第3章 C#.NET语法进阶

本章将详细讲解 C#.NET 的语法进阶知识，包括数组知识及应用、函数的定义与调用等。每个知识点都配套经典示例以便学生理解相关理论和掌握知识点的应用。还介绍 C# 编程过程中的一些编码规范，包括源代码书写规则和标识符命名规则。

【本章要点】

- 数组的创建与使用
- 函数的定义与调用
- 程序调试与异常处理
- C# 编码规范

3.1 数组

3.1.1 一维数组的创建和使用

在 C#.NET 程序设计中，数组与字符串一样，是最常用的引用类型之一，数组能够按一定的规律把相关的数据组织在一起，并能通过“索引”或“下标”快速地管理这些数据。

标量变量只能存储单个数据；数组是一组相同类型数据连续存储的集合，这一组数据在内存中的存储空间是相邻接的，每个存储空间都存储了多个数组元素。通过数组可以对性质相同的数据进行存储和管理。数组与变量的对比效果如图 3.1 所示。

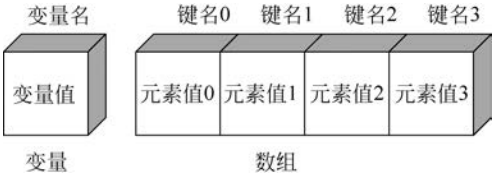

图 3.1 数组与变量的对比

数组由数组元素组成，每个元素都包含一个“键”(Key)和一个“值”(Value)，可以通过键名来访问相应的数组元素值，数组元素的“键”由整数组成，数组元素的“值”可以是任何数据类型，包括数组或对象。如果数组元素的值是另外一个数组，那么这个数组就是二维数组，因此，数组又可以分为一维数组、二维数组和多维数组。

一维数组即数组的维数为 1，它相当于一组小抽屉，抽屉格子的数量就是一维数组元素的个数。

1. 一维数组的创建

数组的索引由数字组成，默认从 0 开始自增，每个索引号都对应数组元素在数组中的位置。创建一维数组的语法格式如下：

格式 1：

```
元素类型[] 数组名称;
```

格式 2：

```
元素类型[] 数组名称 = new 元素类型[元素个数]{"元素值 1","元素值 2",… };
```

例如：

```
int[] a = new int[3]{8,10,12};
```

2. 一维数组的使用

一维数组的使用包括数组元素的赋值和取值，对一维数组的赋值比较简单，通过索引号找到相应数组元素，然后对该数组元素进行赋值和取值。一维数组的使用示例源代码如下：

```
int[] a = new int[3];                    //定义数组
a[0] = 8;                                //为数组元素赋值
a[1] = 10;
a[2] = 12;
```

对一维数组的操作可以通过索引号对指定数组元素进行访问，也可以采用 for 循环语句或 foreach 循环语句对数组进行遍历访问。

【例 3.1】 本例学习一维数组的创建及使用，首先创建一个长度为 4 的字符串类型数组并赋初始化值，接着使用 for 循环语句遍历输出该数组的各个元素，然后访问并修改各数组元素的值，最后再次使用 for 循环语句遍历输出数组各元素值。

创建一个 C＃控制台应用程序，命名为 case0301。在 Program 类的 Main()函数中编写如下源代码：

```
string[] a = new string[4]{"春天", "夏天", "秋天","冬天"};     //创建一维数组并赋初始化值

//采用 for 循环语句遍历数组元素
Console.WriteLine("中文的一年四季:");
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine(a[i]);
}

a[0] = "Spring";                                                //对数组元素重新赋值
a[1] = "Summer";
a[2] = "Autumn";
a[3] = "Winter";

Console.WriteLine("Seasons in English:");
for (int i = 0; i < a.Length; i++)
```

```
{
    Console.WriteLine(a[i]);
}
Console.Read();
```

保存项目文件并运行程序,程序运行效果如图 3.2 所示。

图 3.2　一维数组的创建和使用

3. 一维数组的遍历

使用 for 和 foreach 语句可以实现数组的遍历操作,可以访问数组中的每个元素。for 循环语句的语法格式如下:

```
for (int i = 0; i < 数组名称.数组长度; i++)
{
    Console.WriteLine(数组名称[i]);
}
```

例如:

```
int[] a = new int[3] { 8,88,888 };
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine(a[i]);
}
```

foreach 循环语句的语法格式如下:

```
foreach (元素类型 元素对象 in 数组名称)
{
    Console.WriteLine(元素对象);
}
```

例如:

```
string[] b = new string[3] {"专科","本科","研究生" };
foreach (string c in b)
{
    Console.WriteLine(c);
}
```

【例 3.2】 本例学习一维数组的创建及使用，首先创建一个长度为 3 的整型数组并赋初始化值，然后对数组第二个元素进行访问并重新赋值，最后分别使用 for 循环语句和 foreach 循环语句遍历输出该数组的各个元素。

创建一个 C＃控制台应用程序，命名为 case0302。在 Program 类的 Main()函数中编写如下源代码：

```
int[] a = new int[3] { 10,20,30 };                    //创建一维数组并赋初始化值
a[1] = a[1] + 1;                                      //对数组第二个元素进行访问并重新赋值

Console.WriteLine("for 循环输出:");
//采用 for 循环语句遍历数组元素
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine(a[i]);
}
Console.WriteLine("foreach 循环输出:");
//采用 foreach 循环语句遍历数组元素
foreach (int b in a)
{
    Console.WriteLine(b);
}
Console.Read();
```

保存项目文件并运行程序，程序运行效果如图 3.3 所示。

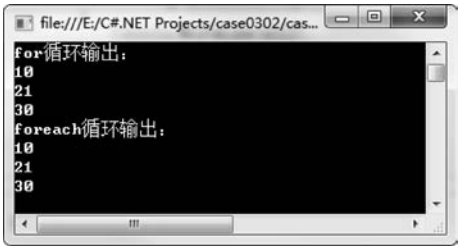

图 3.3 数组的遍历

【例 3.3】 本例学习使用冒泡排序法对数组元素进行排序。冒泡排序的过程是：假设数组长度为 n，以从大到小排序为例，先将第一个元素值与第二个元素值比较，如果第一个元素值小于第二个元素值，则将两者值对调。以此类推，直到完成最后一个元素值的比较，至此完成一趟冒泡排序，此时最后一个元素值已经是本数组中最小的元素值了。然后执行 $n-1$ 趟冒泡排序，就可以实现将整个数组元素值进行降序排列了。

创建一个 C＃控制台应用程序，命名为 case0303。在 Program 类的 Main()函数中编写如下源代码：

```
int[] arr = new int[10] {0, 6, 12, 18, 24, 30, 46, 42, 48, 54 };     //定义整型数组并赋初始值
Console.WriteLine("初始数组:");
foreach (int a in arr)
    Console.Write("{0} ", a);                                        //遍历排序前数组
```

```
Console.WriteLine("\n开始排序...");
//冒泡排序(降序)
for (int i = 0; i < arr.Length - 1; i++)
{
    for (int k = 0; k < arr.Length - i - 1; k++)
    {
        if (arr[k] < arr[k + 1])
        {
            int temp = arr[k];              //两者值对调
            arr[k] = arr[k + 1];
            arr[k + 1] = temp;
        }
    }
}

foreach (int a in arr)
    Console.Write("{0} ", a);               //遍历排序后的数组

Console.Read();
```

保存项目文件并运行程序,运行效果如图 3.4 所示。

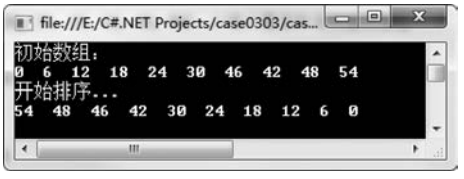

图 3.4　数组元素的冒泡排序

读者可以尝试使用冒泡排序法实现对数组元素的升序排序,将升序排序算法与降序排序算法进行对比,总结两种算法的区别与联系。

3.1.2　二维数组的创建和使用

二维数组即数组的维数是 2,二维数组本质上是一种以一维数组作为数组元素的数组。二维数组相当于一个表格,表格由行和列组成,每一行相当于一个一维数组。二维数组又称为矩阵,行列数相等的矩阵称为方阵。

1. 二维数组的创建

创建二维数组的语法格式如下。

格式 1:

```
元素类型[,] 数组名称;
```

格式 2:

```
元素类型[,] 数组名称 = new 元素类型[行数,列数]{{"元素值 1","元素值 2",…},{…} };
```

例如：

```
int[,] a = new int[2,2];
int[,] a = new int[2,2]{{11,12},{21,22}};
```

2. 二维数组的使用

二维数组的使用与一维数组相同，也包括数组元素的赋值和取值。

【例 3.4】 本例学习二维数组的创建及使用，首先创建一个 3 行 2 列的字符串类型二维数组并赋初始化值，接着尝试访问第 1 行第 2 列数组元素并为之重新赋值，最后使用嵌套 for 循环语句遍历输出该数组的各个元素值。

创建一个 C# 控制台应用程序，命名为 case0304。在 Program 类的 Main()函数中编写如下源代码：

```
string[,] booklist = new string[3, 2] { { "1", "C++项目开发" }, { "2", "PHP 项目开发" },
{ "3", "Java 项目开发" } };

booklist[0,1] = "C#.NET 项目开发";                    //访问数组元素并重新赋值

for (int i = 0; i < booklist.GetLength(0); i++)       //获取二维数组第一维的长度，即行数
{
    for (int j = 0; j < booklist.GetLength(1); j++)   //获取二维数组第二维的长度，即列数
    {
        Console.Write("{0} ", booklist[i, j]);
    }
    Console.WriteLine();
}
Console.Read();
```

保存项目文件并运行程序，程序运行效果如图 3.5 所示。

图 3.5 二维数组的创建和使用

【知识延伸】

booklist.GetLength(0) 返回第一维的长度(即行数)。

booklist.GetLength(1) 返回第二维的长度(即列数)。

3.1.3 ArrayList 对象的创建和使用

ArrayList 类相应于一种高级的动态数组，它是 Array 类的升级版本。ArrayList 类位于 System.Collections 命名空间下，它可以动态地添加、修改和删除元素，在使用 ArrayList 类时需要先引用命名空间 System.Collections。可以将 ArrayList 类看作扩充了功能的数组，但它并不等同于数组。与数组相比，ArrayList 类为开发人员提供了以下功能。

(1) 数组的容量是固定的,而 ArrayList 的容量可以根据需要动态扩充。

(2) ArrayList 提供添加、删除和插入某一范围元素的方法,但在数组中,只能一次获取或设置一个元素的值。

(3) ArrayList 提供将只读和固定大小返回到集合的方法,而数组不提供。

(4) ArrayList 只能是一维形式,而数组可以是多维的。

1. 定义 ArrayList 对象

ArrayList 的声明方式有三种,具体如下。

方式一:

```
ArrayList 对象名 = new ArrayList();                    //以默认(16)的大小来初始化内部的数组
```

方式二:

```
ArrayList 对象名 = new ArrayList(现有数组名称);    //将该集合的元素添加到 ArrayList 中
```

方式三:

```
ArrayList list = new ArrayList(int 整型数据);      //用指定的大小初始化内部的数组
```

【例 3.5】 本例学习 ArrayList 对象的创建及使用,首先分别使用三种声明方式创建 ArrayList 对象并赋值,最后使用 foreach 循环语句遍历输出各个对象值。

创建一个 C # 控制台应用程序,命名为 case0305。由于 ArrayList 类位于 System.Collections 命名空间下,因此需要在 Program 类文件中先引用命名空间 System.Collections,接着编写如下源代码:

```
//增加
using System.Collections;

namespace case0305
{
    class Program
    {
        static void Main(string[] args)
        {
            //方式一
            string[] studentlist = new string[3] { "张三", "李四", "王五" };
            ArrayList arr = new ArrayList(studentlist);

            //采用 foreach 循环语句遍历集合元素
            Console.WriteLine("集合 1");
            foreach (string stu in arr)
            {
                Console.WriteLine(stu);
            }

            ArrayList arr2 = new ArrayList();
```

```
            arr2.Add(100);
            arr2.Add(200);
            arr2.Add(300);
            Console.WriteLine("集合 2");
            foreach (int number in arr2)
            {
                Console.WriteLine(number);
            }
            Console.Read();
        }
    }
}
```

保存项目文件并运行程序,程序运行效果如图 3.6 所示。

图 3.6 ArrayList 对象的创建

2. 向 ArrayList 添加元素

向 ArrayList 集合中添加元素时,可以使用 ArrayList 类提供的 Add()方法和 Insert()方法。Add()方法用于将对象添加到 ArrayList 集合的结尾处,Insert()方法用于将元素插入 ArrayList 集合的指定索引处,语法格式如下。

定义:

```
ArrayList  arr = new ArrayList();
```

Add()方法:

```
arr.Add(Object 元素);
```

Insert()方法:

```
arr.Insert (int 插入位置索引号, Object 元素);
```

3. 删除 ArrayList 的元素

在 ArrayList 集合中删除元素时,可以使用 ArrayList 类提供的 Clear()方法、Remove()方法、RemoveAt()方法和 RemoveRange()方法。其中,Clear()方法用于从 ArrayList 中移除所有元素,Remove()方法用于从 ArrayList 中移除特定对象的第一个匹配项,RemoveAt()方法用于移除 ArrayList 的指定索引处的元素(索引号从 0 开始),RemoveRange()方法用于

移除 ArrayList 列表中从指定索引处开始的若干元素。各个方法的具体语法格式如下。

定义：

```
ArrayList  arr = new ArrayList();
```

Clear()方法：

```
arr.Clear();
```

Remove()方法：

```
arr.Remove(Object 元素);
```

RemoveAt()方法：

```
arr. RemoveAt(int 删除位置索引号);
```

RemoveRange()方法：

```
arr. RemoveRange(int 起始索引号, int 删除元素个数);
```

4. ArrayList 的遍历

ArrayList 集合的遍历与数组的遍历相似，可以使用 foreach 语句。

【例 3.6】 本例学习 ArrayList 对象元素的添加、使用及删除。由于 ArrayList 类位于 System.Collections 命名空间下，因此需要先引用命名空间 System.Collections，接着创建一个 ArrayList 对象，分别使用 Add()方法和 Insert()方法为该对象添加元素，然后分别使用 Remove()、RemoveAt()方法从该对象中删除指定元素，最后使用 foreach 循环语句遍历输出各个元素值。

创建一个 C# 控制台应用程序，命名为 case0306。在 Program 类文件中先引用命名空间 System.Collections，接着编写如下源代码：

```
//新增加
using System.Collections;

namespace case0306
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arr = new ArrayList();
            arr.Add("第 1 章");
            arr.Add("第 2 章");
            arr.Add("第 3 章");
            arr.Insert(0,"前言");                    //在第 1 个元素前插入值
```

```
            foreach (string a in arr)
            {
                Console.WriteLine(a);
            }
            Console.WriteLine("开始删除...");
            arr.Remove("第 2 章");                    //删除值为"第 2 章"的元素
            arr.RemoveAt(1);                          //删除索引号为 1 的元素,即第 2 个元素
            foreach (string a in arr)
            {
                Console.WriteLine(a);
            }
            Console.Read();
        }
    }
}
```

保存项目文件并运行程序,程序运行效果如图 3.7 所示。

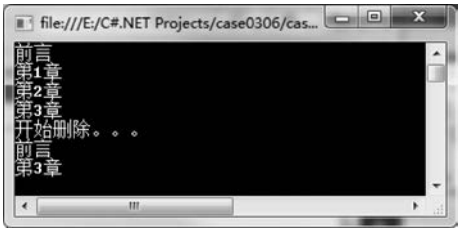

图 3.7 ArrayList 元素操作

【例 3.7】 本例学习使用 ArrayList 存储图书信息。创建一个 C# 控制台应用程序,命名为 case0307。在 Program 类文件中先引用命名空间 System.Collections,接着编写如下源代码:

```
//新增加
using System.Collections;

namespace case0307
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arr = new ArrayList();
            string[] book1 = new string[4] { "1", "XML 程序设计", "32 元", "2011 - 10 - 1" };
            string[] book2 = new string[4] { "2", "C# 程序设计", "59 元", "2014 - 03 - 1" };
            string[] book3 = new string[4] { "3", "PHP 程序设计", "53 元", "2013 - 07 - 1" };

            arr.Add(book1);
            arr.Add(book2);
            arr.Add(book3);
```

```
            Console.WriteLine("序号\t 书名\t\t 价格\t 出版日期");
            foreach (string[] b in arr)
            {
                Console.Write("{0}\t", b[0]);
                Console.Write("{0}\t", b[1]);
                Console.Write("{0}\t", b[2]);
                Console.Write("{0}\n", b[3]);
            }
            Console.Read();
        }
    }
}
```

保存项目文件并运行程序,程序运行效果如图 3.8 所示。

图 3.8 使用 ArrayList 存储图书信息

读者在本例的基础上,可以尝试分别使用 Add()方法和 Insert()方法为该对象添加元素,然后分别使用 Remove()、RemoveAt()方法从该对象中删除指定元素,最后使用 foreach 循环语句遍历输出各个元素值。

3.2 函数

在程序开发过程中,经常要重复某些操作或处理,如果每次都重复编写相同功能的源代码,不仅造成工作量加大,还会使程序源代码产生冗余、可读性差,项目后期的维护及运行效率也受到影响,因此引入函数的概念。所谓函数,就是将一些重复使用到的功能写在一个独立的源代码块中,在需要时单独调用。

3.2.1 函数的定义和调用

1. 函数的定义

C# 函数分为系统内置函数和用户自定义函数两种。定义函数的语法格式如下:

```
访问修饰符 返回值类型 函数名(参数类型 参数名 1,…)
{
    函数体;
    [return 返回值;]
 }
```

C# 中的函数命名应遵循以下规则:

➢ 函数名不能与内部函数名、C# 关键字重名;

- 函数名区分大小写，但建议按照大小写规范进行命名和调用；
- 函数名只能以字母开头，不能由下画线和数字开头，不能使用点号和中文字符。

2. 函数的调用

函数的调用可以在函数定义之前或之后，调用函数的语法格式如下：

```
函数名(实际参数列表);
```

【例 3.8】 本例学习函数的定义和调用，首先定义函数 GetSum()，其功能是计算传入的两个参数的和并输出结果。然后在 Main()主函数中调用 GetSum()函数，输出结果到控制台。

创建一个 C＃控制台应用程序，命名为 case0308。本章实例都保存在 E:\C＃.NET Projects 目录下。在 Program 类的 Main()函数中编写如下源代码：

```
static void Main(string[] args)
{
    int i = 80, j = 90;
    int k = GetSum(i, j);                    //调用函数
    Console.WriteLine("{0}和{1}的和是{2}", i, j, k);
    Console.Read();
}

static int GetSum(int a, int b)              //定义函数
{
    int c = a + b;                           //计算并返回结果
    return c;
}
```

保存项目文件并运行程序，程序运行效果如图 3.9 所示。

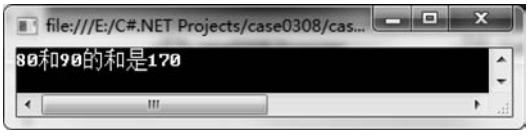

图 3.9 函数的定义和调用

3.2.2 参数传递

函数的使用经常需要用到参数，参数可以将数据传递给函数。在调用函数时需要填写与函数形式参数个数和类型相同的实际参数，实现数据从实际参数到形式参数的传递。参数传递方式有值传递、引用传递两种。

1. 值传递

值传递是指将实参的值复制到对应的形参中，然后使用形参在被调用函数内部进行运行，运算的结果不会影响到实参，即函数调用结束后，实参的值不会发生改变。值传递的语法格式如下：

```
//1 定义函数
访问修饰符 返回值类型 函数名(参数类型 参数名 1,…)
//2 调用函数时
函数名(实际参数 1,…);
```

【例 3.9】 本例实现函数参数的值传递调用,观察函数调用是否对实际参数造成影响。创建一个 C# 控制台应用程序,命名为 case0309。在 Program 类的 Main()函数中编写如下源代码:

```
static void Main(string[] args)
{
    int a = 20;
    Console.WriteLine("调用函数之前,函数外 a 的值:{0}", a);      //函数调用前

    example(a);          //调用函数,a 为实参,调用函数,此处传递的是值

    Console.WriteLine("调用函数之后,函数外 a 的值:{0}", a);   //函数调用后,实参值不变,a = 20
    Console.Read();
}

static void example(int a)                                          //定义函数,a 为形参
{
    a = a * a;
    Console.WriteLine("自定义函数内 a 的值:{0}", a);              //输出形参的值
}
```

保存项目文件并运行程序,程序运行效果如图 3.10 所示。

图 3.10 值传递的应用

2. 引用传递

引用传递也称为按地址传递,就是将实际参数的内存地址传递到形式参数中。此时被调用函数内形式参数的值发生改变时,实际参数也发生相应改变。引用传递的语法格式如下:

```
//1 定义函数时,在形式参数前面加上 ref 符号
访问修饰符 返回值类型 函数名(ref 参数类型 参数名 1,…)
//2 调用函数时,在实际参数前面加上 ref 符号
函数名(ref 实际参数 1,…);
```

【例 3.10】 本例实现函数参数的引用传递调用,与例 3.9 相似,仅在定义函数的形式参数前面和调用函数时实际参数前面多了一个 ref 符号。注意观察比较函数调用是否对实际参数造成影响。

创建一个C#控制台应用程序，命名为case0310。在Program类的Main()函数中编写如下源代码：

```
static void Main(string[] args)
{
    int a = 20;
    Console.WriteLine("调用函数之前,函数外a的值:{0}", a);      //函数调用前

    example(ref a);        //调用函数,a为实参,此处传递的是地址

    Console.WriteLine("调用函数之后,函数外a的值:{0}", a);        //函数调用后,实参值改变了
    Console.Read();
}

static void example(ref int a) //定义函数,a为形参
{
    a = a * a;
    Console.WriteLine("自定义函数内a的值:{0}", a);              //输出形参的值
}
```

保存项目文件并运行程序，程序运行效果如图3.11所示。

图3.11 引用传递的应用

3.3 程序调试与跟踪

程序调试是指对编写程序进行跟踪，以检查源代码正确性的过程。在开发过程中，程序调试是检查源代码并验证它能够正常运行的有效方法。在开发时，如果发现程序不能正常工作，就必须找出并解决有关问题。程序调试的基本操作包括断点设置、程序调试和变量跟踪。

3.3.1 程序调试

断点是一个信号，它通知调试器在某个特定点上暂时将程序执行挂起。当程序执行到某个断点处时，程序将挂起，称为程序处于中断模式。中断模式并不会终止或结束程序的执行。可以手动选择逐条语句继续执行、逐语句块继续执行或全部继续执行。一个程序中可以设置多个断点。

【例3.11】 以例3.10中的程序为例，演示如何在程序中设置断点，并且对程序运行过程进行跟踪，查看各个变量的值的变化。

(1) 断点设置。

在程序源代码窗口中,找到需要开始跟踪调试的源代码位置,在要设置断点的源代码行左边的灰色空白处单击。此时就会出现红色圆点,代表断点设置成功,如图 3.12 所示。

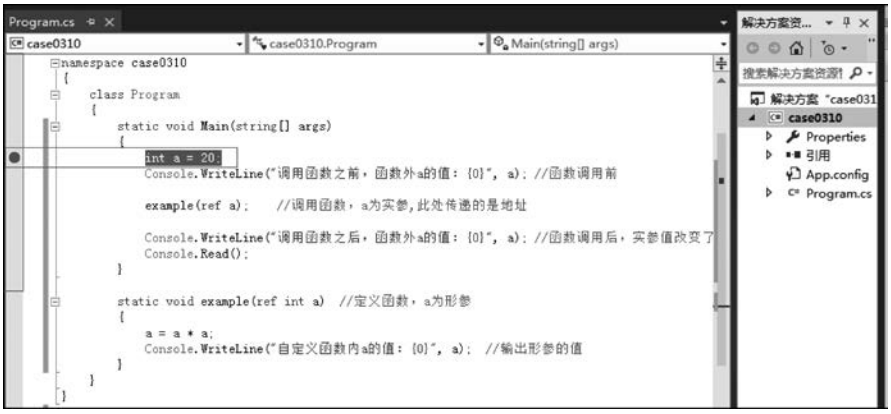

图 3.12　断点设置

也可以先选中某行源代码并右击,在弹出的快捷菜单中选择“断点”→“插入断点”命令即可为该行源代码设置断点。想要删除断点时,再次单击源代码行左侧的红色圆点即可。

(2) 程序调试。

断点设置完成后,运行程序,当程序运行到断点处时,程序自动暂停。此时代表断点的红色圆点就会变为黄色箭头,如图 3.13 所示。

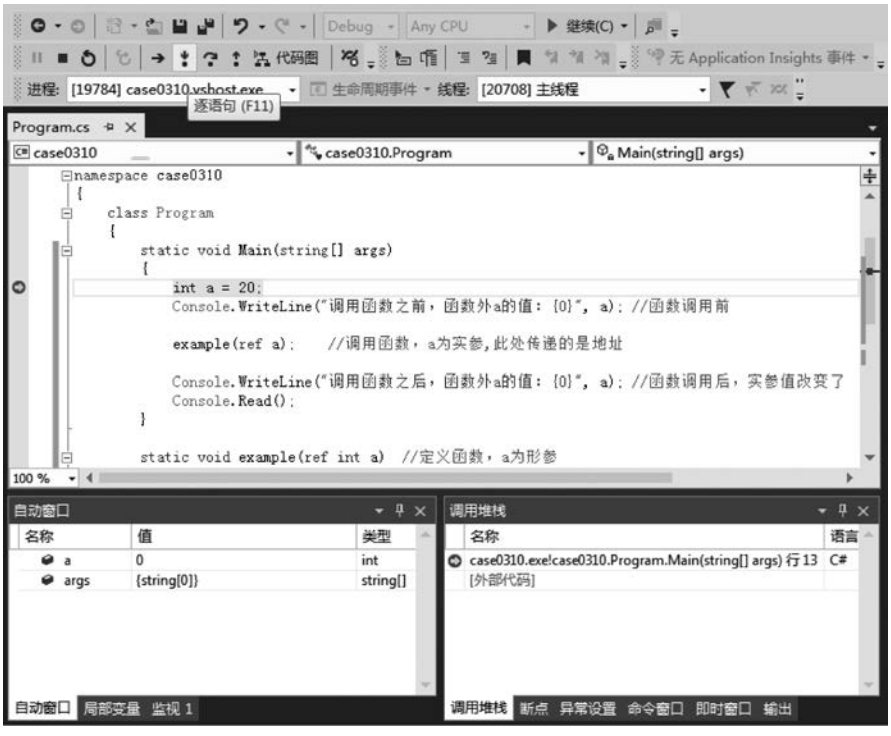

图 3.13　程序调试

注意，此时该行语句尚未被执行。接下来开始进行程序的调试，程序的调试运行方式主要有三种，分别是“逐语句”“逐过程”“全部执行”。

① 逐语句。

逐语句的调试快捷键是 F11 键，即每按一次 F11 键，程序执行一条语句。可以跟踪每个变量在程序执行过程中值的变化情况，如图 3.14 所示。

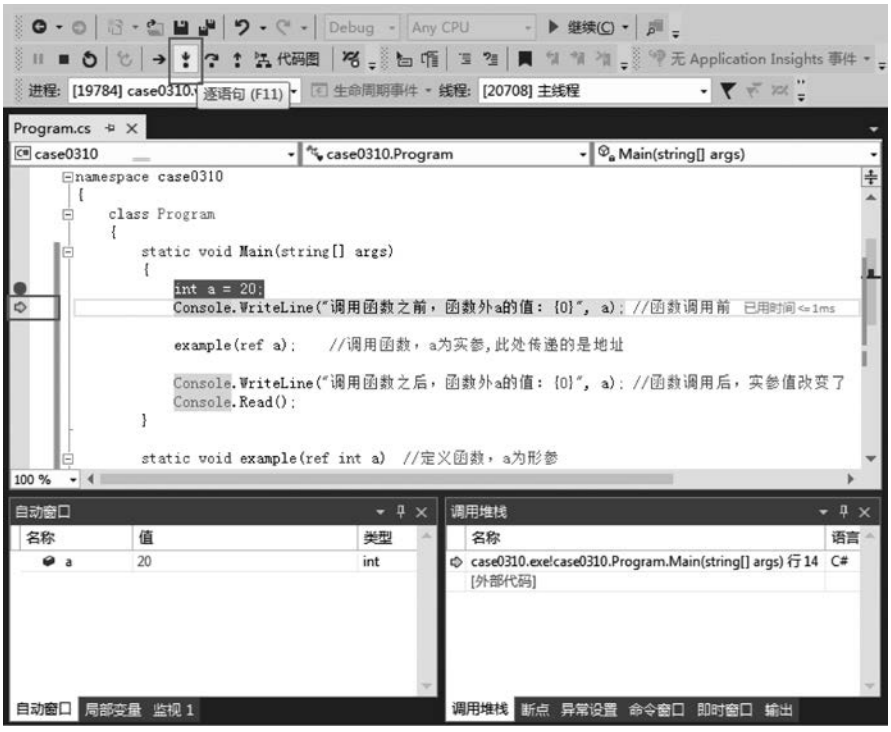

图 3.14 逐语句调试

② 逐过程。

逐过程的调试快捷键是 F10 键，即每按一次 F10 键，程序执行一个过程块。在简单的语句跟踪时，逐过程调试也是一条一条语句地跟踪执行。但当调试过程中遇到调用其他函数时，逐条语句调试将继续进入被调用函数并逐语句跟踪。而逐过程调试则将被调用函数当成一个整体，只跟踪被调用函数的运行结果返回值，而不监视被调用函数内部语句。

③ 全部执行。

全部执行的快捷键是 F5 键，即按一次 F5 键，程序将从断点处往下全部执行，直到遇到下一处断点。

(3) 变量值的跟踪。

在程序调试过程中，系统将自动显示一个“自动窗口”，用于实时显示当前程序中各个变量的值。如果该窗口被关闭，也可以使用“添加监视”和 “快速监视”等命令对各变量值进行监视。跟踪变量值的方法为：右击要跟踪的变量，在弹出的快捷菜单中选择“添加监视”命令，系统就会将该变量添加到监视面板中，如图 3.15 所示。

为变量添加监视后，可以实时查看每条语句执行时该变量的值的变化情况，从而使程序

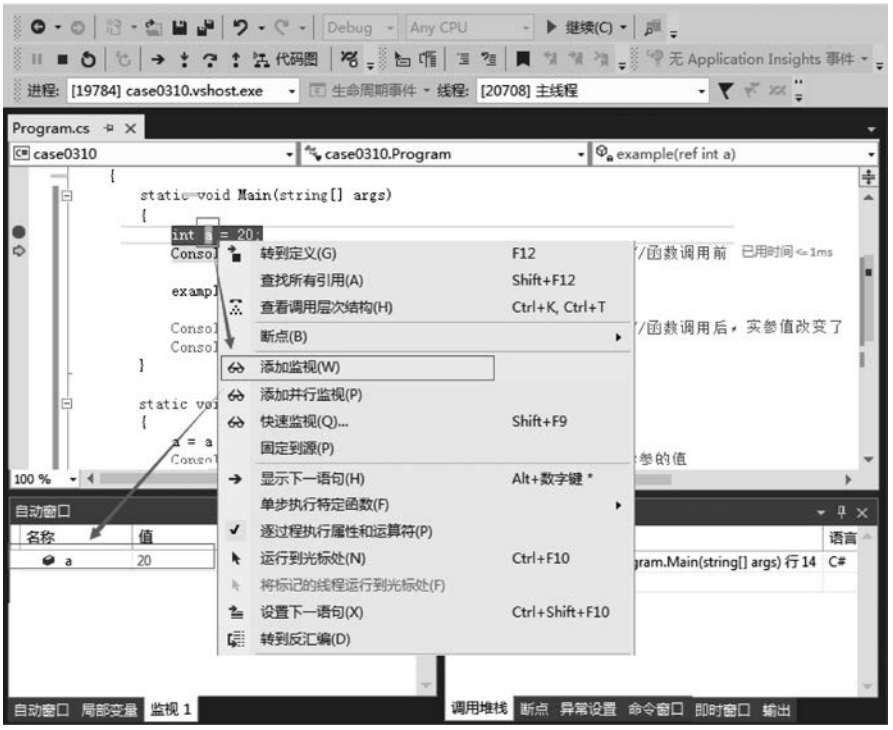

图 3.15　为变量添加监视

员更加清楚地知道程序的运行情况,方便程序员找出程序中的错误。

3.3.2　程序错误分析

程序的错误主要有三种,分别是语法错误、运行时错误和逻辑错误。

1. 语法错误

语法错误也称为编译错误,是指在源代码编写过程中不符合语法规则导致的错误,如变量名错误、表达式书写错误等。Visual Studio 编辑器会在发生错误的源代码处使用红色波浪线标出,编译后在集成开发环境下的错误列表窗口列出所有错误,双击错误条目就可以定位到出现错误的位置。语法错误示例如图 3.16 所示。

2. 运行时错误

运行时错误是指程序已经通过编译并能够正常运行,但是在运行过程中由于用户输入错误、磁盘出错、数据库无法使用、网络不可用等造成的程序出错。

例如,下面程序尝试计算 100 的阶乘,但由于计算的值超出了变量 b 的范围,造成程序运行错误。

```
int a = 1,b = 1;
while (a <= 100)
{
    b *= a;
    a++;
}
```

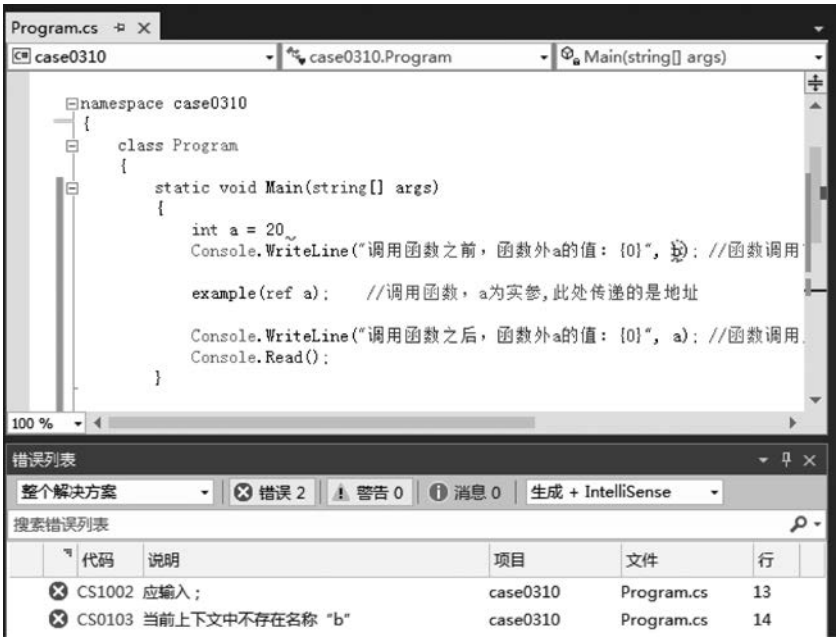

图 3.16 语法错误示例

3. 逻辑错误

如果程序没有出现语法错误，但运行后得到的实际结果与预期结果不符合，则说明该程序存在逻辑错误。例如，语句次序不对、逻辑判断不正确、循环语句条件不正确等。程序员应仔细阅读分析程序、通过调试器来帮助分析错误位置并分析产生错误的原因。

3.3.3 程序异常处理

在软件项目开发过程中，经常出现由于程序本身的缺陷或程序输入的不确定性原因，如程序源代码的错误、用户输入非法数据等，这些都会导致程序运行时发生错误或出现意外的情况，这就是程序的异常。.NET 环境提供了一个基于异常对象和保护源代码块的异常处理模型，它提供了能在程序中定义一个异常控制处理模块的程序控制机制来处理异常情况，并自动将出错时的流程交给异常控制处理模块处理，以保证程序能继续向前执行或正常结束。

try…catch…finally 语句允许在 try 子句中放置可能发生异常情况的程序源代码，对这些程序源代码进行监控。在 catch 子句中放置处理错误的程序源代码，以处理程序发生的异常。在 finally 子句中放置最后要执行的操作源代码，即在任何情形中都必须执行的源代码。try…catch…finally 语句的语法格式如下：

```
try
{  被监控代码  }
catch (异常类名 2 异常变量名 1)
{  异常处理代码  }
catch (异常类名 2 异常变量名 2)
{  异常处理代码  }
finally
{  最后要执行的代码  }
```

语法说明：

(1) try 语句只能出现一次，catch 语句可以出现多次，每一个 catch 语句分别负责一种类型的异常处理。

(2) 如果程序运行过程中没有出现异常，则不会运行 catch 语句。在 catch 子句中，异常类名必须为 System. Exception 或从 System. Exception 派生的类型。

(3) 不论程序运行过程中是否出现异常，都会执行 finally 语句。finally 语句块用于清除 try 语句块中分配的任何资源以及运行任何即使在发生异常时也必须执行的源代码。系统控制总是传递给 finally 语句块，与 try 语句块的退出方式无关。

【例 3.12】 本例学习使用 try…catch…finally 语句完成对输入信息中异常的处理。

【实现步骤】

创建一个控制台应用程序，命名为 case0312，实现输入用户的姓名和年龄并且输出，具体源代码如下：

```
static void Main(string[] args)
{
    Console.WriteLine("请输入您的姓名:");
    string name = Console.ReadLine();
    Console.WriteLine("您的姓名是:" + name);
    Console.WriteLine("请输入您的年龄:");
    int age = int.Parse(Console.ReadLine());
    Console.WriteLine("您的年龄是:" + age + "岁");
    Console.Read();
}
```

保存项目文件并运行程序，在界面中分别输入用户名“Jianguo”和年龄“18”，程序将输入的年龄字符串"18"成功转换为整数 18 并且显示，程序运行效果如图 3.17 所示。

图 3.17 程序运行效果

重新运行程序，当输入错误的年龄格式(例如"aaa")时，程序由于无法将其转换为整数，导致整个程序崩溃，效果如图 3.18 所示。

由于用户可以在界面上输入任意字符，因此，年龄输入这个功能属于潜在的异常输入场景，程序需要对此进行防范控制。接下来使用 try…catch 语句进行可能出现的异常情况的捕捉和处理，具体源代码如下：

```
static void Main(string[] args)
{
    Console.WriteLine("请输入您的姓名:");
    string name = Console.ReadLine();
    Console.WriteLine("您的姓名是:" + name);
    Console.WriteLine("请输入您的年龄:");
    try
    {
```

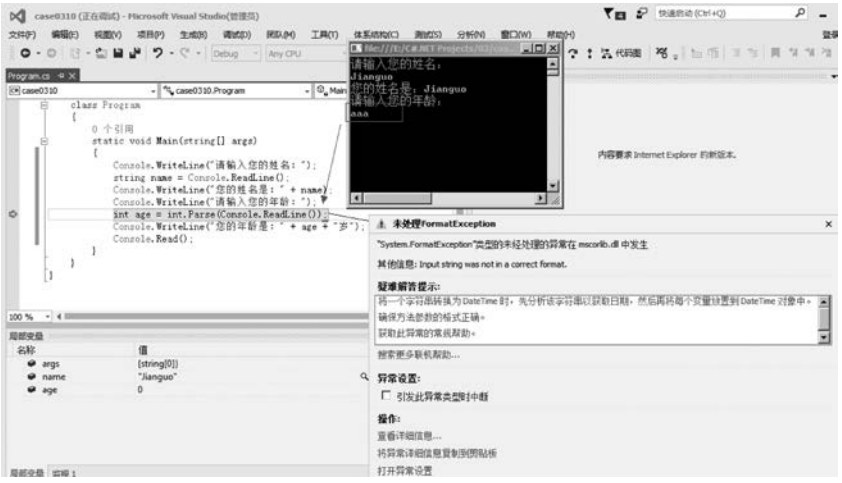

图 3.18 程序遇到异常情况运行效果

```
            int age = int.Parse(Console.ReadLine());
            Console.WriteLine("您的年龄是:" + age + "岁");
        }
        catch (Exception er)                    //如果出现异常
        {
            Console.WriteLine("对不起,您输入的年龄格式不正确!,提示信息是:" + er.Message);
        }
        Console.Read();
    }
```

运行程序,重新输入错误的年龄字符,此时,程序并没有崩溃,而是根据预定义的异常处理流程,给予相应的提示,程序运行效果如图 3.19 所示。

图 3.19 程序异常处理结果

在本书后续的所有编程实验中,读者可以对各个程序中的潜在异常场景进行判断,自行在相应的源代码段中增加异常处理功能。

3.4 习题

一、选择题

1. 要使用程序逐语句运行需要按(　　)键。

A. F4　　B. F5　　C. F7　　D. F11

2. C#程序中,ArrayList 集合类是位于(　　)命名空间下。

A. System. Data　　B. System. Collections

C. System. IO　　D. System. Windows

二、填空题

1. C#函数调用中,参数传递方式有两种,分别是_____和_____。

2. C#程序调试过程中,运行程序需要按_____键,逐条语句运行需要按_____键。

三、判断题

1. (　　)在 C#中,int[][]是定义一个 int 型的二维数组。

2. (　　)C#源代码书写规则中,每个函数都不能写注释,也不需要解释函数的功能。

3. (　　)C#源代码书写规则中,最好将所有类源代码都写在同一个文件中。

4. (　　)try…catch…finally 语句允许在 try 子句中放置可能发生异常情况的程序源代码,对这些程序源代码进行监控。

5. (　　)程序开发过程中,可以手动选择逐条语句执行、逐语句块执行或全部执行。