第 5 章 函数、域与属性

主要内容

- 函数的定义、使用
- 属性与域的定义以及使用

C# 中函数的定义

- 在大多数的应用软件的设计中，将应用程序分成若干个功能单元。由于小段的程序更易于理解、设计、开发和调试，因此采用功能单元是应用程序设计的核心法则。分为若干个功能单元以便在应用程序中重用功能构件。另外，在整个大的程序中，某些任务常常要在一个程序中运行好多次，举个最简单的例子，对多个数组进行排序。此时我们就可以把这些相同的代码段写成一个单独的单元，需要的时候我们就来调用它。 C#.NET 程序语言中我们把这个单独的单元叫做函数

函数的特点

- 函数拥有自己的名称，可以使用合法的 C#.NET 标识符来命名。但其名称不能与变量、常数或定义在类内的属性或者其他方法名重复。
- 函数内声明的变量属于局部变量，也就是说 C#.NET 在不同函数内声明的变量彼此互不相关，其作用域局限在该函数内。所以在不同的函数内允许声明相同局部变量名称。
- 函数有特定功能，程序代码简单明确，可读性高而且容易调试和维护。

5.1 函数的定义和使用

- 函数就是代码的逻辑片断，它可以执行特定的操作。对象或者类调用函数来实现函数的功能。函数可以有返回类型，当然，返回类型也可以是 Void 。

函数声明的语法为：

< 修饰符 >< 返回类型 >< 函数名称 >(参数 1 ，参数 2 ，…)

函数的修饰符

- 函数的修饰符有很多，如：
  - new
  - public
  - protected
  - internal
  - private
  - static
  - virtual
  - sealed
  - override
  - abstract
  - extern

函数修饰符的含义

- 上面诸多函数修饰符中， Public 、 Protected 、 Internal 、 Protected Internal 、 Private 是对函数作用域的修饰，其余的关键字有其他的含义，在此，我们只讲函数作用于修饰符的意义，

函数修饰符的含义

函数访问修饰符	功能说明
Public	函数的访问权限完全没有限制
Protected	只有本类或者继承自本类的子类（即以本类作父类的类）可以使用
Internal	函数的使用仅限于当前项目
protected internal	函数的使用仅限于当前项目或者继承于此类的的类
Private	只有类本身可存取而已（默认）

一个简单的调用函数的例子

- 声明函数之后，我们就可以调用任何类或者对象所使用的函数。
- 例：第一个函数的定义以及调用

```
namespace ch05_1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(" 请输入用户名： ");
            string name = Console.ReadLine();
            Console.WriteLine(" 请输入密码 :");
            string pwd = Console.ReadLine();
            print(name, pwd);
        }
        private static void print(string str1, string str2)
        {
            Console.WriteLine(" 用户名： {0}， 密码： {1}",str1,str2);
        }
    }
}
```

程序运行结果


C:\Windows\system32\cmd.exe
请输入用户名：
张三
请输入密码：
123456
用户名：张三，密码：123456
请按任意键继续. . .

5.2 函数参数的传递方式

- 在调用函数的时候，可以向函数传递参数列表。 C# 中函数的参数有 4 种类型：
  - 值参数，不含任何修饰符。
  - 引用型函数，以 Ref 修饰符声明。
  - 输出参数，以 Out 修饰符声明。
  - 数组型参数，以 Params 修饰符声明。
- 若 A 语句中调用函数 B ，两者间有参数传递，那么，我们将 A 调用语句中传送的参数称为实参；被调用的函数 B 中使用的参数成为形参。

5.2.1 值参数

- 当利用值向函数传递参数时，编译程序给实参的值做一份复制，并且将此复制传递给该函数。被调用的函数不会修改内存中实参的值，所以使用值参数时，可以保证实际值是安全的。

值参数传递示例

- **例** 值参数传递示例。
- 程序代码：

```
public partial class Form1 : Form
  {
     public Form1()
     {
        InitializeComponent();
        txtPre.Text = "a=" + a + ",b=" + b;
     }
     int a = 2, b = 3;
     private void btnChange_Click(object sender, EventArgs e)
     {
        Swap(a, b);
        txtNext.Text = "a=" + a + ",b=" + b;
     }
     private static void  Swap(int n1,int n2)
     {
        int temp;
        temp = n1;
        n1 = n2;
        n2 = temp;
     }
  }
```

程序运行结果

- 此程序的输出结果是 a=2 ， b=3 ，可见我们并没有达到交换的目的，在这个程序里我们采用了值参数传递，形参值的修改并不影响实参的值。

5.2.2 引用型参数

- 与值参不同的是，引用型参数并不开辟新的内存区域。当利用引用型参数向函数传递形参时，编译程序将把实际值在内存中的地址传递给函数。
- 在函数中，引用型参数通常已经初始化。

引用型参数传递例子

- **例** 把上面例子改写成引用型参数传递。
  程序代码：

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            txtPre.Text = "a=" + a + ",b=" + b;
        }
        int a = 2, b = 3;
        private void btnChange_Click(object sender, EventArgs e)
        {
            Swap(ref a, ref b);
            txtNext.Text = "a=" + a + ",b=" + b;
        }
        private static void Swap(ref int n1,ref int n2)
        {
            int temp;
            temp = n1;
            n1 = n2;
            n2 = temp;
        }
    }
```

运行结果及分析

- 此程序的输出结果是 a=3， b=2 。在鼠标单击事件中，调用了 Swap 函数，使用引用型参数，成功地实现了 a 和 b 的交换。n1 和 n2 所处的内存区域其实就是 a 和 b 所处的内存区域，所以当 n1 和 n2 的值互换时， a 和 b 的值自然会发生变化。

5.2.3 输出型参数

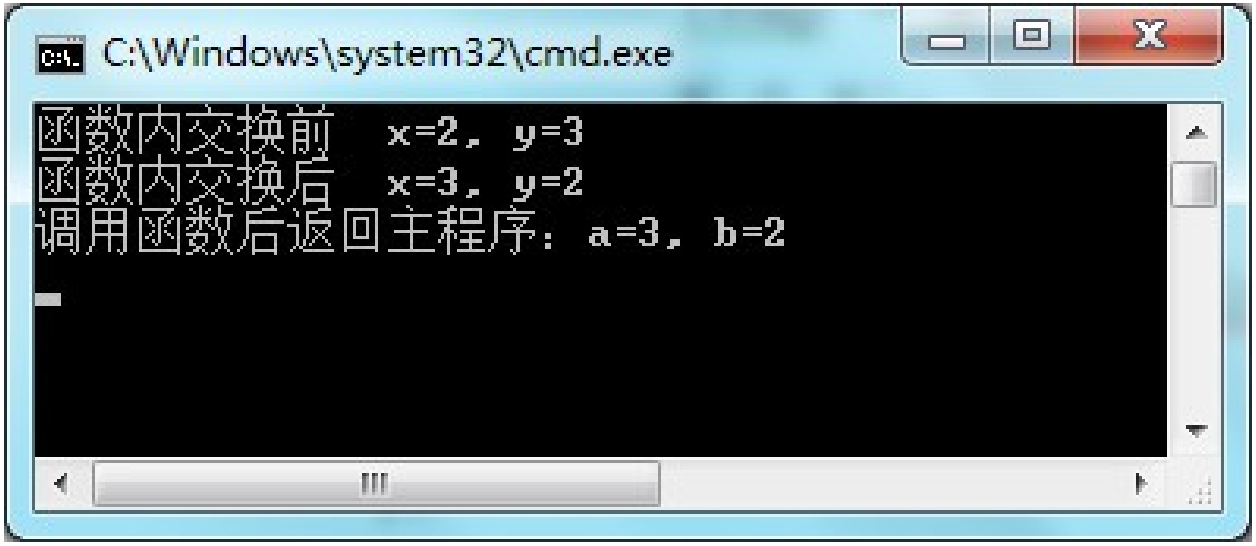
- 与引用型参数类似，输出型参数也不开辟新的内存区域。与引用型参数的差别在于，调用前不需对变量进行初始化。输出型参数用于传递方法返回的数据。
- Out 修饰符后应跟随与形参的类型相同的类型声明。在方法返回后，传递的变量被认为经过了初始化。

Out 关键字使用小例子

- 例 使用 Out 关键字练习编写输出参数。

```
class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            UseOut(out a, out b);
            Console.WriteLine(" 调用函数后返回主程序： a={0}, b={1}", a, b);
            Console.ReadLine();
        }
        private static void UseOut(out int x, out int y)
        {
            int temp;
            x = 2;
            y = 3;
            Console.WriteLine(" 函数内交换前    x={0}, y={1}", x, y);
            temp = x;
            x = y;
            y = temp;
            Console.WriteLine(" 函数内交换后    x={0}, y={1}", x, y);
        }
    }
```

运行结果


C:\Windows\system32\cmd.exe
函数内交换前 x=2, y=3
函数内交换后 x=3, y=2
调用函数后返回主程序: a=3, b=2

5.2.4 数组型参数

- C# 允许为函数指定一个（只能指定一个）特定的函数，这个参数必须是函数定义中的最后一个参数，成为数组型参数。数组型参数可以使用个数不定的参数调用函数，它可以使用 params 关键字来定义。另外，参数只允许是一维数组。比如 int[] 和 int[][] 类型都可以作为数组型参数，而 int[,] 则不可以。最后，数组型参数不能再有 Ref 和 Out 修饰符。

数组型参数示例

- 例数组型参数示例

```
class Program
    {
        static void Main(string[] args)
        {
            int[] a=new int[6];
            for (int i = 0; i < 6; i++)
            { a[i] = i + 1;
            Console.Write(a[i] + " ");
            }
            double dl = Age(a);
            Console.WriteLine("\n 平均值为： {0}",dl);
        }
        static double Age(params int[] b)
        {
            int sum = 0;
            for (int i = 0; i < b.Length; i++)
            {
                sum = sum + b[i];
            }
            return (sum * 1.0) / b.Length;
        }
    }
```

程序运行结果

5.2.5 参数的匹配

- 在调用函数时，实参和形参必须完全匹配，这意味着形参与实参之间类型、个数和顺序都要完全匹配。例如下面的函数：

```
private  void f(int a,string  b)
    {
       Console.WriteLine("dsahfsd;hnf");
    }
```

- 则不能使用下面的代码调用；

```
f(1,2);
```

- 这是因为，函数的形参第一个为整型，第二个为 string 型。而调用函数的代码中第一个实参为整型，第二个还是整型，与函数的第二个形参不匹配。

  同样，上面的函数也不能被以下代码调用：

  ```
  f("there");
  ```

  这里的实参形参个数明显不一样，所以不可以。

  参数的不匹配通不过编译，因为编译器要求必须匹配函数的签名。

5.3 区块变量与属性成员

5.3.1 区块变量

- C#.NET 语言中，区块变量被定义于某个区块中，比如说前面所讲的 While 循环语句中声明的变量，只能在所定义的 While 循环中使用。也就是说，某区块中定义的变量，只能供这个区块使用，区块以外使用则报错。

5.3.2 属性成员

- 属性成员跟在第二章中介绍的静态变量一样，不同的是对象属性是放在堆里面的，必须对对象进行实例化，才可以使用属性；而静态字段是放在全局变量区的，不需要实例化对象就直接可以引用静态变量。

```
class test
{
  public static int value;       // 静态变量
   public int value1;            // 对象属性
}
```

5.4 运算符重载

- 运算符重载十分有用，因为我们可以在运算符重载中执行所需要的任何操作。在前面，我们讲过了算术操作符，但是我们知道，在 C# 中，所有的数据要么属于某个类，要么属于某个类的实例，充分体现了面向对象的思想。因此为了表达的方便，人们希望可以重新给已定义的操作符赋予新的含义，在特定的类的实例上进行新的解释。这就需要通过运算符重载来解决。

可以重载的运算符

- 可以重载的运算符有：
  - 一元运算符：＋、－、！、～、 ++、－－、 true、 false
  - 二元运算符：＋、－、 *、 /、 &、 |、 ^、 <<、 >>
  - 比较运算符： ==、！ =、 <、 >、 <=、 >=

5.4.1 一元运算符重载

- 顾名思义，一元运算符重载时操作符只作用于一个对象，此时参数表为空，当前对象作为操作符的单操作数。
- 举一个极为常见的例子：比如说在某游戏中，若某个兵营遭到抢劫，那么钱物、武器、战斗力、兵营面积都会变小；若对别的兵营进行掠夺，则钱物、武器、战斗力、兵营面积都会变多。

二元运算符重载

- 上面讲述的一元运算符重载我们平时使用的不多，二元运算符我们使用的特别多。使用二元运算符时，参数表中有一个参数，当前对象作为该操作符的左操作数，参数作为操作符的右操作数。

5.4.3 比较运算符重载

- 比较运算符的重载比较常见，但是要注意的是，在重载比较运算符的过程中，有些必须成对重载，比如，在重载运算符“ <”的同时，必须也对运算符“ >”进行重载。下面的代码就是对运算符“ ==”和运算符“ !=”运算符的成对重载。

```
class Class1
{
  public int x;
  // 重载运算符
  public static bool operator==(Class1 c1,Class1 c2)
   {
     return (c1.x==c2.x);
   }
  public static bool operator!=(Class1 c1,Class1 c2)
    {
      return !(c1 ==c2);
    }
}
```

5.5 Main() 函数

- 所有的 C# 应用程序必须在它的一个类中定义一个名为 Main 的函数。这个函数作为应用程序的入口点，它必须被定义为静态的。具体在哪个类中使用 Main() 函数对 C# 编译器并无影响，而且你选择的类也不影响编译的次序。这与 C++ 不同，在 C++ 中编译应用程序时必须密切注意依赖性。 C# 编译器很精明，可以自己在源代码文件中自动搜寻到 Main （）函数。因此，这个最重要的方法是所有 C# 应用程序的入口点。

- 虽然一个 C# 应用中可能会有很多类，但是其中只有一个入口。在同一个应用中，可能多个类都有 Main() 函数，但是只有一个 Main （）函数是被执行的。你需要在编译的时候指定究竟使用哪一个 Main() 函数。
- 常见的 Main() 函数是这样的：

```
static void Main(string[] args)
{  }
```

- Main() 函数中的参数 Args 是从应用程序的外部接受信息的方法，这些信息在运行期间指定，其形式是命令行参数。

- 细心的读者会注意到， Main() 函数必须定义为静态的，这是因为 C#.NET 是一门真正的面向对象的编程语言， Main() 函数是整个应用程序的入口， Static 可以保证程序调用的时候不需要实例化就可以运行程序。

  看看下面的一段代码：

```
namespace test
{
    class Test
    {
        public void InstanceMethod() { }    //   实例成员（非静态）
        public static void StaticMethod { }    //   类型成员（静态）
        static void Main(string[] args)
        {
            InstanceMethod();    //   错误！调用了实例成员，而此时并没有建立实例
            StaticMethod();      //   正确！可以调用静态成员
            Test SomeTest = new Test();    //   建立本类型的一个实例
            SomeTest.InstanceMethod();      // 再在这个实例上调用实例成员就对了
            SomeTest.StaticMethod();   // 附加一句，在实例上调用静态成员也是错误的！
        }
}
```

  上面的代码中注释很明确，在这里就不多解释了。

5.6 字段

- 字段声明方法：

  field-modifiers type variable-declarators ;

  其中 field-modifiers 表示字段的修饰符，type 表示字段的具体类型， variable-declarators 表示字段的变量名。

- 字段的修饰符 field-modifiers 可以是：
  - new
  - public
  - protected
  - internal
  - private
  - static
  - readonly

5.7 属性

- 属性的定义跟域有些相似，但是内容要比域的内容多。属性是对现实世界中实体特征的抽象，它提供了对类或对象性质的访问。比如，一个用户的姓名、一个文件的大小、一件物品的重量都可以作为属性。类的属性所描述的是状态信息，在类的某个实例中属性的值表示该对象的状态值。

属性的修饰符

- 属性的修饰符 property-modifiers 有：
  - new
  - public
  - protected
  - internal
  - private
  - static
  - virtual
  - sealed
  - override
  - abstract

属性定义举例

- 下面的代码简单地介绍了属性 myProperties 的定义。

```
class A
{
   private int i;
  public int myProperties
  {
          get
          {
             return i;
          }
          set
         {
            i=value;
         }
   }
}
```

Get 与 set 语句块的解释

- Get 语句块是用于读取属性值的方法，其中没有任何参数，但是返回属性声明语句中所定义的数据类型值。在 Get 语句块中，包含 Return 或者 Throw 语句，这个可以有效地防止执行控制权超出 Get 语句块。简单的属性一般与一个私有域相关联，以控制对这个域的访问，此时 Get 语句块可以直接返回该域的值。
- Set 语句块以类似的方式把一个值赋给域，可以使用关键字 Value 引用用户提供的属性值。比如上面定义的属性，我们要把整数 10 赋值给 myProperties 属性，可以使用下面的语句：

- 在声明属性语句中，可以对属性进行分类：
  - 只读属性：属性定义中只有 Get 语句，表明属性的值只能进行读出而不能设置。
  - 只写属性：属性定义中只有 Set 语句，表明属性的值只能进行设置而不能读出。
  - 读写属性：属性定义中既有 Get 语句又有 Set 语句，表明属性的值既能进行读出又能进行设置。

结论

- 本章我们主要介绍了函数的定义、使用以及几种参数传递的不同和注意事项。后面我们还介绍了属性与域的概念以及使用方法。
- 函数是各种程序语言中极其重要的一部分，在 Visual C# 中也不例外，函数拥有多种特性，如继承、委托等，这些我们将在后面的章节予以介绍。本章还详细讨论了函数中各种参数传递的方式以及函数的返回值。
- 之后，我们在本章讨论了字段与属性这 2 个概念，它们都是用来保存类的实例的各种数据信息。其中，属性实现了良好的数据封装和数据隐藏；字段可以分为静态字段和非静态字段。

谢谢！