

# 第 7 章

## 调试和错误处理

本章内容:

---

- IDE 中的调试方法
- C#中的错误处理技术

本书到目前为止介绍了在 C#中进行简单编程的所有基础知识。在讨论本书后面章节的面向对象编程之前,先看看 C#代码中的调试和错误处理问题。

代码中有时难免存在错误。无论程序员多么优秀,程序总是会出现一些问题,出色的程序员会找出其中一部分错误,并更正它们。当然,一些问题比较小,不会影响应用程序的执行,例如,按钮上的拼写错误等,但一些错误可能比较严重,会导致应用程序完全失败(通常称为致命错误),致命错误包括妨碍代码编译的简单错误(语法错误),或者只在运行期间发生的更严重的错误。一些错误可能会更微妙。也许应用程序不能给数据库添加一个记录,因为遗漏了一个请求的字段,或者在其他有限制的环境中把错误的的数据添加到记录中。应用程序的逻辑在某些方面有瑕疵时,就会产生这样的错误,此类错误称为语义错误(或逻辑错误)。

当应用程序的用户抱怨说程序不能正常工作时,就出现了比较难以处理的错误。此时需要跟踪代码,确定发生了什么问题,并修改代码,使之按照希望的那样工作。在此类情况下,VS 和 VCE 的调试功能就可以大显身手了。本章的第一部分就介绍一些调试技巧,并把它们应用到一些常见问题上。

接着,讨论 C#中的错误处理技术。利用它们,可以对可能发生错误的地方采取预防措施,并编写弹性代码来处理可能会致命的错误。这些是 C#语言的一部分,而不是调试功能,但 IDE 也提供了一些工具来帮助我们处理错误。

### 7.1 VS和VCE中的调试

前面提到,可以来用两种方式执行应用程序:调试模式或非调试模式。在 VS 或 VCE 中执行应用程序时,它默认在调试模式下执行。例如,按下 F5 键或单击工具栏中的绿色“启动调试”(Play)

按钮(▶)时,就是在调试模式下执行应用程序。要在非调试模式下执行应用程序,应选择“**调试 ⇌ 开始执行(不调试)**”【Debug ⇌ Start Without Debugging】,或者按下 Ctrl+F5 键。

VS 和 VCE 都允许在两种配置下创建应用程序:调试(默认)和发布(实际上,还可以定义其他配置,但这是一种高级技术,本书不涉及)。使用标准工具栏中的“**解决方案配置**”(Solution Configurations)下拉框可以在这两种配置之间切换。



在 VCE 中,默认情况下不激活这个下拉列表。为了阅读本章,应启用它,方法是选择“**工具⇌选项**”(Tools⇌Options),在“**选项**”(Options)对话框中选择“**显示所有设置**”(Show All Settings),再选择“**项目和解决方案**”(Projects and Solutions)类别中的“**常规**”(General)子类别,启用“**显示高级生成配置**”(Show Advanced Build Configurations)选项。

在调试配置下生成应用程序,在调试模式下运行程序时,并不仅仅是运行编写好的代码。调试程序包含了应用程序的符号信息,所以 IDE 知道执行每行代码时发生了什么。符号信息意味着跟踪(例如)未编译代码中使用的变量名,这样,它们就可以匹配已编译的机器码应用程序中现有的值,而机器码程序不包含人们易于读取的信息。此类信息包含在.pdb 文件中,这些文件位于计算机的 Debug 目录下。它们可以执行许多有用的操作,包括:

- 向 IDE 输出调试信息
- 在执行应用程序期间查看和编辑变量的值
- 暂停程序和重启程序
- 在代码的某个位置自动暂停程序的执行
- 一次执行程序中的一行代码
- 在应用程序的执行期间监视变量内容的变化
- 在运行期间修改变量内容
- 测试函数的调用

在发布配置中,优化应用程序代码,但我们不能执行这些操作。但发布版本运行得比较快,完成了应用程序的开发时,一般应给用户提提供发布版本,因为发布版本不需要调试版本所包含的符号信息。

本节介绍调试技巧,以及如何使用它们确定未按预期方式执行的那些代码,并修改它们,这个过程称为调试。按照这些技术的使用方法把它们分为两个部分。一般情况下,可以先中断程序的执行,再进行调试,或者注上标记,以便以后加以分析。在 VS 和 VCE 术语中,应用程序可以处于运行状,也可以处于中断模式,即暂停正常的执行。下面首先介绍非中断模式(运行期间或正常执行)技术。

### 7.1.1 非中断(正常)模式下的调试

本书常常使用的一个命令是 Console.WriteLine()函数,它可以把文本输出到控制台上。在开发应用程序时,这个函数可以方便地获得操作的额外反馈,例如:

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc("Do something.");
Console.WriteLine("MyFunc() Function execution completed.");
```

这段代码说明了如何获取 MyFunc()函数的额外信息。这么做完全正确,但控制台的输出结果会

比较混乱。在开发其他类型的应用程序时，如 Windows 窗体应用程序，没有用于输出信息的控制台。作为一种替代方法，可以把文本输出到另一个位置上——IDE 中的“输出”(Output)窗口。

第2章简要介绍了“错误列表”(Error List)窗口，其他窗口也可以显示在这个位置上。其中一个窗口就是“输出”(Output)窗口，在调试时这个窗口非常有用。要显示这个窗口，可以选择“视图 ⇌ 输出”(View ⇌ Output)。在这个窗口中，可以查看与代码的编译和执行相关的信息，包括在编译过程中遇到的错误等，还可以将自定义的诊断信息直接写到窗口中，来使用这个窗口显示自定义信息。该窗口如图 7-1 所示。

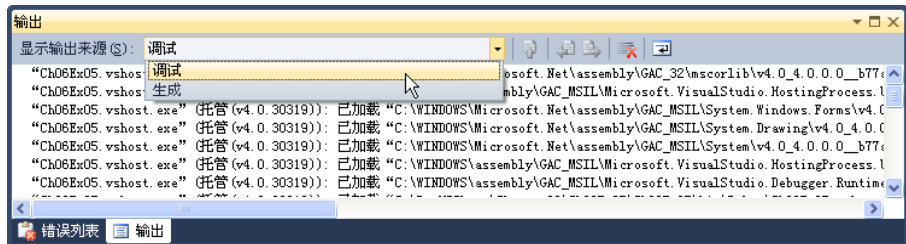


图 7-1



“输出”(Output)窗口有两种模式“生成”和“调试”(Build 和 Debug)，使用其中的下拉列表可以选择这些模式。“生成”和“调试”(Build 和 Debug)模式分别显示编译和运行期间的信息。本节提到“写入输出(Output)窗口”时，实际上是指“写入输出(Output)窗口的调试(Debug)模式视图”。

另外，还可以创建一个日志文件，在运行应用程序时，会把信息添加到该日志文件中。把信息写入日志文件所用的技巧与把文本写到“输出”(Output)窗口上所用的技巧相同，但需要理解如何从 C# 应用程序中访问文件系统。我们把这个功能放在后面的章节中，因为目前不必了解文件访问技巧也可以完成很多工作。

### 1. 输出调试信息

在运行期间把文本写入“输出”(Output)窗口是非常简单的。只要用需要的调用替代 Console.WriteLine() 调用，就可以把文本写到希望的地方。此时可以使用如下两个命令：

- Debug.WriteLine()
- Trace.WriteLine()

这两个命令函数的用法几乎完全相同，但有一个重要区别。第一个命令仅在调试模式下运行，而第二个命令还可用于发布程序。实际上，Debug.WriteLine() 命令甚至不能编译为可发布的程序，在发布版本中，该命令会消失，这肯定有其优点(首先，编译好的代码文件比较小)。实际上，一个源文件可以创建出两个版本的应用程序。调试版本显示所有的额外诊断信息，而发布版本没有这个开销，也不向用户显示信息，否则会引起用户的反感。

这两个函数的语法与 Console.WriteLine() 是不同的。其唯一的字符串参数用于输出消息，而不需要使用 {X} 语法插入变量值。这意味着必须使用 + 等串联运算符在字符串中插入变量值。它们还可以有第二个字符串参数，用于显示输出文本的类别，这样，如果应用程序的不同地方输出了类似的消息，我们就可以马上确定“输出”(Output)窗口中显示的是哪些输出信息。

这些函数的一般输出如下所示：

```
<category>: <message>
```

例如，下面的语句把 MyFunc 作为可选的类别参数：

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

其结果为：

```
MyFunc: Added 1 to i
```

下面的示例按这种方式输出调试信息。

### 试一试：把文本输出到“输出”(Output)窗口

(1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新的控制台应用程序 Ch07Ex01。

(2) 修改代码，如下所示：



可从  
wrox.com  
下载源代码

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

namespace Ch07Ex01
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] testArray = { 4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9 };
            int[] maxValIndices;
            int maxVal = Maxima(testArray, out maxValIndices);
            Console.WriteLine("Maximum value {0} found at element indices:",
                             maxVal);
            foreach (int index in maxValIndices)
            {
                Console.WriteLine(index);
            }
            Console.ReadKey();
        }
        static int Maxima(int[] integers, out int[] indices)
        {
            Debug.WriteLine("Maximum value search started.");
            indices = new int[1];
            int maxVal = integers[0];
            indices[0] = 0;
            int count = 1;
            Debug.WriteLine(string.Format(
                "Maximum value initialized to {0}, at element index 0.", maxVal));
            for (int i = 1; i < integers.Length; i++)
            {
                Debug.WriteLine(string.Format(
```

```

        "Now looking at element at index {0}.", i));
    if (integers[i] > maxVal)
    {
        maxVal = integers[i];
        count = 1;
        indices = new int[1];
        indices[0] = i;
        Debug.WriteLine(string.Format(
            "New maximum found. New value is {0}, at element index {1}.",
            maxVal, i));
    }
    else
    {
        if (integers[i] == maxVal)
        {
            count++;
            int[] oldIndices = indices;
            indices = new int[count];
            oldIndices.CopyTo(indices, 0);
            indices[count - 1] = i;
            Debug.WriteLine(string.Format(
                "Duplicate maximum found at element index {0}.", i));
        }
    }
}
Trace.WriteLine(string.Format(
    "Maximum value {0} found, with {1} occurrences.", maxVal, count));
Debug.WriteLine("Maximum value search completed.");
return maxVal;
}
}
}

```

代码段 Ch07Ex01\Program.cs

(3) 在“调试”(Debug)模式下执行代码,结果如图7-2所示。



图 7-2

(4) 中断应用程序的执行,查看“输出”(Output)窗口中的内容(在“调试”【Debug】模式下),如下所示(有删节):

```

...
Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.

```

```

Now looking at element at index 4.
Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The thread 'vshost.RunParkingWindow' (0x110c) has exited with code 0 (0x0).
The thread '<No Name>' (0x688) has exited with code 0 (0x0).
The program '[4568] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with
code 0 (0x0).

```

(5) 使用标准工具栏上的下拉列表切换到 **Release** 模式, 如图 7-3 所示。

(6) 再次运行程序, 这次是在 **Release** 模式下运行, 并在执行中止时, 再查看一下“输出”(Output)窗口。结果如下所示(有删节):

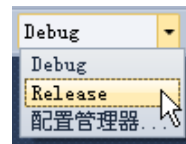


图 7-3

```

...
Maximum value 9 found, with 2 occurrences.
The thread 'vshost.RunParkingWindow' (0xa78) has exited with code 0 (0x0).
The thread '<No Name>' (0x130c) has exited with code 0 (0x0).
The program '[4348] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with
code 0 (0x0).

```

### 示例的说明

这个应用程序是第 6 章中一个示例的扩展版本, 它使用一个函数计算整数数组中的最大值。这个版本也返回一个索引数组, 表示最大值在数组中的位置, 以便调用代码处理这些元素。

首先在代码开头使用了一个额外的 **using** 指令:

```
using System.Diagnostics;
```

这简化了本例前面讨论的函数访问, 因为它们包含在 **System.Diagnostics** 名称空间中, 没有这个 **using** 语句, 下面的代码:

```
Debug.WriteLine("Bananas");
```

就需要进一步加以限制, 重新编写这行语句, 如下所示:

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

**using** 语句使代码更简单, 缩短了代码的长度。

**Main()** 中的代码仅初始化一个测试用的整数数组 **testArray**, 并声明了另一个整数数组 **maxValIndices**, 以存储 **Maxima()** 的索引输出结果(执行计算的函数), 接着调用这个函数。函数返回后, 代码就会输出结果。

**Maxima()** 稍复杂一些, 但没有使用前面介绍的那么多代码。在数组中进行搜索的方式与第 6 章

的 `MaxVal()` 函数类似，但要用一个记录存储最大值的索引。

特别需要注意用来跟踪索引的函数(而不是输出调试信息的那些代码行)。`Maxima()` 并没有返回一个足以存储源数组中每个索引的数组(需要与源数组有相同的维数)，而是返回一个正好能容纳搜索到的索引的数组。这可以在搜索过程中连续重建不同长度的数组来实现。这是必要的，因为一旦创建好数组，就不能重新设置长度。

开始搜索时，假定源数组(本地称为 `integers`)中的第一个元素就是最大值，且数组中只有一个最大值。因此可以为 `maxVal`(函数的返回值，即搜索到的最大值)和 `indices`(out 参数数组，存储搜索到的最大值的索引)设置值。`maxVal` 被赋予 `integers` 中第一个元素的值，`indices` 被赋予一个值 0，即数组中第一个元素的索引。在变量 `count` 中存储搜索到的最大值的个数，以跟踪 `indices` 数组。

函数的主体是一个循环，它迭代 `integers` 数组中的各个值，但忽略第一个值，因为它已经处理过了。每个值都与 `maxVal` 的当前值进行比较，如果 `maxVal` 更大，就忽略该值。如果当前处理的值比 `maxVal` 大，就修改 `maxVal` 和 `indices`，以反映这种情况。如果当前处理的值与 `maxVal` 相等，就递增 `count`，用一个新数组替代 `indices`。这个新数组比旧 `indices` 数组多一个元素，包含搜索到的新索引。

最后一个功能的代码如下所示：

```
if (integers[i] == maxVal)
{
    count++;
    int[] oldIndices = indices;
    indices = new int[count];
    oldIndices.CopyTo(indices, 0);
    indices[count - 1] = i;
    Debug.WriteLine(string.Format(
        "Duplicate maximum found at element index {0}.", i));
}
```

这段代码把旧 `indices` 数组备份到 `if` 代码块的 `oldIndices` 局部整型数组中。注意使用 `<array>.CopyTo()` 函数把 `oldIndices` 中的值复制到新的 `indices` 数组中。这个函数的参数是一个目标数组和一个用于复制第一个元素的索引，并把所有的值都粘贴到目标数组中。

在代码中，各个文本部分都使用 `Debug.WriteLine()` 和 `Trace.WriteLine()` 函数来进行输出，这些函数使用 `string.Format()` 函数把变量值嵌套在字符串中，其方式与 `Console.WriteLine()` 相同。这比使用 `+` 串联运算符更加高效。

在 `Debug` 模式下运行应用程序时，其最终结果是一个完整的记录，它记述了在循环中计算出结果所采取的步骤。在 `Release` 模式下，仅能看到计算的最终结果，因为没有调用 `Debug.WriteLine()` 函数。

除了这些 `WriteLine()` 函数外，还需要注意其他一些问题。首先是 `Console.Write()` 的等价函数：

- `Debug.Write()`
- `Trace.Write()`

这两个函数使用的语法与 `WriteLine()` 函数相同(一个或两个参数，即一个信息和可选的类别)，但它们是有区别的，因为它们没有添加行尾字符。

还有下列命令：

- `Debug.WriteLineIf()`
- `Trace.WriteLineIf()`
- `Debug.WriteLineIf()`

## ● Trace.WriteLine()

这些函数的参数都与没有 if 的对应函数相同，但增加了一个必选参数，且该参数放在列表参数的最前面。这个参数的值为布尔值(或者计算结果为布尔值的表达式)，只有这个值为 `true` 时，函数才会输出文本。使用这些函数可以有条件地把文本输出到“输出”(Output)窗口中。

例如，只需在某些情况下输出调试信息，所以代码中有许多 `Debug.WriteLineIf()` 语句，它们都取决于具体的条件。如果没有这个条件，就不显示它们，以防“输出”(Output)窗口显示多余的信息。

## 2. 跟踪点

另一种把信息输出到“输出”(Output)窗口中的方法是使用跟踪点。这是 VS 的一个功能，而不是 C# 的功能，但其作用与使用 `Debug.WriteLine()` 相同。它实际上是输出调试信息且不修改代码的一种方式。



只能在 VS 中使用跟踪点，不能在 VCE 中使用。如果读者使用的是 VCE，就可以跳过本节。

为了演示跟踪点，可以使用它们替代上一个示例中的调试命令(请参阅本章的下载代码中的 `Ch07Ex01TracePoints` 文件)。添加跟踪点的过程如下所示：

- (1) 把光标放在要插入跟踪点的代码行上。注意，跟踪点会在执行这行代码之前被处理。
- (2) 右击该行代码，选择“断点 ⇄ 插入跟踪点”(Breakpoint ⇄ Insert Tracepoint)。
- (3) 在打开的“命中断点时”(When Breakpoint Is Hit)对话框中，在“打印消息”(Print a Message): 文本框中键入要输出的字符串。如果要输出变量值，应把变量名放在花括号中。
- (4) 单击“确定”(OK)按钮。在包含跟踪点的代码行左边会出现一个红色的菱形，该行代码也会突出显示为红色。

看一下添加跟踪点的对话框标题和所需要的菜单选项，显然，跟踪点是断点的一种形式(可以暂停应用程序的执行，就像断点一样)。断点一般用于更高级的调试目的，本章稍后将介绍断点。

图 7-4 显示了 `Ch07Ex01TracePoints` 中第 31 行所需的跟踪点。在删除已有的 `Debug.WriteLine()` 语句后，对代码行编号。

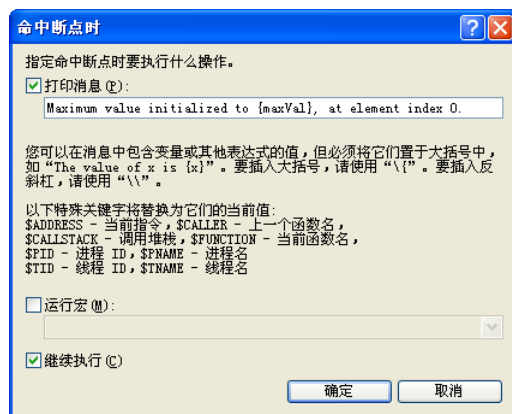


图 7-4





如图 7-4 中的文本所示,跟踪点允许插入与跟踪点的位置和上下文相关的其他有用信息。用户应试着使用这些值,尤其是\$FUNCTION 和\$CALLER,看看可以得到什么额外信息。还可以看出,跟踪点可以执行宏,但这是一个高级功能,这里不予介绍。

还有一个窗口(只能在 VS 中使用)可用于快速查看应用程序中的跟踪点。要显示这个窗口,可以从 VS 菜单中选择“调试 ⇨ 窗口 ⇨ 断点”(Debug ⇨ Windows ⇨ Breakpoints)。这是显示断点的通用窗口(如前所述,跟踪点是断点的一种形式),可以定制显示的内容,从这个窗口的“列”(Columns)下拉框中添加“命中条件”(When Hit)列,显示与跟踪点关系更密切的信息。图 7-5 显示的窗口配置了这个列,还显示了添加到 Ch07Ex01TracePoints 中的所有跟踪点。

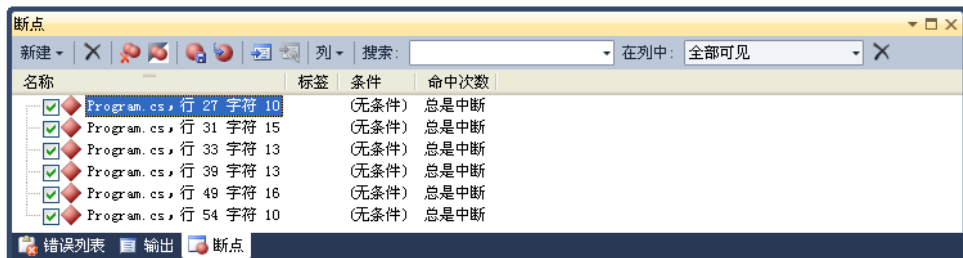
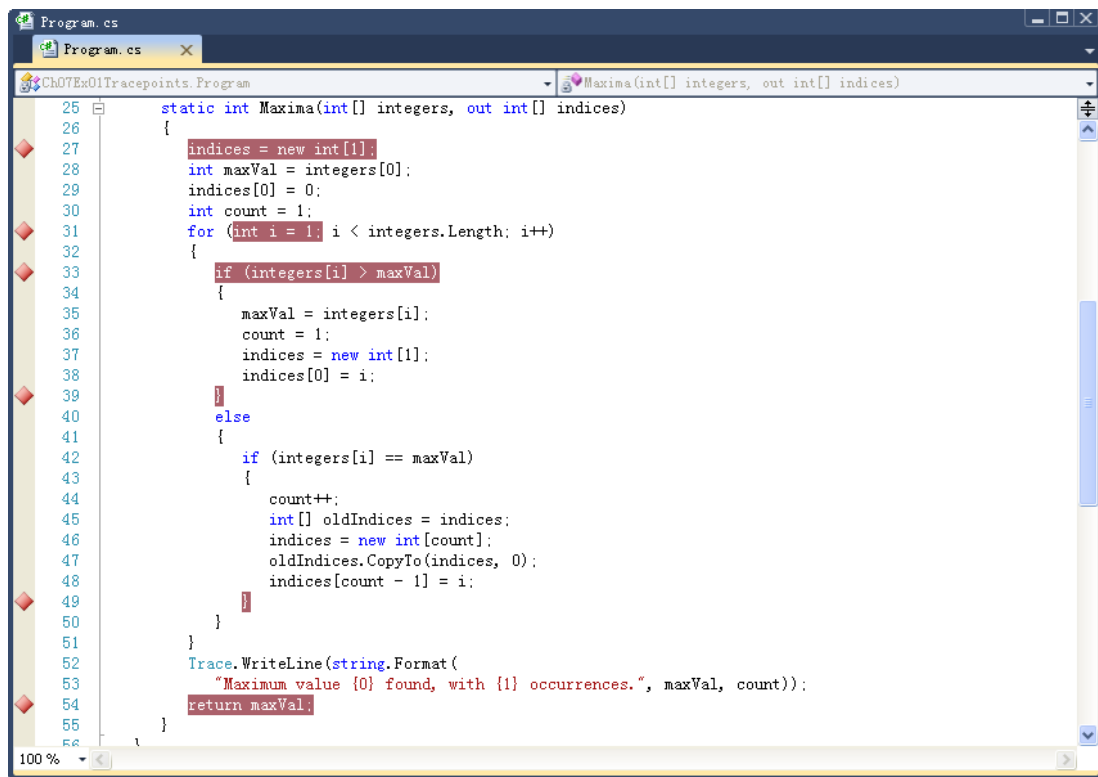


图 7-5

在调试模式下执行这个应用程序，会得到与前面完全相同的结果。在代码窗口中右击跟踪点，或者利用“断点”(Breakpoints)窗口，就可以删除或临时禁用跟踪点。在“断点”(Breakpoints)窗口中，跟踪点左边的复选框确定是否启用跟踪点；禁用的跟踪点未被选中，在代码窗口中显示为菱形框，而不是实心菱形。

### 3. 诊断输出与跟踪点

前面介绍了两种输出相同信息的方法，下面看看它们的优缺点。首先，跟踪点与 Trace 命令并不等价，也就是说，不能使用跟踪点在发布版本中输出信息。这是因为跟踪点并没有包含在应用程序中。跟踪点由 VS 处理，在应用程序的已编译版本中，跟踪点是不存在的。只有应用程序运行在 VS 调试器中时，跟踪点才起作用。

跟踪点的主要缺点也是其优点，即它们存储在 VS 中，因此可以在需要时快速、方便地添加到应用程序中，而且也非常容易删除。如果输出非常复杂的信息字符串，觉得跟踪点非常讨厌，只需单击表示其位置的红色菱形，就可以删除跟踪点。

跟踪点的一个优点是允许方便地添加额外的信息，如上一节提到的 \$FUNCTION。这个信息可以用 Debug 和 Trace 命令来编写，但比较难。总之，输出调试信息的两种方法是：

- **诊断输出：**总是要从应用程序中输出调试结果时使用这种方法，尤其是在要输出的字符串比较复杂，涉及几个变量或许多信息的情况下，使用该方法比较好。另外，如果要在发布模式下获得执行应用程序的调试结果，Trace 命令常常是唯一的选择。
- **跟踪点：**调试应用程序时，希望快速输出重要信息，以便解决语义错误，应使用跟踪点。另一个明显的区别是跟踪点只能在 VS 中使用，而诊断输出可以在 VS 和 VCE 中使用。

#### 7.1.2 中断模式下的调试

调试技术的剩余内容是在中断模式下工作。可以通过几种方式进入这种模式，这些方式都可以暂停程序的执行。

##### 1. 进入中断模式

进入中断模式的最简单方式是在运行应用程序时，单击 IDE 中的“全部中断”(Pause)按钮。这个“全部中断”(Pause)按钮在“调试”(Debug)工具栏上，应将该工具栏添加到 VS 默认显示的工具栏中。为此，右击工具栏区域，并选择“调试”(Debug)，这个工具栏如图 7-6 所示。

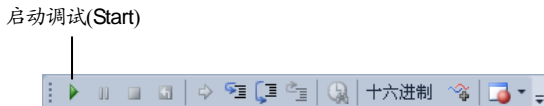


图 7-6

在这个工具栏上，前 4 个按钮可以手工控制中断。在图 7-6 上，其中的 3 个按钮显示为灰色，因为在程序没有运行时，它们是不能工作的。还有个按钮“启动调试”(Start)是可以使用的，这个按钮与标准工具栏上的“启动调试”(Start)按钮相同。在后面的章节需要其他的按钮时，再介绍它们。

运行一个应用程序时，工具栏就如图 7-7 所示。



图 7-7

现在, 就可以使用之前显示为灰色的 3 个按钮了。它们可以:

- 暂停应用程序的执行, 进入中断模式
- 完全停止应用程序的执行(不进入中断模式, 而是退出应用程序)
- 重新启动应用程序

暂停应用程序是进入中断模式的最简单方式, 但这并不能更好地控制停止程序运行的位置。我们可能会很自然地停止运行应用程序, 例如, 要求用户输入信息。还可以在长时间的操作或循环过程中进入中断模式, 但停止的位置可能相当随机。一般情况下, 最好使用断点。

### 断点

断点是源代码中自动进入中断模式的个标记, 可以在 VS 和 VCE 中使用, 但断点在 VS 中更加灵活。它们可以配置为:

- 在遇到断点时, 立即进入中断模式
- (只用于 VS)在遇到断点时, 如果布尔表达式的值为 true, 就进入中断模式
- (只用于 VS)遇到某断点一定的次数后, 进入中断模式
- (只用于 VS)在遇到断点时, 如果自从上次遇到断点以来变量的值发生了变化, 就进入中断模式
- (只用于 VS)把文本输出到“输出”(Output)窗口中, 或者执行一个宏(参见本章上一节)

注意, 上述功能仅能用于调试程序。如果编译发布程序, 将会忽略所有断点。

添加断点有几种方法。要添加简单断点, 当遇到该断点所在的代码行时, 就中断执行, 可以单击该代码行左边的灰色区域, 右击该代码行, 选择“断点 ⇄ 插入断点”(Breakpoint ⇄ Insert Breakpoint)菜单项; 选择“调试 ⇄ 切换断点”(Debug ⇄ Toggle Breakpoint); 或者按下 F9。

断点在该代码行的旁边显示为个红色的圆圈, 而该行代码也突出显示, 如图 7-8 所示。

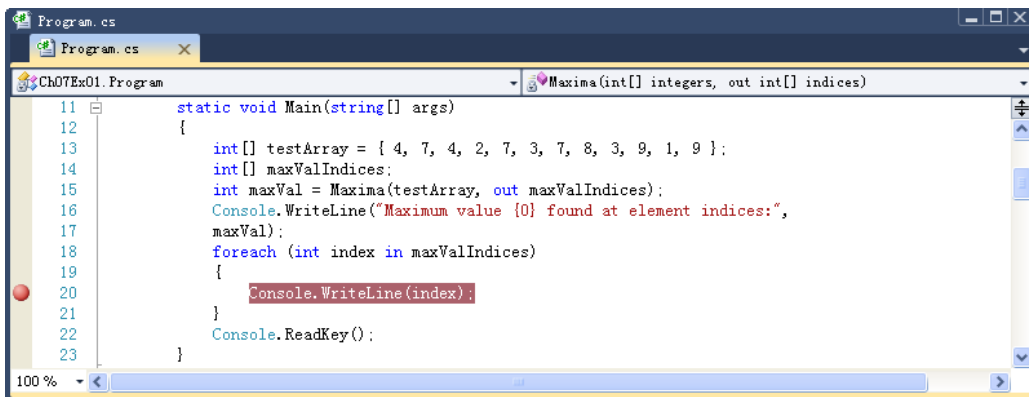


图 7-8

本节的剩余内容仅适用于 VS，不适用于 VCE。如果使用的是 VCE，可以跳到“进入中断模式的其他方式”一节。

在 VS 中，使用“断点”(Breakpoints)窗口还可以查看文件中的断点信息(在“跟踪点”一节中介绍过启用该窗口的方法)。在“断点”(Breakpoints)窗口中，可以禁用断点(删除描述信息左边的记号：禁用的断点用未填充的红色圆圈来表示)，删除断点，编辑断点的属性。

这个窗口中显示的“条件”(Condition)和“命中次数”(Hit Count)列是唯一的两个可用列，它们是非常有用的。右击断点(在代码或这个窗口中)，选择“条件”(Condition)或“命中次数”(Hit Count)菜单项，就可以编辑它们。

选择“条件”(Condition)按钮，将弹出一个对话框。在该对话框中可以键入任意布尔表达式，该表达式可以包含断点涉及的任何变量。例如，可以配置一个断点，输入表达式 `maxVal > 4`，选择“为 true”(Is true)选项，则在遇到这个断点，且 `maxVal` 的值大于 4 时，就会触发该断点。还可以检查这个表达式是否有变化，仅当发生变化时，断点才会被触发(例如，如果在遇到断点时，`maxVal` 的值从 2 改为 6，就会触发该断点)。

选择“命中次数”(Hit Count)按钮，将弹出一个对话框。在这个对话框中可以指定在触发前，要遇到该断点多少次。下拉列表提供了如下选项：

- 总是中断
- 中断，条件是命中次数等于(在 Hit Count 等于多少次时中断)
- 中断，条件是命中次数几倍于(在 Hit Count 是某个数的倍数时中断)
- 中断，条件是命中次数大于或等于(在 Hit Count 大于或等于多少次时中断)

所选的选项与在旁边的文本框中输入的值共同确定断点的行为。这个“命中次数”(Hit Count)按钮在比较长的循环中很有用，例如，在执行了前 5000 次循环后需要中断。如果不这么做，中断并重新启动 5000 次是很痛苦的。



带有附加属性集(例如，条件或遇到断点的次数)的断点，在显示时略有区别。已配置的断点不是显示一个简单的圆圈(●)，而是在红色圆圈中有一个白色的加号(⊕)。这是很有用的，因为它允许很快辨认出哪个断点总是进入中断模式，哪个断点只在某种情况下才进入中断模式。

### 进入中断模式的其他方式

进入中断模式还有两种方式。一种是在抛出一个未处理的异常时选择进入该模式。这种方式在本章后面讨论到错误处理时论述。另一种方式是生成一个判定语句(assertion)时中断。

判定语句是可以用用户定义的消息中断应用程序的指令。它们常常用于应用程序的开发过程，作为测试程序是否能平滑运行的一种方式。例如，在应用程序的某一处要给定的变量值小于 10，此时就可以使用一个判定语句，确定它是否为 `true`，如果不是，就中断程序的执行。当遇到判定语句时，可以选择“终止”(Abort)，中断应用程序的执行，也可以选择“重试”(Retry)，进入中断模式，还可以选择“忽略”(Ignore)，让应用程序像往常一样继续执行。

与前面的调试输出函数一样，判定函数也有两个版本：

- `Debug.Assert()`
- `Trace.Assert()`

其调试版本也是仅用于编译调试和序。

这两个函数带3个参数第一个参数是一个布尔值，其值为 false 会触发判定语句。第二、三个参数是两个字符串，分别把信息写到弹出的对话框和“输出”(Output)窗口中。上面的示例需要一个函数调用，如下所示：

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.",
    "Assertion occurred in Main().");
```

判定语句通常在应用程序的早期使用比较有效。可以分发应用程序的一个发布程序，其中包含 Trace.Assert() 函数，以列出各种信息。如果触发了判定语句，用户就会收到通知，把这些消息传递给开发人员。这样，即使开发人员不知道错误是如何发生的，也可以改正这个错误。

例如，在第一个字符串中提供错误的简短描述，在第二个字符串中提供下一步该如何操作的指示：

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
    "Please contact vendor with the error code KCW001.");
```

如果触发了这个判定语句，用户就会看到如图 7-9 所示的对话框。

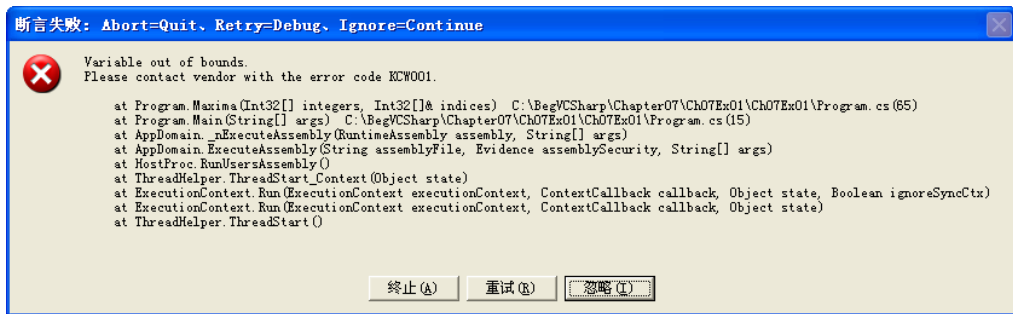


图 7-9

诚然，这并不是最友好的对话框，因为它包含了许多令人迷惑的信息，但如果用户给开发人员发送了错误的屏幕图，开发人员就可以很快找出问题所在。

下一个要论述的主题是应用程序中断，以及进入中断模式后，我们可以做什么。一般情况下，进入中断模式的目的是找出代码中的错误(或确保程序工作正常)。一旦进入中断模式，就可以使用各种技巧分析代码，分析应用程序在暂停处的确切状态。

## 2. 监视变量的内容

监视变量的内容是 VS 和 VCE 帮助我们使工作变得简单的一个方面。查看变量值的最简单方式是在中断模式下，使鼠标指向源代码中的变量名，此时就会出现一个黄色的工具提示，显示该变量的信息，其中包括该变量的当前值。

还可以高亮显示整个表达式，以相同的方式得到该表达式的结果。对于比较复杂的值，例如，数组，甚至可以扩展工具提示中的值，查看各个数组元素项。

注意，在运行应用程序时，IDE 中各个窗口的布局发生了变化，在默认情况下，运行期间会发生如下变化(变化的情况会根据具体的安装略有区别)：

- “属性” (Properties)窗口消失，其他一些窗口也会消失，包括“解决方案资源管理器” (Solution Explorer)窗口
- “错误列表” (Error List)窗口会被屏幕底部的两个新窗口代替
- 新窗口中会出现几个新的选项卡

新的屏幕布局如图 7-10 所示。这可能与读者的显示情况不完全相同，一些选项卡和窗口可能不完全匹配。但是，这些窗口的功能(后面将讨论)是相同的，这个显示完全可以通过“视图” (View)和“调试 ⇌ 窗口” (Debug ⇌ Windows)菜单来定制(在中断模式下)，也可以在屏幕上拖动窗口，重新设定它们的位置。

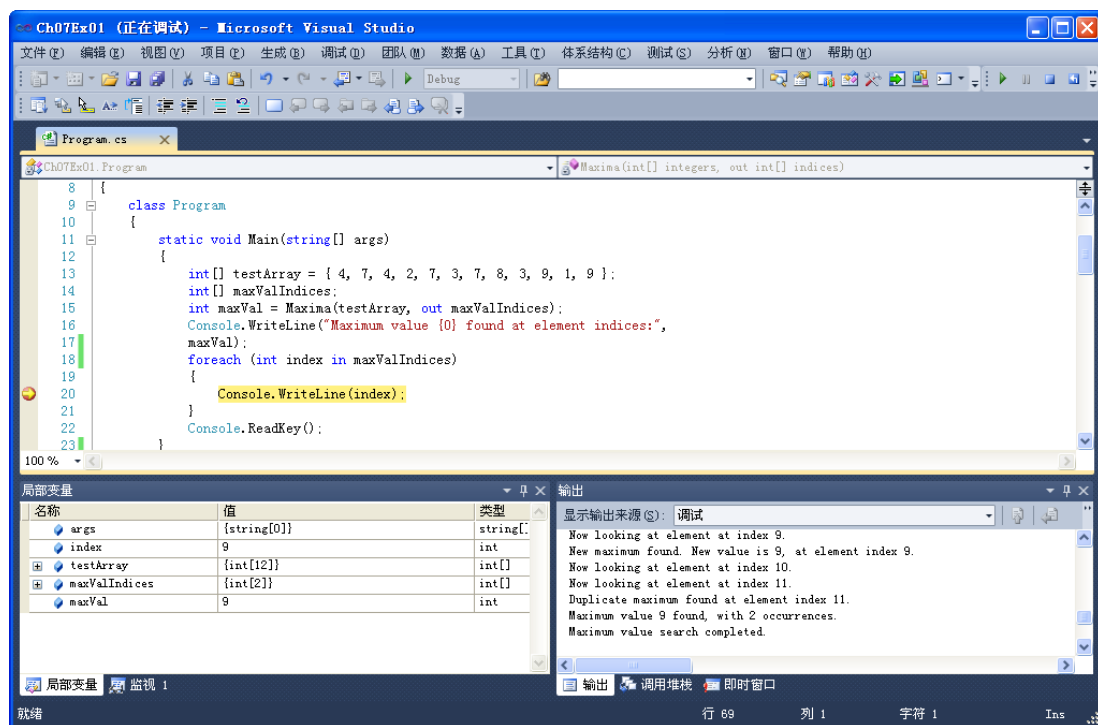


图 7-10

底部左角边的新窗口在调试时非常有用，它允许在中断模式下，在应用程序的变量值上保留标签，它包含 3 个选项卡，如下所示(在 VS 和 VCE 中有所不同)：

- “自动窗口” (Autos)【只在 VS 中有】——当前和前面的语句使用的变量(Ctrl+D, A)
- “局部变量” (Locals)——作用域内的所有变量(Ctrl+D, L)
- “监视 N”(Watch N)——可定制的变量和表达式显示【其中 N 从 1~4，在“调试 ⇌ 窗口 ⇌ 监视” (Debug ⇌ Windows ⇌ Watch)上】

这些选项卡的工作方式或多或少有些类似，并根据它们的特定功能添加了各种附加特性。一般情况下，每个选项卡都包含一个变量列表，其中包括变量的名称、值和类型等信息。更复杂的变量(如数组)可以使用变量名左边的+和-(展开/折叠)符号进一步查看，它们的内容可以以树状视图的方式显示。例如，在前面的示例中，在代码中放置了一个断点，得到的“局部变量” (Locals)选项卡如图 7-11 所示，其中显示了数组变量 'maxValIndices' 的展开视图。



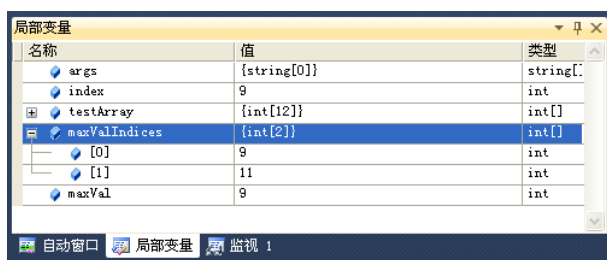


图 7-11

在这个视图中，还可以编辑变量的内容。它有效地绕过了前面代码中的其他变量赋值。为此，只需在“值”(Value)列中为要编辑的变量输入一个新值即可。也可以把这种技巧用于其他情况，例如，需要修改代码才能编辑变量值的情况。

可以通过“监视”(Watch)窗口【或 VS 中的“监视”(Watch)窗口，至多可以显示 4 个】监视特定变量或涉及特定变量的表达式。要使用这个窗口，只需在“名称”(Name)列中键入变量名或表达式，就可以查看它们的结果，注意，并不是应用程序中的所有变量在任何时候都在作用域内，并在“监视”(Watch)窗口中对变量做出标记。例如，图 7-12 显示了一个“监视”(Watch)窗口，其中包含几个示例变量和表达式，在遇到 Maxima() 函数末尾前面的一个断点时，会显示这个“监视”(Watch)窗口。

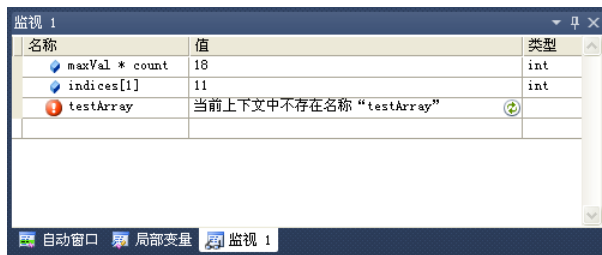


图 7-12

testArray 数组对于 Main() 来说是局部数组，所以在该图中没有值，而是显示了一个信息，告诉我们这个变量不在作用域内。



要在“监视”(Watch)窗口中添加变量，还可以把变量从源代码拖动到该窗口中。

在这个窗口中可以访问变量的各种显示结果，一个优点是它们可以显示变量在断点之间的变化情况，新值显示为红色而不是黑色，所以很容易看出哪个值发生了变化。

如前所述，要在 VS 中添加更多的“监视”(Watch)窗口，可以在中断模式下，使用“调试 ⇨ 窗口 ⇨ 监视 ⇨ 监视 N”(Debug ⇨ Windows ⇨ Watch ⇨ Watch N)菜单选项打开或关闭“监视”(Watch)的 4 个窗口。每个窗口都可以包含变量和表达式的一组观察结果，所以可以把相关的变量组合在一起，以便于访问。

除了这些“监视”(Watch)窗口外，VS 还有一个“快速监视”(QuickWatch)窗口，它能快速提供源代码中某个变量的详细信息。要使用这个窗口，可以右击要查看的变量，选择“快速监视”

(QuickWatch)菜单选项。但在大多数情况下，使用标准的“**监视**”(Watch)窗口就足够了。

“**监视**”(Watch)窗口可以在应用程序的各个执行过程之间保留下来。如果中断应用程序，再重新运行，就不必再次添加“**监视**”(Watch)窗口了，IDE 会记住上次使用的“**监视**”(Watch)窗口。

### 3. 单步执行代码

前面介绍了如何在中断模式下查看应用程序的运行情况，下面论述如何在中断模式下使用 IDE 单步执行代码，查看代码的执行结果，人们的思维速度不会比计算机运行得更快，所以这是一个极有价值的技巧。

进入中断模式后，在代码视图的左边，正在执行的代码旁边会出现一个光标(如果使用断点进入中断模式，该光标最初应显示在断点的红色圆圈中)，如图 7-13 所示。

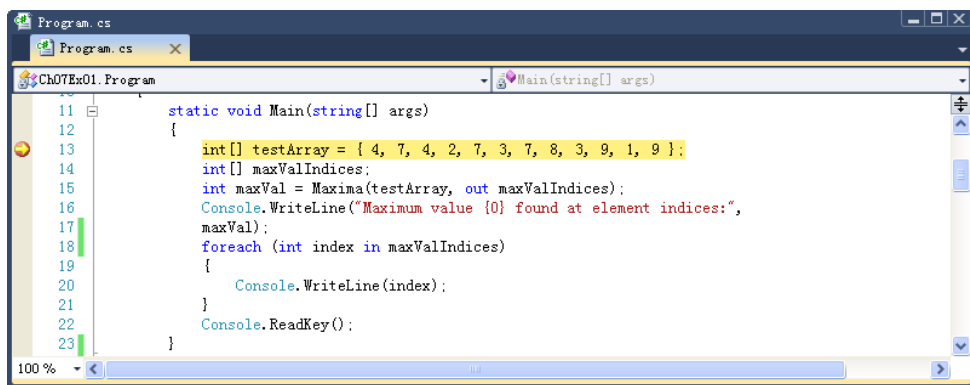


图 7-13

这显示了在进入中断模式时程序执行到的位置。在这个位置上，可以选择逐行执行。为此，使用前面看到的其他一些“**调试**”(Debug)工具栏按钮，如图 7-14 所示。



图 7-14

第 6、7、8 个图标控制了中断模式下的程序流。它们依次是：

- “**逐语句**”(Step Into)——执行并移动到下一个要执行的语句上
- “**逐过程**”(Step Over)——同上，但不进入嵌套的代码块，包括函数
- “**跳出**”(Step Out)——执行到代码块的末尾，在执行完该语句块后，重新进入中断模式

如果要查看应用程序执行的每个操作，可以使用“**逐语句**”(Step Into)按顺序执行指令，这包括在函数中执行，例如，上面示例中的 `Maxima()`。当光标到达第 15 行，调用 `Maxima()` 时，单击这个图标，会使光标移动到 `Maxima()` 函数内部的第一行代码上。而如果光标移到第 15 行时单击“**逐过程**”(Step Over)，就会使光标移动到第 16 行，不必进入 `Maxima()` 中的代码(但仍执行这段代码)。如果单步执行到不感兴趣的函数，可以单击“**跳出**”(Step Out)，返回到调用该函数的代码。在单步执行代码时，变量的值可能会发生变化。注意观察上一节讨论的“**监视**”(Watch)窗口，可以看到变量



值的变化情况。

在存在语义错误的代码中，这个技巧也许是最有效的。可以单步执行代码，当执行到有错误的代码时，错误会像正常运行程序那样发生。在这个过程中，可以监视数据，看看什么地方出错。本章后面将使用这个技巧查看示例应用程序的执行情况。

#### 4. “即时” (Immediate)和“命令” (Command)窗口

“命令” (Command)【只有 VS 中有】和“即时” (Immediate)窗口【选择“调试”窗口 (Debug) 窗口 (Windows) 菜单】可以在运行应用程序的过程中执行命令。通过“命令” (Command)窗口可以手动执行 VS 操作(例如，菜单和工具栏操作)，“即时” (Immediate)窗口可以执行源代码，计算表达式，还可以执行其他代码。

VS 中的这些窗口在内部是链接在一起的(实际上，VS 的早期版本把它们当作同一个窗口)。甚至可以在它们之间切换：输入命令 `immed`，可以从“命令” (Command)窗口切换到“即时” (Immediate)窗口；输入 `>cmd` 可以从“即时” (Immediate)窗口切换到“命令” (Command)窗口。

下面详细讨论“即时” (Immediate)窗口，因为“命令” (Command)窗口仅适用于复杂的操作，只能在 VS 中使用，而“即时” (Immediate)窗口可以在 VS 和 VCE 中使用。“即时” (Immediate)窗口最简单的用法是计算表达式，有点像“监视” (Watch)窗口中的一次性使用。为此，只需键入一个表达式，并按回车键即可。接着就会显示请求的信息，如图 7-15 所示。

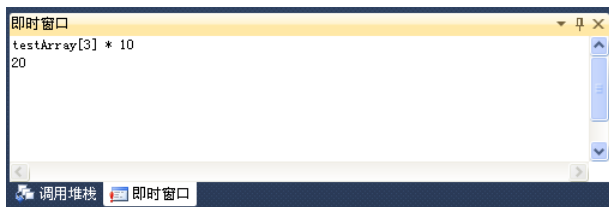


图 7-15

也可以在这里修改变量的内容，如图 7-16 所示。

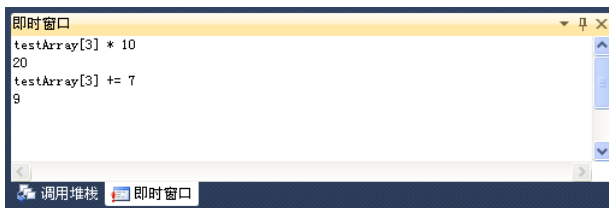


图 7-16

在大多数情况下，使用前面介绍的变量监视窗口更容易得到相同的效果，但这个技巧对于常常发生变化的变量值仍很方便，也适合于测试以后不感兴趣的表达式。

#### 5. “调用堆栈” (Call Stack)窗口

这是最后一个要讨论的窗口，它描述了程序是如何执行到当前位置的。简言之，该窗口显示了当前函数、调用它的函数以及调用函数的函数(即一个嵌套的函数调用列表)。调用的位置也被记录下来。

在前面的示例中，在执行到 `Maxima()` 时进入中断模式，或者使用代码单步执行功能移动到这个函数的内部，得到如图 7-17 所示的信息。

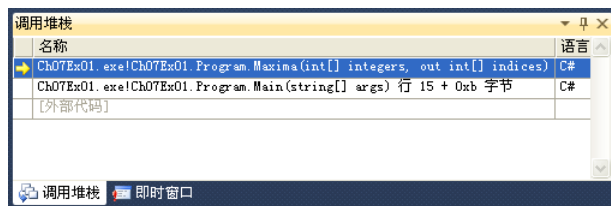


图 7-17

如果双击某一项，就会移动到相应的位置，跟踪代码执行到当前位置的过程。在第一次检测错误时，这个窗口非常有用，因为它们可以查看临近错误发生时的情况。对于常用函数中出现的错误，则有助于找到错误的源头。



有时“调用堆栈” (Call Stack) 窗口会显示一些非常杂乱的信息，例如，有时因为以错误的方式使用了外部函数，错误在应用程序的外部发生，就会出现这种情况。此时，这个窗口中会列出一个很长的列表，其中只有一个选项是我们熟悉的。如果右击该窗口，选择“显示外部代码” (Show External Code)，查看外部引用。

## 7.2 错误处理

本章的第一部分讨论如何在应用程序的开发过程中查找和改正错误，使这些错误不会在发布的代码中出现。但有时，我们知道可能会有错误发生，但不能 100% 地肯定它们不会发生。此时，最好能预料到错误的发生，编写足够强壮的代码以处理这些错误，而不必中断程序的执行。

错误处理就是用于这个目的，下面还将介绍异常和处理它们的方式。异常是在运行期间代码中产生的错误，或者由代码调用的函数产生的错误。这里的“错误”定义要比以前更含糊，因为异常可能是在函数等结构中手工产生。例如，如果函数的一个字符串参数不是以 `a` 开头，就产生一个异常。这并不是严格意义上的函数外部错误，但调用该函数的代码把它看作函数外部错误。

您已在本书前面已经遇到几次异常了。最简单的示例是试图定位一个超出范围的数组元素，例如：

```
int[] myArray = { 1, 2, 3, 4 };
int myElem = myArray[4];
```

这会产生如下异常信息，并中断应用程序的执行：

索引超出了数组界限。(Index was outside the bounds of the array.)



前面介绍了异常辅助信息窗口的一些示例。该窗口中的一行把它与出错的代码连接起来，还包含 .NET 帮助文件中相关主题的链接和一个“查看详细信息” (View Detail) 链接，利用该链接可以找到所发生异常的更多信息。

异常在命名空间中定义，大多数异常的名称清晰地说明了它们的用途。在这个示例中，产生的异常叫做 `System.IndexOutOfRangeException`，说明我们提供的 `myArray` 数组索引不在允许使用的索引范围内。在异常未处理时，这个信息才会显示出来，应用程序也才会中断执行。下一节将讨论如何处理异常。

### 7.2.1 try ... catch ... finally

C#语言包含结构化异常处理(Structured Exception Handling, SEH))的语法。用 3 个关键字可以标记能处理异常的代码和指令，如果发生异常，就使用这些指令处理异常。用于这个目的的 3 个关键字是 `try`、`catch` 和 `finally`。它们都有一个关联的代码块，必须在连续的代码行中使用。其基本结构如下：

```
try
{
    ...
}
catch (<exceptionType> e)
{
    ...
}
finally
{
    ...
}
```

也可以只有 `try` 块和 `finally` 块，而没有 `catch` 块，或者有一个 `try` 块和好几个 `catch` 块。如果有一个或多个 `catch` 块，`finally` 块就是可选的，否则就是必需的。这些代码块的用法如下：

- **try**——包含抛出异常的代码(在谈到异常时，其中的“抛出”在 C#语言中也可以是“生成”或“导致”)。
- **catch**——包含抛出异常时要执行的代码。`catch` 块可以使用 `<exceptionType>`，设置为只响应特定的异常类型(如 `System.IndexOutOfRangeException`)，以便提供多个 `catch` 块，还可以完全省略这个参数，让一般的 `catch` 块响应所有异常。
- **finally**——包含总是会执行的代码，如果没有产生异常，则在 `try` 块之后执行，如果处理了异常，就在 `catch` 块后执行，或者在未处理的异常上移到调用堆栈之前执行。“上移到调用堆栈”表示，SEH 允许嵌套 `try...catch...finally` 块，可以直接嵌套，也可以在 `try` 块包含的函数调用中嵌套。例如，如果在被调用的函数中没有 `catch` 块能处理某个异常，就由调用代码中的 `catch` 块处理。如果始终没有匹配的 `catch` 块，就终止应用程序。`finally` 块在此之前处理，是因为存在这个块，否则也可以在 `try...catch...finally` 结构的外部放置代码。这个嵌套功能将在后面的“异常处理的注意事项”一节中进一步讨论，所以如果对这个功能感到迷惑，请不必担心。

在 `try` 块的代码中出现异常后，发生的事件依次是：

- `try` 块在发生异常的地方中断程序的执行。
- 如果有 `catch` 块，就检查该块是否匹配已抛出的异常类型。如果没有 `catch` 块，就执行 `finally` 块(如果没有 `catch` 块，就一定要有 `finally` 块)。
- 如果有 `catch` 块，但它与已发生的异常类型不匹配，就检查是否有其他 `catch` 块。
- 如果有 `catch` 块匹配已发生的异常类型，就执行它包含的代码，再执行 `finally` 块(如果有)。

- 如果 catch 块都不匹配已发生的异常类型，就执行 finally 块(如果有)。

下面用一个“试一试”示例来说明异常的处理。这个示例以几种方式抛出和处理异常，以便读者了解其工作情况。

### 试一试：异常处理

(1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新控制台应用程序 Ch07Ex02。

(2) 修改代码，如下所示(这里显示的行号注释有助于将代码与后面的讨论相匹配，在本章的可下载代码中复制了它们，以便使用)：



可从  
wrox.com  
下载源代码

```
class Program
{
    static string[] eTypes = { "none", "simple", "index", "nested index" };

    static void Main(string[] args)
    {
        foreach (string eType in eTypes)
        {
            try
            {
                Console.WriteLine("Main() try block reached."); // Line 23
                Console.WriteLine("ThrowException(\"{0}\") called.", eType);
                                                                    // Line 24
                ThrowException(eType);
                Console.WriteLine("Main() try block continues."); // Line 26
            }
            catch (System.IndexOutOfRangeException e)                // Line 28
            {
                Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                                + " block reached. Message:\n\"{0}\",
                                e.Message);

            }
            catch                                                    // Line 34
            {
                Console.WriteLine("Main() general catch block reached.");
            }
            finally
            {
                Console.WriteLine("Main() finally block reached.");
            }
            Console.WriteLine();
        }
        Console.ReadKey();
    }

    static void ThrowException(string exceptionType)
    {
                                                                    // Line 49
        Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
        switch (exceptionType)
        {
            case "none":
                Console.WriteLine("Not throwing an exception.");
        }
    }
}
```

```

        break; // Line 54
    case "simple":
        Console.WriteLine("Throwing System.Exception.");
        throw (new System.Exception()); // Line 57
    case "index":
        Console.WriteLine("Throwing System.IndexOutOfRangeException.");
        eTypes[4] = "error"; // Line 60
        break;
    case "nested index":
        try // Line 63
        {
            Console.WriteLine("ThrowException(\"nested index\") " +
                "try block reached.");
            Console.WriteLine("ThrowException(\"index\") called.");
            ThrowException("index"); // Line 68
        }
        catch // Line 70
        {
            Console.WriteLine("ThrowException(\"nested index\") general"
                + " catch block reached.");
        }
        finally
        {
            Console.WriteLine("ThrowException(\"nested index\") finally"
                + " block reached.");
        }
        break;
    }
}
}

```

代码段 Ch07Ex02\Program.cs

(3) 运行应用程序，结果如图 7-18 所示。

```

file:///C:/BegYCSHarp/Chapter07/Ch07Ex02/Ch07Ex02/bin/Debug/Ch07Ex02.EXE
Main> try block reached.
Main> ThrowException("none") called.
Main> ThrowException("none") reached.
Main> Not throwing an exception.
Main> try block continues.
Main> finally block reached.

Main> try block reached.
Main> ThrowException("simple") called.
Main> ThrowException("simple") reached.
Main> Throwing System.Exception.
Main> general catch block reached.
Main> finally block reached.

Main> try block reached.
Main> ThrowException("index") called.
Main> ThrowException("index") reached.
Main> Throwing System.IndexOutOfRangeException.
Main> System.IndexOutOfRangeException catch block reached. Message:
"索引超出了数组界限。"
Main> finally block reached.

Main> try block reached.
Main> ThrowException("nested index") called.
Main> ThrowException("nested index") reached.
Main> ThrowException("nested index") try block reached.
Main> ThrowException("index") called.
Main> ThrowException("index") reached.
Main> Throwing System.IndexOutOfRangeException.
Main> ThrowException("nested index") general catch block reached.
Main> ThrowException("nested index") finally block reached.
Main> try block continues.
Main> finally block reached.

搜狗拼音 半:

```

图 7-18

### 示例的说明

这个应用程序在 Main() 中有一个 try 块，它调用函数 ThrowException()。这个函数会根据调用时使用的参数抛出异常：

- ThrowException("none")——不抛出异常。
- ThrowException("simple")——生成一般异常。
- ThrowException("index")——生成 System.IndexOutOfRangeException 异常。
- ThrowException("nested index")——包含它自己的 try 块，其中的代码调用 ThrowException("index") 生成一个 System.IndexOutOfRangeException 异常。

其中的每个 string 参数都存储在全局数组 eTypes 中，在 Main() 函数中迭代，用每个可能的参数调用 ThrowException()。在迭代过程中，会把各种信息写到控制台上，说明发生了什么情况。这段代码可以使用本章前面介绍的代码单步执行技巧。在执行代码的过程中，一次执行一行代码可以确切地了解代码的执行进度。

在代码的第 23 行添加一个新断点(用默认的属性)，该行代码如下：

```
Console.WriteLine("Main() try block reached.");
```



这里使用了行号，因为它们显示在这段代码的下载版本中。如果关闭了行号，可以通过“工具 ⇨ 选项” (Tools ⇨ Options) 菜单项打开“选项” (Options) 对话框，在“文本编辑器 ⇨ C# ⇨ 常规” (Text Editor ⇨ C# ⇨ General) 选项区域中，选择打开它们。上述代码包含注释，这样读者可以阅读文本，而无需打开文件。

在调试模式下运行应用程序。程序立即进入中断模式，此时光标停在第 23 行上。如果选择变量监视窗口中的“局部变量” (Locals) 选项卡，就会有到 eType 当前是 none。使用“逐语句” (Step Into) 按钮处理第 23 和 24 行，看看第一行文本是否已经写到控制台上。接着使用“逐语句” (Step Into) 按钮单步执行第 25 行的 ThrowException() 函数。

执行到 ThrowException() 函数(第 49 行)后，“局部变量” (Locals) 窗口会发生变化。eType 和 args 不再能访问(因为它们是 Main() 的局部变量)，我们看到的是 exceptionType 局部参数，它当然是 none。继续单击“逐语句” (Step Into)，到达 switch 语句，检查 exceptionType 的值，执行代码，把字符串 Not throwing an exception 写到屏幕上。在执行第 54 行上的 break 语句时，将退出函数，继续处理 Main() 中的第 26 行代码。因为没有抛出异常，所以继续执行 try 块。

接着处理 finally 块。再单击“逐语句” (Step Into) 几次，执行完 finally 块和 foreach 的第一次循环。下次执行到第 25 行时，使用另一个参数 simple 调用 ThrowException()。

继续使用“逐语句” (Step Into) 单步执行 ThrowException()，最终会执行到第 57 行：

```
throw (new System.Exception());
```

这里使用 C# throw 关键字生成一个异常，需要为这个关键字提供新初始化的异常作为其参数，抛出一个异常，这里使用名称空间 System 中的另一个异常 System.Exception。



在这个 case 块中不需要 break 语句，使用 throw 就可以结束该块的执行。

在使用“逐语句”(Step Into)执行这个语句时,将从第34行开始执行一般的 catch 块。因为与第28行开始的 catch 块都不匹配,所以执行这个一般的 catch 块。单步执行这段代码,前后执行 finally 块,最后返回另一个循环周期,该循环在第25行用一个新参数调用 ThrowException(),这次的参数是 index。

这次 ThrowException()在第60行生成一个异常:

```
eTypes[4] = "error";
```

eTypes 是一个全局数组,所以可以在这里访问它。但是这里试图访问数组中的第5个元素(其索引从0开始计数),这会生成一个 System.IndexOutOfRangeException 异常。

这次 Main()中有一个匹配的 catch 块,单步执行该 catch 块,从第28行开始。这个块中调用的 Console.WriteLine()使用 e.Message, 输出存储在异常中的信息(可以通过 catch 块的参数访问异常)。之后再次单步执行 finally 块(而不是第二个 catch 块,因为异常已经处理完了)。返回循环,再次调用第25行的 ThrowException()。

在执行到 ThrowException()中的 switch 结构时,进入一个新的 try 块,从第63行开始。在执行到第68行时,将遇到 ThrowException()的一个嵌套调用,这次使用 index 参数。可以使用“逐过程”(Step Over)按钮跳过其中的代码行,因为前面已经单步执行过了。与前面一样,这个调用生成一个 System.IndexOutOfRangeException 异常。但这个异常在 ThrowException()中的嵌套 try...catch...finally 结构中处理。这个结构没有明确匹配这种异常的 catch 块,所以执行一般的 catch 块(从第70行开始)。

与前面的异常处理一样,现在单步执行这个 catch 块,以及关联的 finally 块,最后返回到函数调用的末尾。但是它们有一个重大的区别:抛出的异常是由 ThrowException()中的代码处理的。这就是说,异常并没有留给 Main()处理,所以直接进入 finally 块,之后应用程序中断执行。

## 7.2.2 列出和配置异常

.NET Framework 包含许多异常类型,可以在代码中自由抛出和处理这些类型的异常,甚至可以在代码中抛出异常,让它们在比较复杂的应用程序中被捕获。IDE 提供了一个对话框,可以检查和编辑可用的异常,可以使用“调试↔异常”(Debug ↔ Exceptions)菜单选项(或按下 Ctrl+D, E)打开该对话框,如图 7-19 所示(如果使用 VCE,则列表中的项会不同,只包含图 7-19 中的第二、三项)。

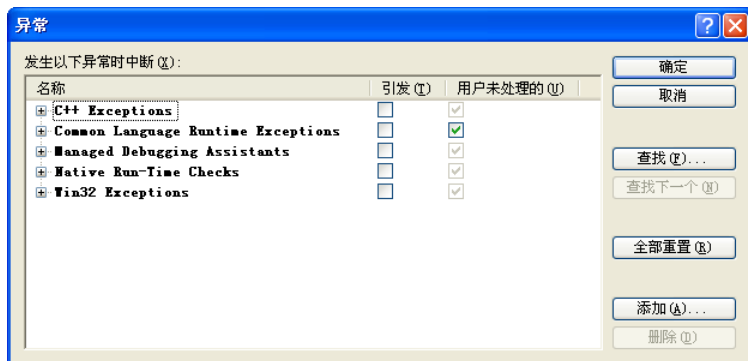


图 7-19

按照类别和.NET 库名称空间列出异常。展开 Common Language Runtime Exceptions 选项,再展开 System 选项,就可以看到 System 名称空间中的异常,这个列表包括上面使用的异常

System.IndexOutOfRangeException。

每个异常都可以使用右边的复选框来配置。可以使用第一个选项“引发”(break when)Thrown 中断调试器，即使是对于已处理的异常，也是这样。第二个选项可以忽略未处理的异常，这样做会对结果有影响。在大多数情况下，这会进入中断模式，所以只需在异常环境下这么做。

在大多数情况下，使用默认设置就足够了。

### 7.2.3 异常处理的注意事项

注意，必须在更一般的异常捕获之前为比较特殊的异常提供 catch 块。如果 catch 块的顺序错误，应用程序就会编译失败。还要注意可以在 catch 块中抛出异常，方法是使用上一个示例中的方式，或使用下述表达式：

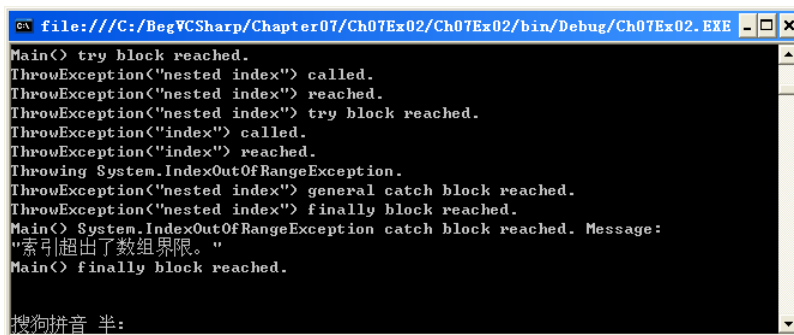
```
throw;
```

这个表达式会再次抛出 catch 块处理过的异常。如果以这种方式抛出异常，该异常就不会由当前的 try...catch...finally 块处理，而是由上一级的代码处理(但嵌套结构中的 finally 块仍会执行)。

例如，如果修改了 ThrowException() 中的 try...catch...finally 块，如下所示：

```
try
{
    Console.WriteLine("ThrowException(\"nested index\") " +
        "try block reached.");
    Console.WriteLine("ThrowException(\"index\") called.");
    ThrowException("index");
}
catch
{
    Console.WriteLine("ThrowException(\"nested index\") general"
        + " catch block reached.");
    throw;
}
finally
{
    Console.WriteLine("ThrowException(\"nested index\") finally"
        + " block reached.");
}
```

则首先执行其中的 finally 块，再执行 Main() 中匹配的 catch 块，得到的控制台输出如图 7-20 所示。



```
file:///C:/BegVCSharp/Chapter07/Ch07Ex02/Ch07Ex02/bin/Debug/Ch07Ex02.EXE
Main(>) try block reached.
ThrowException("nested index") called.
ThrowException("nested index") reached.
ThrowException("nested index") try block reached.
ThrowException("index") called.
ThrowException("index") reached.
Throwing System.IndexOutOfRangeException.
ThrowException("nested index") general catch block reached.
ThrowException("nested index") finally block reached.
Main(>) System.IndexOutOfRangeException catch block reached. Message:
"索引超出了数组界限。"
Main(>) finally block reached.
搜狗拼音 半:
```

图 7-20



在这个屏幕图中，Main() 函数输出了额外的几行，因为这个函数捕获了 System.IndexOutOfRangeException 异常。

## 7.3 小结

本章主要论述了调试应用程序所使用的技巧。有许多这方面的技巧，其中的大部分可用于各类项目，而不仅仅是控制台应用程序。

前面介绍了创建简单控制台应用程序的所有内容，以及调试它们的各种方法。本书的下一部分将讨论强大的面向对象编程技术。

## 7.4 练习

(1) “使用 Trace.WriteLine() 要优于 Debug.WriteLine()，因为调试版本仅能用于调试程序。”这个观点正确吗？为什么？

(2) 为一个简单的应用程序编写代码，其中包含一个循环，该循环在运行 5000 次后产生一个错误。使用断点在第 5000 次循环出现错误前进入中断模式(注意产生错误的一种简单方式是试图访问一个不存在的数组元素，例如在一个有 100 个元素的数组中，访问 myArray[100])。

(3) “只有在不执行 catch 块的情况下，才执行 finally 代码块”，对吗？

(4) 下面定义了一个枚举数据类型 orientation。编写一个应用程序，使用结构化异常处理(SEH)把 byte 类型的变量安全地强制转换为 orientation 类型变量。注意，可以使用 checked 关键字强制抛出异常，下面是一个示例。可以在应用程序中使用这段代码：

```
enum Orientation : byte
{
    North = 1,
    South = 2,
    East = 3,
    West = 4
}

myDirection = checked((Orientation)myByte);
```

附录 A 给出了练习答案。

## 7.5 本章要点

主 题	重 要 概 念
错误类型	在编译期间和运行期间，致命错误(语法错误)都会使应用程序完全失败，语义错误或逻辑错误比较微妙，可能会使应用程序执行不正确，或者以未预料到的方式执行
输出调试信息	我们可以编写代码，把有帮助的信息输出到“输出”(Output)窗口中，以帮助在 IDE 中进行调试。为此需要使用 Debug 和 Trace 系列函数，其中 Debug 函数在发布版本中会被忽略。对于投入生产的应用程序，应把调试输出写入日志文件。在 VS 中，还可以使用跟踪点输出调试信息

(续表)

主 题	重 要 概 念
中断模式	可以通过断点、判定语句, 或者发生未处理的异常时, 手工进入中断模式(暂停应用程序的状态)。可以在代码的任意位置添加断点, 在 VS 中, 还可以把断点配置为仅在特定条件下中断执行。在中断模式下, 可以检查变量的内容(使用各种调试信息窗口), 每次执行一行代码, 以帮助确定哪里出现了错误
异常	异常是运行期间发生的错误, 可以通过编程方式捕获和处理这种错误, 以防应用程序终止。调用函数或处理变量时, 可能会发生许多不同类型的异常。还可以使用 <code>throw</code> 关键字生成异常
异常处理	代码中未处理的异常会使应用程序终止。使用 <code>try</code> 、 <code>catch</code> 和 <code>finally</code> 代码块处理异常。 <code>try</code> 块标记了一个启用异常处理的代码断, <code>catch</code> 块包含的代码仅在异常发生时执行, 它可以匹配特定类型的异常, 还可以包含多个 <code>catch</code> 块。 <code>finally</code> 块指定异常处理完毕后执行的代码, 如果没有发生异常, <code>finally</code> 块就指定在 <code>try</code> 块执行完毕后执行的代码。只能包含一个 <code>finally</code> 块, 如果包含了 <code>catch</code> 块, <code>finally</code> 块就是可选的