法的定义包括四部分，它们按顺序依次是：

返回类型、方法名、一对小括号中的参数列表以及一对大括号括起来的执行体。如果没有返回值,那么返回类型为 void。如果没有参数,小括号中的内容为空;如有多个参数则相互之间以逗号分隔。

下面的 Circle 类中定义了两个公有方法,一个有参数而无返回值,另一个无参数而有返回值。

```
class Circle
{
    public const double Pi = 3.14159;
    public double R = 1.0;

    public void Scale(double d)
    {
        R = R * d;
    }

    public double GetArea()
    {
        return Pi * R * R;
    }
}
```

方法的执行体中通过 return 语句来返回值,return 关键字之后的表达式应与方法定义的返回类型保持一致。除非返回类型为 void,否则执行体代码中必须出现 return 语句。只要执行到 return 语句,那么就结束对方法的调用,剩余的代码将被跳过。编译程序时,如果在无条件的 return 语句后仍存在代码,编译器会警告“检测到不可达的代码”。

对于无返回值的方法来说,其执行代码中可以没有 return 语句,也可以包含单独的 return 语句。例如:

```
public void Scale(double d)
{
    if (d <= 0)
    {
        Console.WriteLine("输入倍数不正确");
        return;
    }
    R = R * d;
    Console.WriteLine("圆被扩大{0}倍", d);
}
```

2. 形参和实参

方法定义中声明的参数叫做形式参数(简称形参),而实际调用时传递给方法的参数叫做实际参数(简称实参)。调用方法时,实参的名称不一定要和形参相同,但类型必须与形参的类型相同,或是能够隐式转换为形参的类型。例如,在上面定义的 Scale 方法中,d 叫做形参,而在下面调用方法的代码中,变量 d 和 $d1$ 分别作为实参传递给 Circle 对象的 Scale 方法。

```
Circle c = new Circle() { R = 2.0 };
double d = 1.5, d1 = 2.0;
c.Scale(d);
c.Scale(d1);
```

3. 方法的递归调用

方法的执行代码中常常要调用其他的方法，而一种特殊的情况是调用方法本身，这叫做递归调用。例如，一个整数的阶乘可以写成 $n! = n * (n-1)!$，那么，计算阶乘的方法代码可以写成如下形式：

```
public int Fact(int n)
{
    if (n > 1) return n * Fact(n - 1);
    else return 1;
}
```

递归定义的方法应当具备终止条件。例如，上面 Fact 方法的终止条件就是 $n \leqslant 1$；如果方法中只有一行执行代码“return n * Fact(n－1);”，那么递归调用就会无休止地继续下去，直至程序因为“堆栈溢出”的错误而崩溃。

在对数组等集合型数据进行操作时，很多情况下需要定义两个方法，一个根据集合范围来递归地调用自身，另一个则直接调用前一个递归方法。下面的程序演示了如何通过递归方法来计算一个数组的和，注意，其中的 Agg 类的 Sum 方法用于计算数组中前 k 个元素之和，其执行代码又调用了自身。

```
//程序清单 P3_4.cs
using System;
namespace P3_4
{
    class Program
    {
        static void Main()
        {
            Agg a = new Agg();
            double[] x = { 73, 68, 56, 84, 92 };
            Console.WriteLine("数组的和为：{0}", a.Sum(5, x));
        }
    }

    class Agg
    {
        public double Sum(int k, double[] array)
        {
            if (k <= 1)
                return array[0];
            else
                return array[k - 1] + Sum(k - 1, array);
        }
    }
}
```

如果计算量比较大,那么,递归方法的执行效率可能会很低。一般可以使用循环语句以及有效的数据结构来取代递归形式。循环语句将在第 5 章中详细介绍。

3.3.2 参数类型

1. 引用型参数

如果参数的类型为值类型,在默认情况下,方法执行体中的代码只能改变形参的值,而不能改变实参的值。下面的程序就定义了一个 Swap 方法,它试图通过一个临时变量 temp 来交换 x 和 y 的值。

```
//程序清单 P3_5.cs
using System;
namespace P3_5
{
    class Program
    {
        static void Main()
        {
            double x = 5, y = 10;
            Console.WriteLine("交换前 x={0}, y={1}", x, y);
            Swap(x, y);
            Console.WriteLine("交换后 x={0}, y={1}", x, y);
        }

        public static void Swap(double x, double y)
        {
            double temp = x;
            x = y;
            y = temp;
        }
    }
}
```

但运行程序,可以看到程序的输出结果如下:

```
交换前 x=5, y=10
交换后 x=5, y=10
```

这说明交换没有成功。其原因是在调用 Swap 方法时,实参 x 和 y 的值会被复制给方法的形参,而方法执行代码中所改变的只是形参的值,而不会修改原始的实参值。

上面这种传递参数的方式叫做值传递。如果希望改变实参的值,则需要另一种参数类型——引用型参数,定义时是在形参的类型前加上一个关键字 ref,这样参数的传递方式就变成了引用传递。例如:

```
public static void RefSwap(ref double x, ref double y)
{
    double temp = x;
```

```
    x = y;
    y = temp;
}
```

而在调用方法时,引用传递的实参之前也要加上 ref 关键字。

```
RefSwap(ref x, ref y);
```

这时传递给方法的就不是实参的拷贝,而是指向实参的引用。将程序 P3_5 中的 Swap 方法替换为 RefSwap 方法,这样才能成功地交换两个数值。

```
交换前 x = 5, y = 10
交换后 x = 10, y = 5
```

对于引用类型的参数,不使用 ref 关键字进行参数传递,会使得形参和实参指向同一个内存对象,对形参的修改同样会影响到实参;而如果使用了 ref 关键字,那么,调用时形参就会成为指向实参的引用(好比是指针的指针)。下面的程序中就定义了两个方法 EditAccount 和 ChangeAccount,它们的参数类型都是引用类型的 Account,不过后者使用了 ref 关键字。

```
//程序清单 P3_6.cs
using System;
namespace P3_6
{
    class Program
    {
        static void Main(string[] args)
        {
            Account a1 = new Account() { ID = "001" };
            EditAccount(a1);
            a1.Query();
            ChangeAccount(ref a1);
            a1.Query();
        }

        static void EditAccount(Account a)
        {
            a.Money = a.Money * 2;
            a = new Account() { ID = "002" };
        }

        static void ChangeAccount(ref Account a)
        {
            a.Money = a.Money * 2;
            a = new Account() { ID = "002" };
        }
    }

    class Account
```

```
    {
        public string ID;
        public decimal Money = 100;

        public void Query()
        {
            Console.WriteLine("账号{0} 余额{1}", ID, Money);
        }
    }
}
```

其中,EditAccount 方法使用的是普通参数,调用该方法能够修改 Account 对象的 Money 字段值;不过在该方法中将形参指向一个新对象后,实参仍为原对象。而 ChangeAccount 方法使用的是引用型参数,那么,在该方法中将形参指向一个新对象后,实参也指向了新对象。程序 P3_6 的输出结果为:

```
账号 001 余额 100
账号 002 余额 200
```

2. 输出型参数

我们知道方法可以返回值,但能否执行一次方法得到多个返回值?答案是肯定的,这时就需要用到输出型参数,定义时是在形参的类型前加上一个关键字 out,而后在方法的执行代码中将其他需要返回的结果放在形参中。例如,下面的 Circle 类型就定义了 Compute 方法,它通过 return 语句返回圆的面积,并通过输出型参数得到圆的周长。

```
class Circle
{
    public const double Pi = 3.14159;
    public double R = 1.0;

    public double Compute(out double p)
    {
        double t = Pi * R;
        p = 2 * t;
        return t * R;
    }
}
```

输出型参数也是采用引用传递的方式,但和引用型参数不同,输出型参数的形参在方法的执行体中必须被赋值,而这样的赋值会同样作用到实参,因为形参和实参指向的是同一引用。

调用方法时,输出型参数的实参前面也要加上 out 关键字。例如:

```
static void Main()
{
    Circle c = new Circle() { R = 7.2 };
    double p;
```

```
    double a = c.Compute(out p);
    Console.WriteLine("面积{0}, 周长{1}", a, p);
}
```

注意,上面的代码在定义变量 p 时并没有对其赋值,这是允许的,因为其后的方法调用会为 p 赋值。正是因为输出型的实参变量可能未被赋值,因此,方法的执行代码在对其进行赋值之前不能使用该参数。例如,下面的代码就是错误的。

```
public double Compute(out double p)
{
    double a = p * R;                                    //错误: 输出型参数在赋值前不能使用
    p = 2 * Pi * R;
    return a;
}
```

3. 数组型参数

回顾前面的程序 P3_4,其中的 Sum 方法使用了 double[]类型的参数 array,那么,可以将一个数组变量传递给该方法,但不能直接将一组数直接传递给方法。例如,下面的代码是错误的。

```
Agg a = new Agg();
double s = a.Sum(73, 68, 56, 84, 92);                   //错误: 参数必须是一个数组
```

如果希望以这种方式来调用方法,那么也很简单,只需要在方法定义中的参数类型 double[]之前加上关键字 params 即可,这样的参数叫做数组型参数。例如:

```
public double Sum(int k, params double[] array)
{
    if (k <= 1)
        return array[0];
    else
        return array[k - 1] + Sum(k - 1, array);
}
```

之后调用方法时既可以为其指定一个数组变量,也可以向其传递一组数组元素型的变量。例如,下面的代码都是合法的。

```
Agg a = new Agg();
double[] x = { 73, 68, 56, 84, 92 };
double s = a.Sum(5, x);
s = a.Sum(5, -2.83, 6.4, 0.57, 5.97, 3.2);
```

也就是说,数组型参数使得方法能够接受不同数量的参数。不过,在每个方法中最多只能定义一个数组型参数,而且该参数必须位于参数列表中的最后。

当方法中使用了数组型参数,且其元素类型为值类型时,如果实参是一个数组变量,则对其采用引用传递方式; 如果是一组变量,则对其采用值传递的方式。看下面的程序示例。

```
//程序清单 P3_7.cs
using System;
```

```
namespace P3_7
{
    class Program
    {
        static void Main()
        {
            int x0 = 1, x1 = 3, x2 = 5;
            AddFL(x0, x1, x2);
            Console.WriteLine("x0 = {0}, x2 = {1}", x0, x2);
            int[] x = { x0, x1, x2 };
            AddFL(x);
            Console.WriteLine("x[0] = {0}, x[2] = {1}", x[0], x[2]);
        }

        static void AddFL(params int[] array)
        {
            array[0] = array[0] + 1;
            array[array.Length - 1] = array[array.Length - 1] + 1;
        }
    }
}
```

方法 AddFL 将数组 array 的第一个和最后一个元素的值分别加 1。第一次调用方法时以 3 个整数共同作为实参,其值不会被改变;第二次调用方法时以一个数组为实参,数组元素的值会被改变。程序的输出结果如下:

```
x0 = 1, x2 = 5
x[0] = 2, x[2] = 6
```

3.3.3 方法的标识与重载

一个类型中不能有两个标识相同的成员。对于方法而言,方法名和参数列表共同组成了方法的标识,因此在一个类中允许存在两个同名的方法,只要方法的参数列表不完全相同(参数数量或类型不同)。这时称该方法具有同名的重载形式,即方法名相同而标识不同。例如,下面的 Swapper 类就定义了多个重载形式的 Swap 方法。

```
class Swapper
{
    public static void Swap(ref double x, ref double y)
    {
        double temp = x;
        x = y;
        y = temp;
    }

    public static void Swap(ref decimal x, ref decimal y)
    {
        decimal temp = x;
```

```
        x = y;
        y = temp;
    }

    public static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}
```

编译器在对程序进行编译时，将根据参数数量和类型来决定实际调用的是哪一个方法。例如，下面的代码将会调用类 Swapper 中定义的第三个 Swap 方法。

```
int x = 50, y = 10;
Swapper.Swap(ref x, ref y);
```

要注意的是，在.NET 公共语言规范中，方法的返回类型不足以对方法进行标识，即不允许两个方法的名称和参数列表完全相同，而仅仅是返回类型不同。例如，一个类中不能同时包含如下两个方法。

```
int Add(int x) { return x + 1; }
void Add(int x) { x = x + 1; }              //错误：返回类型不足以标识方法
```

此外，如果两个方法中的同名参数一个是普通类型，另一个是 ref 或 out 类型，那么将这两个参数视为不同类型。例如，下面定义的两个方法可以共存于一个类型中。

```
int Add(int x) { return x + 1; }
void Add(ref int x) { x = x + 1; }
```

但如果两个同名参数一个是 ref 类型，另一个是 out 类型，它们的区别不足以标识不同的方法。例如，下面定义的两个方法不能共存于一个类型中。

```
int Add(ref int x) { return x + 1; }
void Add(out int x) { x = 1; }              //错误：ref 和 out 的参数区别不足以标识方法
```

3.3.4 可选参数和命名参数

有些情况下，方法的重载仅仅是为了简化调用代码。例如，下面的 Rectangle 类定义了两个重载的 Scale 方法。

```
class Rectangle
{
    public double Width = 1, Height = 1;

    public void Scale(double d1)
    {
        Width = Width * d1;
    }
```

```
    public void Scale(double d1, double d2)
    {
        Width = Width * d1;
        Height = Height * d2;
    }
}
```

其中,第二个方法的功能其实已经涵盖了第一个方法。C#语言从4.0版本开始支持可选参数的概念,即允许在方法定义时为参数指定默认值;如果在调用时未指定这些参数值,那么就视这些参数的值为默认值。例如,可以为Rectangle类只定义一个如下的Scale方法。

```
public void Scale(double d1, double d2 = 1.0)
{
    Width = Width * d1;
    Height = Height * d2;
}
```

而在调用该方法时,为其指定一个或两个参数都是合法的:

```
Rectangle r1 = new Rectangle();
r1.Scale(1.5, 3.0);
r1.Scale(2.0);                          //相当于 r1.Scale(2.0, 1.0)
```

为了使编译器能够明确方法的调用形式,C#对可选参数的使用也有一些基本要求:

- 为可选参数指定的默认值必须是一个常量表达式。例如,不能是某个对象的字段值。
- 可选参数不能是引用型(ref)或输出型(out)参数。
- 如果方法中同时包含必选参数和可选参数,那么在参数列表中可选参数应放在必选参数之后。
- 如果方法中使用了多个可选参数,那么不能省略前面的参数值而去指定后面的参数值。

此外,如果一个类型中同时包含带可选参数的方法和不带可选参数的方法,而在调用时两种方法都符合要求,那么C#将优先调用不带可选参数的方法。例如,设Rectangle类同时定义了下面两个Scale方法,那么对Rectangle对象r1执行"r1.Scale(2.0)"这样的语句,实际调用的将是第一个Scale方法。

```
public void Scale(double d1)
{
    Width = Width * d1;
}

public void Scale(double d1, double d2 = 1.0)
{
    Width = Width * d1;
    Height = Height * d2;
}
```

提示：对于一个带可选参数的方法，C#编译器将其编译为.NET中间语言后，实际上得到的仍是多个重载的方法。

命名参数是指在方法调用中同时指定参数的名称和值，这样就不必考虑方法定义中参数出现的顺序。例如，下面的代码会将3.0传递给Scale方法的参数*d*1，将2.0传递给参数*d*2。

```
r1.Scale(d2: 2.0, d1: 3.0);
```

类似地，如果方法包含多个参数，而在调用时只对其中部分使用命名参数的方式，那么这些命名参数应出现在其他参数的后面，而一般参数仍会按照它们在方法定义中的位置来进行传递。

很多情况下，将可选参数和命名参数结合起来使用才能够发挥更大的作用。下面的小程序演示了这一点。

```
//程序清单 P3_8.cs
using System;
namespace P3_8
{
    class Program
    {
        static void Main()
        {
            Visit(u1: "download");
            Visit(u2: "sina");
            Visit(u0: "https", u2: "google");
            Visit(u2: "Tsinghua", u3: "edu.cn");
        }

        static void Visit(string u0 = "http", string u1 = "www", string u2 = "microsoft",
string u3 = "com")
        {
            Console.WriteLine("访问网站{0}://{1}.{2}.{3}", u0, u1, u2, u3);
        }
    }
}
```

程序P3_8的输出结果如下：

```
访问网站 http://download.microsoft.com
访问网站 http://www.sina.com
访问网站 https://www.google.com
访问网站 http://www.Tsinghua.edu.cn
```

3.3.5 实例方法和静态方法

和字段一样，方法也分为实例方法和静态方法，后者在定义时要加上static修饰符。例如，Console类的WriteLine和ReadLine等方法就是静态方法，它们不是通过某个Console

类的实例来调用,而是通过类本身来调用。

一般来说,如果一个方法不需要访问类中的实例字段和其他实例方法,那么就可以使用 static 关键字将其定义为静态方法。第 3.3.3 节中定义的 Swapper 类的各个成员方法就是这样的例子。

在一个类型中,实例方法的执行代码可以直接访问静态成员和非静态成员。但静态方法的代码只能直接访问静态成员,访问非静态成员时需要指明其所属的实例。例如,在下面的代码中,非静态的 Query 方法能够直接访问当前对象的 ID 和 Money 字段,而静态的 Query 方法则需要通过一个对象变量 *a* 才能访问。

```
class Account
{
    public static string Cur = "人民币";    //静态字段
    public string ID;
    public decimal Money = 100;

    public void Query()                      //实例方法可以访问实例字段和静态字段
    {
        Console.WriteLine("卡号{0} 余额{1}{2}", ID, Money, Cur);
    }

    public static void Query(Account a)      //静态方法不能直接访问实例字段
    {
        Console.WriteLine("卡号{0} 余额{1}{2}", a.ID, a.Money, Cur);
    }
}
```

如果一个类的成员都是静态成员,那么该类实际上不需要创建对象,此时可使用 static 修饰符将其定义为静态类。例如,Console 类就是一个静态类。

3.4 委托与方法调用

结构和类的对象都可以作为变量或参数进行传递,这为程序之间的信息交互奠定了基础。例如,下面的两段方法代码分别定义了两个数学函数。

```
static int f(int x)
{
    return 3 * x;
}

static int g(int x)
{
    return x * x;
}
```

要对某个数计算上述两个函数的复合函数值,只需将该数作为参数传递给第一个函数,再将其返回值作为参数传递给第二个函数。

```
int x = g(f(10));
```

有时候，希望将方法(或函数)也作为变量或参数进行传递。在数学上，如果函数的参数或返回值也是函数，那么称为高阶函数，它能够给计算带来巨大的灵活性。并不是所有程序设计语言都提供了这种功能，C 和 C++是通过指针来封装方法，而 C＃的实现方式就是委托。和指针相比，委托是面向对象且类型安全。委托的使用过程分为 3 步：

(1) 定义委托原型。委托的定义类似于方法的签名，不过前面需要加上一个关键字 delegate。例如，下面的代码定义了一个名为 MyDelegate 的委托。

```
delegate bool MyDelegate(decimal x);
```

(2) 创建委托对象。即将某个方法作为参数封装到委托对象的创建表达式中，此时要求方法的参数和返回类型都应与委托原型中的定义完全一致，否则就会出错。例如，下面的代码基于一个 BankCard 对象的 Pay 方法来创建了一个 MyDelegate 委托对象。

```
BankCard c1 = new BankCard();
MyDelegate d1 = new MyDelegate(c1.Pay);
```

上面最后一行代码还可以直接简写为：

```
MyDelegate d1 = c1.Pay;
```

(3) 通过委托对象来调用方法。例如：

```
d1(300);
```

下面的程序 P3_9 演示了基于委托的高阶函数计算。程序定义了两个一元函数 f 和 g，以及一个高阶函数 h，其中，h 的前两个参数类型为委托类型 Fun，那么就可以将不同的一元函数传递给 h，进而输出各种函数复合的计算结果。

```
//程序清单 P3_9.cs
using System;
namespace P3_9
{
    delegate int Fun(int x);

    class Program
    {
        static void Main()
        {
            h(f, g, 10);
        }

        static int f(int x)
        {
            return 3 * x;
        }

        static int g(int x)
        {
```

```
            return x * x;
        }

        static void h(Fun f, Fun g, int x)
        {
            Console.WriteLine("(f·f) {0} = {1}", x, f(f(x)));
            Console.WriteLine("(g·g) {0} = {1}", x, g(g(x)));
            Console.WriteLine("(f·g) {0} = {1}", x, f(g(x)));
            Console.WriteLine("(g·f) {0} = {1}", x, g(f(x)));
        }
    }
}
```

该程序的输出结果如下：

```
(f·f) 10 = 90
(g·g) 10 = 10000
(f·g) 10 = 300
(g·f) 10 = 900
```

最后指出一点，程序里定义的所有委托类型默认都是 System 程序集中 Delegate 类的派生类。

3.5 成员访问限制

信息封装是面向对象思想中的一个重要概念，即每个对象负责维护自己的数据信息，对象之间主要通过发送消息进行通信。外部对象要访问其数据信息或与之进行消息通信，就必须有足够的访问权限。C＃中的访问限制修饰符就用于设置类型及成员的访问权限。

在前面的一些程序示例中，可以看到很多类型的字段和方法的定义前都加上了 public 修饰符，它表示成员是公有的，可以从外部进行访问。与之相反，修饰符 private 表示成员是私有的，只允许在类型内部进行访问。例如，在下面定义的 Contact 类中，字段 Name 和 Phone 是公有的，而 gender 和 address 是私有的。

```
class Contact
{
    public string Name;
    public string Phone;
    private bool gender;
    private string address;

    public char GetGenderChar()
    {
        if (gender) return '男';
        else return '女';
    }
}
```

私有字段只允许其所属类型的其他方法成员进行访问。例如，GetGenderChar 方法中就访问了私有字段 gender。而下面 Main 方法中的最后一行代码是错误的，因为从 Program 类的方法中不允许访问 Contact 对象的私有字段。

```
class Program
{
    static void Main()
    {
        Contact c1 = new Contact();
        c1.Name = "王小明";
        c1.gender = true;                    //错误：不能访问 Contact 对象的私有字段
    }
}
```

对于类型中的方法而言，如果需要通过它对外提供服务，即通过对象可以调用该方法，那么它应当定义为公有的。私有方法的作用通常是封装一部分功能，以供类型中的其他方法调用。

提示：如果成员定义中没有指定访问限制修饰符，那么默认为私有的。

修饰符 protected 表示成员是保护的，只能被当前类及其派生类中的方法成员访问。在下面的程序示例中，Account 类的字段 id 属于保护成员，那么可在该类的成员方法 Query 和派生类 BankCard 的成员方法 Pay 中进行访问，但不能在 Program 类的 Main 方法中访问。由于 Account 和 BankCard 都没有提供修改该字段的操作，因此使用 new 关键字构造的 BankCard 对象和 CreditCard 对象的 id 字段值将一直为空。在静态方法 Account.Create 中可以创建 Account 对象并设置其 id 值，但创建之后在外部也没有办法修改该字段值。

```
//程序清单 P3_10.cs
using System;
namespace P3_10
{
    class Program
    {
        static void Main()
        {
            Account acc1 = new Account();
            acc1.Query();                    //输出"账号 余额 100 人民币"
            BankCard card1 = new BankCard() { Money = 200 };
            card1.Pay(150);                  //输出"账号 消费 150 余额 50 人民币"
            Account acc2 = Account.Create("001");
            acc2.Money = 1000;
            acc2.Query();                    //输出"账号 001 余额 1000 人民币"
        }
    }

    public class Account
    {
        public readonly string Cur = "人民币";
        protected string id;
        public decimal Money = 100;
```

```
        public void Query()
        {
            Console.WriteLine("账号{0} 余额{1}{2}", id, Money, Cur);
        }

        public static Account Create(string newid)
        {
            Account a = new Account();
            a.id = newid;
            return a;
        }
    }

    class BankCard : Account
    {
        public void Pay(decimal price)
        {
            Money = Money - price;
            Console.WriteLine("账号{0} 消费{1}余额{2}{3}", id, price, Money, Cur);
        }
    }
}
```

另外,还有一个 internal 修饰符,它表示可在程序集内部访问,即可以通过当前类型或同一程序集中的其他类型访问,而不允许其他程序集中的类型来访问。对于类型的成员来说,上述修饰符中只有 protected 和 internal 二者可以同时使用。此时除了当前类型之外,在当前程序集中定义的派生类可以访问这些内部保护成员;而在其他程序集中即使定义了当前类的派生类,也无法访问这些成员。

除了类型的成员之外,public 和 internal 还可用于修饰整个类型,前者表示类型可从外部程序集访问,后者则不允许。例如,程序 P3_10 中的 Account 类就声明为公有的,而 BankCard 类默认是内部的。

由以上介绍可知,public、protected、internal、private 这些修饰符的访问限制级别是依次由低到高的。对类型而言,派生类的访问限制级别不能低于基类的访问限制级别。这种要求是显而易见的,否则外部程序就无法通过派生类对象来调用其基类的成员,因此也就失去了继承的意义。例如,在程序 P3_10 中,不能将基类 Account 定义为内部的,而将派生类 BankCard 定义为公有的。

而对类型的成员而言,字段的访问限制级别不能低于其类型的访问级别,方法的访问限制级别也不能低于方法参数类型的访问限制级别,否则就会出现"能访问成员但不知道其类型"这种荒谬的情况。例如,下面的代码是错误的,因为 Address 结构是内部的,那么 Contact 类的字段 address 和方法 PrintAddress 就不能为公有的或保护的。

```
public class Contact
{
    public string Name;
    public string Phone;
```

```
    protected Address address;                  //错误：字段类型比字段的可访问性低

    public void PrintAddress(Address a)         //错误：参数类型比方法的可访问性低
    {
        Console.WriteLine("{0}市{1}街{2}号", a.City, a.Street, a.Number);
    }
}

internal struct Address
{
    public string City, Street;
    public int Number;
}
```

改正上面的代码有几种方式，一是将 Address 类型改为公有的，二是将 Contact 类型改为内部的，三是将 Contact 类的成员 address 和 PrintAddress 改为私有的。

此外，如果一个类型是嵌套定义在另一个类型中，那么内部类型的访问限制级别不能高于外部类型。

本章小结

类和结构中可以定义用于描述状态的字段成员以及用于行为实现的方法成员。方法可以接收参数，从而根据传入的参数值来执行相应的计算。委托能够以面向对象的方式来封装方法，从而提高方法调用的灵活性。

按照成员是属于类型的实例还是类型本身所有，还可以将其区分为实例（非静态）成员和静态成员。这些成员的访问权限可以通过访问限制修饰符进行控制。

本章介绍的大部分成员用法既适用于类，也适用于结构。对于一般方法而言，学习的重点在于掌握形参和实参的区别以及不同类型参数的使用方法。类还有一些特殊的方法成员，对它们的介绍将放在第 6 章。

习题 3

1. 在 C＃程序中定义变量名有什么规定？
2. 在 C＃程序中，方法的标识由哪几部分组成？
3. 简述局部变量和成员字段在用法上的相同点和不同点。
4. 常数字段和只读字段有何不同？
5. 方法的返回值和输出型参数有什么区别？它们各自适用于什么场合？
6. 编写程序来求解以下物理问题：(1)计算物体从指定高度落下后的速度；(2)计算两个物体间的万有引力。程序中需要定义和使用哪些常量？
7. 参照程序 P3_4，编写计算一组数中的最大值和最小值的程序，要求使用数组型参数。