

```
try
{
    string userInput;

    Console.Write("Input a number between 0 and 5," +
        "(or just hit return to exit)> ");
    userInput = Console.ReadLine();

    if (userInput == "")
    {
        break;
    }

    int index = Convert.ToInt32(userInput);

    if (index < 0 || index > 5)
    {
        throw new IndexOutOfRangeException(
            "You typed in " + userInput);
    }

    Console.WriteLine("Your number was " + index);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Exception: " +
        "Number should be between 0 and 5. {0}", ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine(
        "An exception was thrown. Message was: {0}", ex.Message);
}
finally
{
    Console.WriteLine("Thank you");
}
```

代码段 SimpleExceptions.cs

这段代码的核心是一个 `while` 循环，它连续使用 `Console.ReadLine()` 方法以请求用户输入。`ReadLine()` 方法返回一个字符串，所以程序首先要用 `System.Convert.ToInt32()` 方法把它转换为 `int` 型。`System.Convert` 类包含执行数据转换的各种有用方法，并提供了 `int.Parse()` 方法的一个替代方法。一般情况下，`System.Convert` 类包含执行各种类型转换的方法，C# 编译器把 `int` 解析为 `System.Int32` 基类的实例。



值得注意的是，传递给 `catch` 块的参数只能用于该 `catch` 块。这就是为什么在上面的代码中，能在后续的 `catch` 块中使用相同的参数名 `ex` 的原因。

在上面的代码中，我们也检查一个空字符串，因为该空字符串是退出 `while` 循环的条件。注意这里用 `break` 语句退出 `try` 块和 `while` 循环——这是有效的。当然，当程序流退出 `try` 块时，会执行 `finally` 块中的 `Console.WriteLine()` 语句。尽管这里仅显示一句问候，仍需要关闭文件句柄，调用各种对象的 `Dispose()` 方法，以执行清理工作。一旦计算机退出了 `finally` 块，它就会继续执行下一条语句，如果没有 `finally` 块，该语句也会执行。在本例中，我们返回 `while` 循环的开头，再次进入 `try` 块(除非执行 `while` 循环中 `break` 语句的结果是进入 `finally` 块，此时就会退出 `while` 循环)。

下面看看异常情况：

```
if (index < 0 || index > 5)
{
    throw new IndexOutOfRangeException("You typed in " + userInput);
}
```

在抛出一个异常时，需要选择要抛出的异常类型。可以使用 `System.Exception` 异常类，但这个类是一个基类，最好不要把这个类的实例当作一个异常抛出，因为它没有包含关于错误的任何信息。而 .NET Framework 包含了许多派生自 `System.Exception` 异常类的其他异常类，每个类都对应于一种特定类型的异常情况，也可以定义自己的异常类。在抛出一个匹配特定错误情况的类的实例时，应提供尽可能多的异常信息。在本例中，`System.IndexOutOfRangeException` 异常类是最佳选择。`IndexOutOfRangeException` 异常类有几个重载的构造函数，我们选择的一个重载，其参数是一个描述错误的字符串。另外，也可以选择派生自己的自定义异常对象，它描述该应用程序环境中的错误情况。

假定用户这次输入了一个不在 0~5 范围内的数字，`if` 语句就会检测到一个错误，并实例化和抛出一个 `IndexOutOfRangeException` 异常对象。计算机将立即退出 `try` 块，并查找处理 `IndexOutOfRangeException` 异常的 `catch` 块。它遇到的第一个 `catch` 块如下所示：

```
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(
        "Exception: Number should be between 0 and 5. {0}", ex.Message);
}
```

由于这个 `catch` 块带合适类的一个参数，因此它就会传递给异常实例，并执行。在本例中，是显示错误信息和 `Exception.Message` 属性(它对应于给 `IndexOutOfRangeException` 异常类的构造函数传递的字符串)。执行了这个 `catch` 块后，控制权就切换到 `finally` 块，就好像没有发生过任何异常。

注意，本例还提供了另一个 `catch` 块：

```
catch (Exception ex)
{
    Console.WriteLine("An exception was thrown. Message was: {0}", ex.Message);
}
```

如果没有在前面的 `catch` 块中捕获到这类异常，则这个 `catch` 块也能处理 `IndexOutOfRangeException` 异常。基类的一个引用也可以指派来自它的类的所有实例，所有的异常都派生自 `System.Exception` 异常类。那么为什么不执行这个 `catch` 块？答案是计算机只执行它在可用的 `catch` 块列表中的第一个合适的 `catch` 块。但为什么还要编写第二个 `catch` 块？不仅 `try` 块包含这段代码，还有另外 3 个方法调用 `Console.ReadLine()`、`Console.Write()` 和 `Convert.ToInt32()` 也包含这段代码，它们是 `System` 名

称空间中的方法。这 3 个方法都可能抛出异常。

如果输入的内容不是数字,如 a 或 hello,Convert.ToInt32()方法就会抛出 System.FormatException 类的一个异常,表示传递给 ToInt32()方法的字符串对应的格式不能转换为 int。此时,计算机跟踪这个方法调用,查找可以处理该异常的处理程序。第一个 catch 块带一个 IndexOutOfRangeException 异常,不能处理这种异常。计算机接着查看第二个 catch 块,显然它可以处理这类异常,因为 FormatException 异常类派生于 Exception 异常类,所以把 FormatException 异常类的实例作为参数传递给它。

该示例的这种结构是非常典型的多 catch 块结构。最先编写的 catch 块用于处理非常特殊的错误情况,接着是比较一般的块,它们可以处理任何错误,我们没有为它们编写特定的错误处理程序。实际上,catch 块的顺序很重要,如果以相反的顺序编写这两个块,代码就不会编译,因为第二个 catch 块是不会执行的(Exception catch 块会捕获所有异常)。因此,最上面的 catch 块应用于最特殊的异常情况,最后是最一般的 catch 块。

前面分析了该示例的代码,现在可以运行它。下面的输出说明了不同的输入会得到不同的结果,并说明抛出了 IndexOutOfRangeException 异常和 FormatException 异常:

```
SimpleExceptions
Input a number between 0 and 5 (or just hit return to exit)>4
Your number was 4
Thank you
Input a number between 0 and 5 (or just hit return to exit)>0
Your number was 0
Thank you
Input a number between 0 and 5 (or just hit return to exit)>10
Exception: Number should be between 0 and 5. You typed in 10
Thank you
Input a number between 0 and 5 (or just hit return to exit)> hello
An exception was thrown. Message was: Input string was not in a correct format.
Thank you
Input a number between 0 and 5 (or just hit return to exit)>
Thank you
```

15.2.2 在其他代码中捕获异常

上面的示例说明了两个异常的处理。一个是 IndexOutOfRangeException 异常,它由我们自己的代码抛出,另一个是 FormatException 异常,它由一个基类抛出。如果检测到错误,或者某个方法因传递的参数有误而被错误调用,库中的代码就常常会抛出一个异常。但库中的代码很少捕获这样的异常。应由客户端代码来决定如何处理这些问题。

在调试时,异常经常从基类库中抛出,调试的过程在某种程度上是确定异常抛出的原因,并消除导致错误发生的缘由。主要目标是确保代码在发布后,异常只发生在非常罕见的情况下,如果可能,应在代码中以适当的方式处理它。

15.2.3 System.Exception 属性

本示例只使用了异常对象的一个 Message 属性。在 System.Exception 异常类中还有许多其他属性,如表 15-1 所示。

表 15-1

属 性	说 明
Data	这个属性可以给异常添加键/值语句, 以提供关于异常的额外信息
HelpLink	链接到一个帮助文件上, 以提供关于该异常的更多信息
InnerException	如果此异常是在 catch 块中抛出的, 它就会包含把代码发送到 catch 块中的异常对象
Message	描述错误情况的文本
Source	导致异常的应用程序名或对象名
StackTrace	栈上方法调用的详细信息, 它有助于跟踪抛出异常的方法
TargetSite	描述抛出异常的方法的 .NET 反射对象

在这些属性中, 如果可以进行栈跟踪, `StackTrace` 和 `TargetSite` 属性由 .NET 运行库自动提供。`Source` 属性总是由 .NET 运行库填充为抛出异常的程序集的名称(但可以在代码中修改该属性, 提供更专门的信息), `Data`、`Message`、`HelpLink` 和 `InnerException` 属性必须在抛出异常的代码中填充, 方法是在抛出异常前设置这些属性。例如, 抛出异常的代码如下所示:

```

if (ErrorCondition == true)
{
    Exception myException = new ClassMyException("Help!!!!");
    myException.Source = "My Application Name";
    myException.HelpLink = "MyHelpFile.txt";
    myException.Data["ErrorDate"] = DateTime.Now;
    myException.Data.Add("AdditionalInfo", "Contact Bill from the Blue Team");
    throw myException;
}

```

其中 `ClassMyException` 是抛出的异常类的名称。注意所有异常类的名称通常以 `Exception` 结尾。`Data` 属性可以用两种方式设置。

15.2.4 没有处理异常时所发生的情况

有时抛出了一个异常后, 代码中没有 `catch` 块能处理这类异常。前面的 `SimpleExceptions` 示例就说明了这种情况。例如, 假定忽略 `FormatException` 异常和通用的 `catch` 块, 只有捕获 `IndexOutOfRangeException` 异常的块。此时, 如果抛出一个 `FormatException` 异常, 会发生什么情况呢?

答案是 .NET 运行库会捕获它。本节后面将介绍如何嵌套 `try` 块——实际上在本示例中, 就有一个在后台处理的嵌套的 `try` 块。 .NET 运行库把整个程序放在另一个更大的 `try` 块中, 对于每个 .NET 程序它都会这么做。这个 `try` 块有一个 `catch` 处理程序, 它可以捕获任何类型的异常。如果出现代码没有处理的异常, 程序流就会退出程序, 由 .NET 运行库中的 `catch` 块捕获它。但是, 事与愿违。代码的执行会立即终止, 并给用户显示一个对话框, 说明代码没有处理异常, 并给出 .NET 运行库能检索到的关于异常的详细信息。至少异常会被捕获, 这就是第 2 章在 `Vector` 示例程序抛出一个异常时发生的情况。

一般情况下, 如果编写一个可执行程序, 就应捕获尽可能多的异常, 并以合理的方式处理它们。如果编写一个库, 最好不要捕获异常(除非某个特殊异常表示在代码中可以处理的情况), 但要假定调用代码可以处理它们。当然, 用户可能不想捕获 Microsoft 预定义的任何异常, 而是抛出自己的异

常对象，该异常对象给客户端代码提供更特定的信息。

15.2.5 嵌套的 try 块

异常的一个特性是 try 块可以嵌套，如下所示：

```
try
{
    // Point A
    try
    {
        // Point B
    }
    catch
    {
        // Point C
    }
    finally
    {
        // clean up
    }
    // Point D
}
catch
{
    // error handling
}
finally
{
    // clean up
}
```

在上面的代码中，每个 try 块都只有一个 catch 块，但可以把多个 catch 块连接在一起。下面详细讨论嵌套的 try 块如何工作。

如果抛出的异常在外层的 try 块中，但在内层 try 块的外部(标记为 A 点和 D 点的代码块)，这种情况就与前面介绍的情况没有任何区别：异常由外层的 catch 块捕获，并执行外层的 finally 块，或者执行 finally 块，由 .NET 运行库处理异常。

如果异常是在内层 try 块(标记为 B 点的代码块)中抛出的，且有一个合适的内层 catch 块处理该异常，这又是我们熟悉的情况：在内层处理异常，执行内层的 finally 块，之后继续执行外层的 try 块(标记为 D 点的代码块)。

现在假定异常是在代码块的内层 try 块中抛出的，但内层的 catch 块中没有合适的处理程序。这时会像通常一样执行内层的 finally 块，但接着 .NET 运行库必须退出内层的 try 块，才能搜索到合适的处理程序。下一个要搜索的区域显然是外层的 catch 块。如果系统在这里找到一个处理程序，就会执行该处理程序，再执行外层的 finally 块。如果没有找到合适的处理程序，就会继续搜索。在这里，执行的是外层的 finally 块，因为没有更多的 catch 块，所以控制权会转移到 .NET 运行库。注意，任何时候不会执行外层 try 块中 D 点后面的代码。

如果异常是在 C 点抛出的，就更有趣了。如果程序执行到 C 点中，它就必须处理在 B 点抛出的异常。在 catch 块中抛出另一个异常很正常。此时，异常的处理就跟它是在外层 try 块中抛出的一

样, 程序流会立即退出内层的 `catch` 块, 执行内层的 `finally` 块, 系统在外层的 `catch` 块中搜索处理程序。同样, 如果在内层的 `finally` 块中抛出一个异常, 搜索会在外层的 `catch` 块开始, 控制权会立即转移到最匹配的处理程序。



在 `catch` 块和 `finally` 块中抛出异常是完全合理的。

尽管本例只有两个 `try` 块, 但无论嵌套了多少个 `try` 块, 规则都是一样的。在每个阶段中, .NET 运行库顺序执行 `try` 块, 查找合适的处理程序。在每个阶段中, 当退出 `catch` 块后, 就会执行对应 `finally` 块中的任何清理代码, 但不执行 `finally` 块外部的代码, 直到找到合适的 `catch` 处理程序, 并执行为止。

`try` 块的嵌套也可能发生在方法之间。例如, 如果方法 A 调用了 `try` 块中的方法 B, 那么方法 B 也包含一个 `try` 块。

前面说明了嵌套的 `try` 块的工作方式。下一个问题显然是为什么要这么做? 这有两个原因:

- 为了修改所抛出的异常的类型。
- 为了能够在代码的不同地方处理不同类型的异常。

1. 修改异常的类型

当最初抛出的异常不足以说明问题时, 修改异常的类型就非常重要。通常的情况是抛出的异常(可能由 .NET 运行库抛出)是一种相当低级的异常, 说明发生溢出(`OverflowException` 异常)或传递给方法的参数不正确(派生于 `ArgumentException` 异常类的一个类)。但是, 由于抛出异常的环境, 我们知道这暴露了一些底层的问题(例如, 因为刚才读取的文件包含了不正确的数据, 才发生了溢出异常)。此时处理程序对于第一个异常所能做的最佳处理就是抛出另一个异常, 以便更准确地说明这个问题, 从而让另一个 `catch` 块更恰当地处理它。也可以通过一个由 `System.Exception` 异常类实现的 `InnerException` 属性来处理最初的异常。`InnerException` 属性只包含另一个抛出的相关异常的引用——最终的处理程序例程需要这些额外信息。

当然, 还应指出, 异常可能在 `catch` 块中抛出。例如, 可以从某个配置文件中读取数据, 这个文件包含处理错误的详细指令——结果可能是这个文件不存在。

2. 在不同的地方处理不同的异常

嵌套 `try` 块的第二个原因是不同类型的异常可以在代码的不同地方处理。例如, 在循环中, 可能会发生各种异常。其中一些异常比较严重, 需要退出整个循环, 而另外一些则不太严重, 只需退出这次迭代, 进入循环的下一次迭代即可。在循环的内部有一个 `try` 块就可以处理不太严重的异常, 再在循环外面用一个外层的 `try` 块来处理比较严重的错误。在下面的异常示例中, 将解释具体的操作情况。

15.3 用户定义的异常类

下面介绍有关异常的第二个示例, 这个示例叫作 `SolicitColdCall`, 它包含两个嵌套的 `try` 块, 说

明了如何定义自定义异常类，再从 `try` 块中抛出另一个异常。

这个示例假定一家销售公司希望有更多的客户。该公司的销售部门打算给一些人打电话，希望他们成为自己的客户。用销售行业的行话来讲，就是“陌生电话”。为此，应有一个文本文件存储这些陌生人的姓名，该文件应有良好的格式，其中第一行包含文件中的人数，后面的行包含这些人的姓名。换言之，正确的格式如下所示。

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

这个示例的目的是在屏幕上显示这些人的姓名(由销售人员读取)，这就是为什么只把姓名放在文件中，但没有电话号码的原因。

程序要求用户输入文件的名称，然后读取文件，并显示其中的人名。这听起来是一个很简单的任务，但也会出现两个错误，需要退出整个过程：

- 用户可能输入不存在的文件名。这作为 `FileNotFoundException` 异常来捕获。
- 文件的格式可能不正确，这里可能有两个问题。首先，文件的第一行不是整数。第二，文件中可能没有第一行指定的那么多人名。这两种情况都需要在一个自定义异常中处理，我们已经专门为此编写了 `ColdCallFileFormatException` 异常。

还会有其他问题，虽然不致于退出整个过程，但需要删除某个人名，继续处理文件中的下一个人名(因此这需要在内层的 `try` 块中处理)。一些人是工业间谍，为销售公司的竞争对手工作，显然，我们不希望让这些通过一个偶然的电话知道我们要做的工作。因此应确定哪些人是工业间谍，搜索条件是他们的姓名以 B 开头。这些人应在第一次准备数据文件时从文件中删除，但万一使工业间谍混入，就需要检查文件中的每个姓名，如果检测到一个工业间谍，就应抛出一个 `SalesSpyFoundException` 异常，当然，这是另一个自定义异常对象。

最后，编写一个类 `ColdCallFileReader` 来实现这个示例，该类维护与 `cold-call` 文件的连接，并从中检索数据。我们将以非常安全的方式编写这个类，如果其方法调用不正确，就会抛出异常。例如，如果在文件打开前，调用了读取文件的方法，就会抛出一个异常。为此，我们编写了另一个异常类 `UnexpectedException`。

15.3.1 捕获用户定义的异常

首先是 `SolicitColdCall` 示例的 `Main()` 方法，它捕获用户定义的异常。注意，下面要调用 `System.IO` 名称空间和 `System` 名称空间中的文件处理类。

```
using System;
using System.IO;

namespace Wrox.ProCSharp.AdvancedCSharp
{
    class MainEntryPoint
    {
        static void Main()
        {
            Console.Write("Please type in the name of the file " +
```

```

        "containing the names of the people to be cold called > ");
string fileName = Console.ReadLine();
ColdCallFileReader peopleToRing = new ColdCallFileReader();

try
{
    peopleToRing.Open(fileName);
    for (int i=0; i<peopleToRing.NPeopleToRing; i++)
    {
        peopleToRing.ProcessNextPerson();
    }
    Console.WriteLine("All callers processed correctly");
}
catch(FileNotFoundException)
{
    Console.WriteLine("The file {0} does not exist", fileName);
}
catch(ColdCallFileFormatException ex)
{
    Console.WriteLine(
        "The file {0} appears to have been corrupted", fileName);
    Console.WriteLine("Details of problem are: {0}", ex.Message);
    if (ex.InnerException != null)
    {
        Console.WriteLine(
            "Inner exception was: {0}", ex.InnerException.Message);
    }
}
catch(Exception ex)
{
    Console.WriteLine("Exception occurred:\n" + ex.Message);
}
finally
{
    peopleToRing.Dispose();
}
Console.ReadLine();
}

```

这段代码基本上只是一个循环，用来处理文件中的人名。开始时，先让用户输入文件名，再实例化 `ColdCallFileReader` 类的一个对象，这个类稍后定义，正是这个类负责处理文件中数据的读取。注意是在第一个 `try` 块的外部读取文件——这是因为这里实例化的变量需要在后面的 `catch` 块和 `finally` 块中使用，如果在 `try` 块中声明它们，它们在 `try` 块的闭合花括号处就超出了作用域，这会导致异常。

在 `try` 块中打开文件(使用 `ColdCallFileReader.Open()` 方法)，并循环处理其中的所有人名。`ColdCallFileReader.ProcessNextPerson()` 方法会读取并显示文件中的下一个人名，而 `ColdCallFileReader.NpeopleToRing` 属性则说明文件中应有多少个人名(通过读取文件的第一行来获得)。有 3 个 `catch` 块，其中两个分别用于处理 `FileNotFoundException` 和 `ColdCallFileFormatException` 异常，第 3 个则用于处理任何其他 .NET 异常。

在 `FileNotFoundException` 异常中，我们会为它显示一条消息，注意在这个 `catch` 块中，根本不

会使用异常实例，原因是这个 `catch` 块用于说明应用程序的用户友好性。异常对象一般会包含技术信息，这些技术信息对开发人员很有用，但对于最终用户来说则没有什么用，所以本例将创建一条更简单的消息。

对于 `ColdCallFormatException` 异常的处理程序，则执行相反的操作，说明了如何提供更完整的技术信息，包括内层异常的细节(如果存在内层异常)。

最后，如果捕获到其他一般异常，就显示一条用户友好消息，而不是让这些异常由 .NET 运行库处理。注意我们选择不处理派生自 `System.Exception` 异常类的某些异常，因为不直接调用非 .NET 的代码。

`finally` 块清理资源。在本例中，这是指关闭已打开的任何文件。`ColdCallFileReader.Dispose()` 方法完成了这个任务。

15.3.2 抛出用户定义的异常

下面看看处理文件读取，以及(可能)抛出用户定义的异常类 `ColdCallFileReader` 的定义。因为这个类维护一个外部文件连接，所以需要确保它根据第 4 章有关释放对象的规则，正确地释放它。这个类派生自 `IDisposable` 类。

首先声明一些变量：

```
class ColdCallFileReader: IDisposable
{
    FileStream fs;
    StreamReader sr;
    uint nPeopleToRing;
    bool isDisposed = false;
    bool isOpen = false;
```

`FileStream` 和 `StreamReader` 都在 `System.IO` 名称空间中，它们都是用于读取文件的基类。`FileStream` 基类主要用于连接文件，`StreamReader` 基类则专门用于读取文本文件，并实现 `StreamReader()` 方法，该方法读取文件中的一行文本。第 29 章在深入讨论文件处理时将讨论 `StreamReader` 基类。

`isDisposed` 字段表示是否调用了 `Dispose()` 方法，我们选择实现 `ColdCallFileReader` 异常，这样，一旦调用了 `Dispose()` 方法，就不能重新打开文件连接，重新使用对象了。`isOpen` 字段也用于错误检查——在本例中，检查 `StreamReader` 基类是否连接到打开的文件上。

打开文件和读取第一行的过程——告诉我们文件中有多少个人名——由 `Open()` 方法来处理：



可从
wrox.com
下载源代码

```
public void Open(string fileName)
{
    if (isDisposed)
        throw new ObjectDisposedException("peopleToRing");

    fs = new FileStream(fileName, FileMode.Open);
    sr = new StreamReader(fs);

    try
    {
        string firstLine = sr.ReadLine();
        nPeopleToRing = uint.Parse(firstLine);
        isOpen = true;
```

```

    }
    catch (FormatException ex)
    {
        throw new ColdCallFileFormatException(
            "First line isn't an integer", ex);
    }
}

```

代码段 SolicitColdCall.cs

与 `ColdCallFileReader` 异常类的所有其他方法一样，该方法首先检查在删除对象后，客户端代码是否不正确地调用了它，如果是，就抛出一个预定义的 `ObjectDisposedException` 异常对象。`Open()` 方法也会检查 `isDisposed` 字段，看看是否已调用 `Dispose()` 方法。因为调用 `Dispose()` 方法会告诉调用者现在已经处理完对象，所以，如果已经调用了 `Dispose()` 方法，就说明有一个试图打开新文件连接的错误。

接着，这个方法包含前两个内层的 `try` 块，其目的是捕获因为文件的第一行没有包含一个整数而抛出的任何错误。如果出现这个问题，.NET 运行库就抛出一个 `FormatException` 异常，该异常捕获并转换为一个更有意义的异常，这个更有意义的异常表示 `cold-call` 文件的格式有问题。注意 `System.FormatException` 异常表示与基本数据类型相关的格式问题，而不是与文件有关，所以在本例中它不是传递回主调例程的一个特别有用的异常。新抛出的异常会被最外层的 `try` 块捕获。因为这里不需要清理资源，所以不需要 `finally` 块。

如果一切正常，就把 `isOpen` 字段设置为 `true`，表示现在有一个有效的文件连接，可以从中读取数据。

`ProcessNextPerson()` 方法也包含一个内层 `try` 块：

```

public void ProcessNextPerson()
{
    if (isDisposed)
    {
        throw new ObjectDisposedException("peopleToRing");
    }

    if (!isOpen)
    {
        throw new UnexpectedException(
            "Attempted to access coldcall file that is not open");
    }

    try {
        {
            string name;
            name = sr.ReadLine();
            if (name == null)
                throw new ColdCallFileFormatException("Not enough names");
            if (name[0] == 'B')
            {
                throw new SalesSpyFoundException(name);
            }
            Console.WriteLine(name);
        }
    }
    catch (SalesSpyFoundException ex)

```

```

    {
        Console.WriteLine(ex.Message);
    }

    finally
    {
    }
}

```

这里可能存在两个与文件相关的错误(假定实际上有一个打开的文件连接, `ProcessNextPerson()` 方法会先进行检查)。第一, 读取下一个人名时, 可能发现这是一个工业间谍。如果发生这种情况, 在这个方法中就使用第一个 `catch` 块捕获异常。因为这个异常已经在循环中被捕获, 所以程序流会继续在程序的 `Main()` 方法中执行, 处理文件中的下一个人名。

如果读取下一个人名, 发现已经到达文件的末尾, 也会发生错误。`StreamReader` 对象的 `ReadLine()` 方法的工作方式是: 如果到达文件末尾, 它就会返回一个 `null`, 而不是抛出一个异常。所以, 如果找到一个 `null` 字符串, 就说明文件的格式不正确, 因为文件的第一行中的数字要比文件中的实际人数多。如果发生这种错误, 就抛出一个 `ColdCallFileFormatException` 异常, 它由外层的异常处理程序捕获(使程序终止执行)。

同样, 这里不需要 `finally` 块, 因为没有要清理的资源, 但这次要放置一个空的 `finally` 块, 表示在这里可以完成用户希望完成的任务。

这个示例就要完成了。`ColdCallFileReader` 异常类还有另外两个成员: `NPeopleToRing` 属性返回文件中假定的人数, `Dispose()` 方法可以关闭已打开的文件。注意 `Dispose()` 方法仅返回它是否被调用——这是实现该方法的推荐方式。它还检查在关闭前是否有一个文件流要关闭。这个例子说明了防御编码技术, 这正是所要实现的:

```

public uint NPeopleToRing
{
    get
    {
        if (isDisposed)
        {
            throw new ObjectDisposedException("peopleToRing");
        }

        if (!isOpen)
        {
            throw new UnexpectedException(
                "Attempted to access cold-call file that is not open");
        }

        return nPeopleToRing;
    }
}

public void Dispose()
{
    if (isDisposed)
    {
        return;
    }
}

```

```

        isDisposed = true;
        isOpen = false;

        if (fs != null)
        {
            fs.Close();
            fs = null;
        }
    }
}

```

15.3.3 定义用户定义的异常类

最后，需要定义 3 个异常类。定义自己的异常非常简单，因为几乎不需要添加任何额外的方法。只需实现构造函数，确保基类的构造函数正确调用即可。下面是实现 `SalesSpyFoundException` 异常类的完整代码：



可从
wrox.com
下载源代码

```

class SalesSpyFoundException: ApplicationException
{
    public SalesSpyFoundException(string spyName)
        : base("Sales spy found, with name " + spyName)
    {
    }

    public SalesSpyFoundException(
        string spyName, Exception innerException)
        : base(
            "Sales spy found with name " + spyName, innerException)
    {
    }
}

```

代码段 SolicitColdCall.cs

注意，这个类派生自 `ApplicationException` 异常类，正是我们期望的自定义异常。实际上，如果要更正式地创建它，可以把它放在一个中间类中，例如 `ColdCallFileException` 异常类，它派生于 `ApplicationException` 异常类，再从这个类派生出两个异常类，并确保处理代码可以很好地控制哪个异常处理程序处理哪个异常即可。但为了使这个示例比较简单，就不这么做了。

在 `SalesSpyFoundException` 异常类中，处理的内容要多一些。假定传送给它的构造函数的信息仅是找到的间谍名，从而把这个字符串转换为含义更明确的错误信息。我们还提供了两个构造函数，其中一个构造函数的参数只是一条消息，另一个构造函数的参数是一个内层异常。在定义自己的异常类时，至少把这两个构造函数都包括进来(尽管以后将不能在示例中使用 `SalesSpyFoundException` 异常类的第 2 个构造函数)。

对于 `ColdCallFileFormatException` 异常类，规则是一样的，但不必对消息进行任何处理：

```

class ColdCallFileFormatException: ApplicationException
{
    public ColdCallFileFormatException(string message)
        : base(message)
    {
    }

    public ColdCallFileFormatException(

```

```

        string message, Exception innerException)
        : base(message, innerException)
        {
        }
    }
}

```

最后是 `UnexpectedException` 异常类, 它看起来与 `ColdCallFormatException` 异常类是一样的:

```

class UnexpectedException: ApplicationException
{
    public UnexpectedException(string message)
    : base(message)
    {
    }

    public UnexpectedException(string message, Exception innerException)
    : base(message, innerException)
    {
    }
}

```

下面准备测试该程序。首先, 使用 `people.txt` 文件, 其内容已经在前面列出了。

```

4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson

```

它有 4 个名字(与文件中第一行给出的数字匹配), 包括一个间谍。接着, 使用下面的 `people2.txt` 文件, 它有一个明显的格式错误:

```

49
George Washington
Benedict Arnold
John Adams
Thomas Jefferson

```

最后, 尝试该例子, 但指定一个不存在的文件名 `people3.txt`, 对这 3 个文件名运行程序 3 次, 得到的结果如下:

```

SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people.txt
George Washington
Sales spy found, with name Benedict Arnold
John Adams
Thomas Jefferson
All callers processed correctly

SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people2.txt
George Washington
Sales spy found, with name Benedict Arnold

```

```
John Adams
Thomas Jefferson
The file people2.txt appears to have been corrupted.
Details of the problem are: Not enough names

SolicitColdCall
Please type in the name of the file containing the names of the people to be cold
called > people3.txt
The file people3.txt does not exist.
```

最后，这个应用程序说明了处理程序中可能存在的错误和异常的许多不同方式。

15.4 小结

本章介绍了 C# 通过异常处理错误情况的多种机制，我们不仅可以输出代码中的一般错误代码，还可以用指定的方式处理最特殊的错误情况。有时一些错误情况是通过 .NET Framework 本身提供的，有时则需要编写自己的错误情况，如本章的例子所示。在这两种情况下，都可以采用许多方式来保护应用程序的工作流，使之不出现不必要和危险的错误。

下一章将利用前面学习的许多内容，在 .NET 开发人员的 IDE——Visual Studio 2010 中实践这些内容。

第 II 部分

Visual Studio

➤ 第 16 章 Visual Studio 2010

➤ 第 17 章 部署

第 16 章

Visual Studio 2010

本章内容:

- 使用 Visual Studio 2010
- 使用 Visual Studio 2010 的重构功能
- Visual Studio 2010 的多目标功能
- 使用 WPF、WCF、WF 等新技术

至此我们已经熟悉了 C# 语言, 下面开始学习本书的应用部分, 这部分将介绍如何使用 C# 编写各种应用程序。在此之前, 需要了解如何使用 Visual Studio 以及 .NET 环境提供的一些功能来编写程序。

本章将介绍在 .NET 环境中编程的真正含义, 讨论 Visual Studio, 这是主要的开发环境, 在该环境中可以编写、编译、调试和优化 C# 程序。本章还提供编写良好的应用程序的规则。Visual Studio 是用于编写 Web 窗体、Windows 窗体、Windows Presentation Foundation(WPF) 应用程序、Silverlight 应用程序、XML Web 服务等的主要 IDE。关于 Windows 窗体及用户界面代码的编写方式详见第 39 章。

本章还介绍构建面向 .NET Framework 4 的应用程序的步骤。使用 Visual Studio 2010 可以间接编写最新的应用程序类型, 例如 Windows Presentation Foundation(WPF)、Windows Communication Foundation(WCF) 和 Windows Workflow Foundation(WF)。

16.1 使用 Visual Studio 2010

Visual Studio 2010 是一个完全集成的开发环境, 用于编写、调试代码, 尽可能简单地把代码编译为程序集进行发布。实际上, Visual Studio 提供了一个非常专业的多文档界面应用程序, 在该应用程序中可以进行与开发代码相关的任何操作, 它提供了:

- **文本编辑器**——在文本编辑器中, 可以编写 C# 代码(以及 Visual Basic 2010 和 VC++ 代码)。这个文本编辑器相当复杂, 例如, 在输入语句时, 它可以自动布局代码, 方法是缩进代码行、匹配代码块的左右花括号、提供彩色编码的关键字等。在输入语句时, 它还能执行一些语法检查, 它给可能产生编译错误的代码加上下划线, 这也称为设计期间的调试。它还提供了 IntelliSense 功能。在开始输入时, IntelliSense 会自动显示类名、字段名或方法名。在开始输入方法的参数时, IntelliSense 也会显示可用的重载方法的参数列表。图 16-1 显示了这个 IntelliSense 功能, 此时操作的是一个 .NET 基类 ListBox。



当 IntelliSense 列表框因某种原因不可见时，请按快捷键 Ctrl+Space，可以在需要时打开 IntelliSense 列表框。

- 设计视图编辑器——它可以在项目中放置用户界面和数据访问控件。此时，Visual Studio 会自动在源文件中添加必要的 C#代码，在项目中实例化这些控件(在.NET 中，所有的控件实际上都是特定基类的实例)。
- 辅助窗口——它们可以查看和修改项目的各个方面。例如，这些窗口可以显示源代码中的类以及 Windows 窗体和 Web 窗体类中的可用属性(和它们的初始值)。也可以使用这些窗口指定编译选项，如代码需要引用哪些程序集。

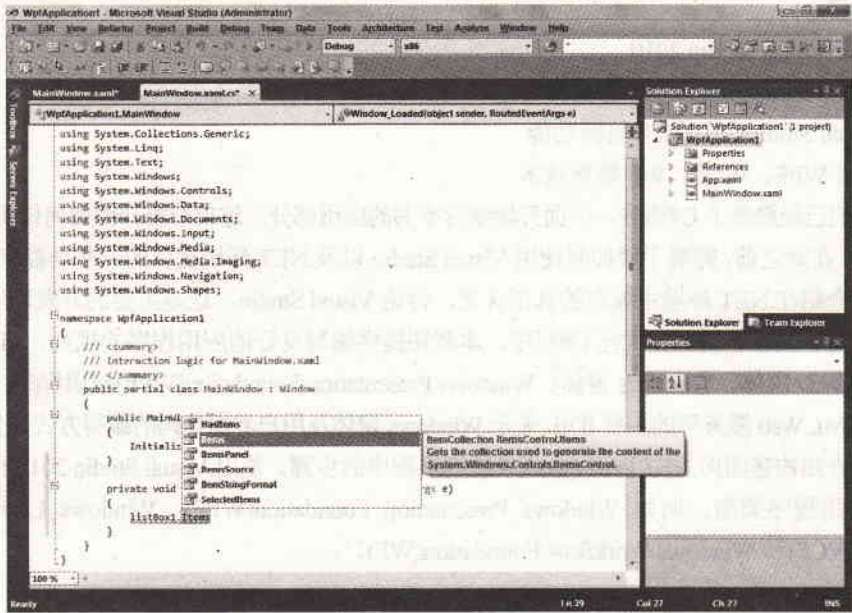


图 16-1

- 能够在环境中编译——可以只选择一个菜单选项编译项目，而不必在命令行上运行 C#编译器。Visual Studio 会调用 C#编译器，把所有的相关命令行参数传递给编译器，例如，要引用哪个程序集和要生成什么类型的程序集(例如，可执行文件或库.dll)。根据需要，Visual Studio 还可以直接运行编译好的可执行文件，用户可以查看这些文件的运行情况是否正常，甚至可以选择不同的编译配置，例如，编译为发布版本或调试版本。
- 集成的调试器——代码在第一次运行时，一般不会正确运行。也许在第二次、第三次才能正确运行。Visual Studio 无缝地链接到一个调试器上，可以在该调试环境中设置断点并监控变量。
- 集成的 MSDN 帮助——Visual Studio 可以在 IDE 中访问 MSDN 文档。例如，在文本编辑器中，如果不能确定某个关键字的含义，可以选择它，按 F1 键，Visual Studio 就打开 MSDN，显示相关的主题。同样，如果不知道某个编译错误是什么意思，可以选择错误消息，按 F1 键，打开相关文档，系统就会显示该错误信息。

- **访问其他程序**——Visual Studio 还能访问许多其他实用程序来查看和修改计算机或网络的一些内容，而无需退出开发环境。利用这些工具，可以检查正在运行的服务和数据库连接，直接查询 SQL Server 表，甚至打开 Internet Explorer 窗口，浏览 Web。

当然，如果用户很熟悉 C++ 或 Visual Basic，就应很熟悉 Visual Studio 6 版本的 IDE，上面列出的许多特性将不是什么新内容了。Visual Studio 把以前在所有 Visual Studio 6 开发环境中可以使用的所有特性都组合起来，无论用户以前在 Visual Studio 6 中使用什么语言，在 Visual Studio 中都会发现一些新增功能。例如，在旧的 Visual Basic 环境中，不能分别编译调试版本和发布版本。另一方面，如果用户有 C++ 编程经验，现在开始使用 C#，就可以获得许多数据访问支持，还可以通过单击把控件拖放到应用程序上，这些功能 Visual Basic 开发人员已经使用很久了，而在 C# 中将是新增功能。



对于有 C++ 编程经验的人，Visual Studio 2010 去除了 Visual Studio 6 中的两个功能：编辑并继续调试和集成的探查器。Visual Studio 2010 也不包含功能完善的探查器应用程序。但在 System.Diagnostics 名称空间中有许多 .NET 类能帮助进行配置。perfmon 配置工具可以在命令行上使用(仅输入 perfmon)，它有许多与 .NET 相关的新性能监视器。

无论用户有什么编程背景，都会发现，与 Visual Studio 6 相比，Visual Studio 2010 开发环境已经有了整体上的改进，包括增加一些新功能，一个跨语言的 IDE 和与 .NET 的集成。菜单和工具栏有一些新选项，许多选项都是 Visual Studio 6 已有的，但进行了重命名。所以，用户需要花一些时间熟悉 Visual Studio 2010 的布局和命令。

Visual Studio 2008 和 Visual Studio 2010 的区别仅限于 Visual Studio 2010 中几个新增的特性。在 Visual Studio 2010 中，最大的变化是整个编辑器在 .NET Framework 4 上进行了重构，使用了 Microsoft Extensibility Framework(MEF)和 WPF。Visual Studio 2010 包含可以面向 .NET Framework 的特定版本(包括 .NET Framework 2.0、3.0、3.5 或 4)，JavaScript IntelliSense 支持和使用 CSS 的新功能。还可以构建 ASP.NET AJAX 应用程序和使用出自 Microsoft 的最新技术的应用程序，包括 WCF、WF 和 WPF。

在安装 Visual Studio 2010 时，会注意到一个最大的变化：这个重新构建的 IDE 与 .NET Framework 4 一起工作。实际上，在安装 Visual Studio 2010 时，如果还没有安装 .NET Framework 4，系统就会自动安装它。与 Visual Studio 2005/2008 一样，这个新的 IDE(Visual Studio 2010)不能与 .NET Framework 1.0 或 1.1 一起工作，如果仍要开发 1.0 或 1.1 版本的应用程序，就应在计算机上分别安装 Visual Studio 2002 或 2003。安装 Visual Studio 2010 时，会安装 Visual Studio 的一个完整的新副本，但不会升级原来的 Visual Studio 2002、2003、2005 或 2008 IDE。根据需要，Visual Studio 的这 4 个版本可以在计算机上并行运行。

如果试图使用 Visual Studio 2010 打开 Visual Studio 2002 和 2008 之间的项目，IDE 就会发出警告：如果继续，就会弹出 Visual Studio Conversion Wizard 窗口(如图 16-2 所示)，将该项目升级到 Visual Studio 2010。

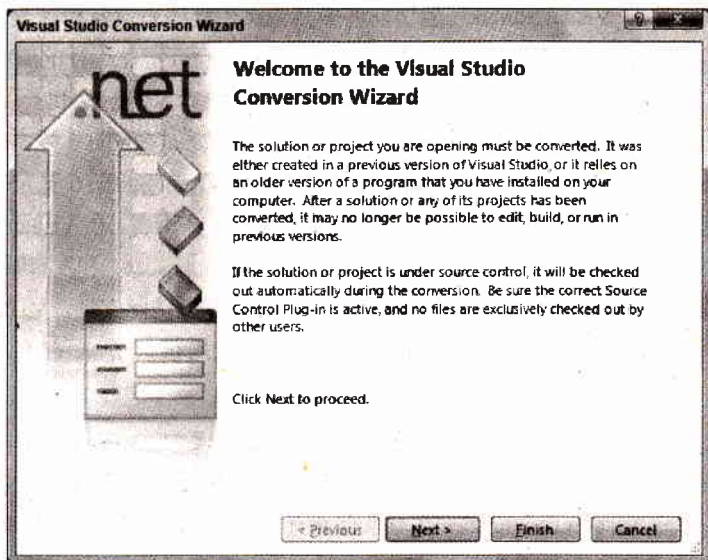


图 16-2

升级向导从 Visual Studio 2003 迁移到 Visual Studio 2010 时进行了巨大的改进。在备份之前，这个向导可以对解决方案进行备份复制(如图 16-3 所示)，还可以备份源控件中包含的解决方案。

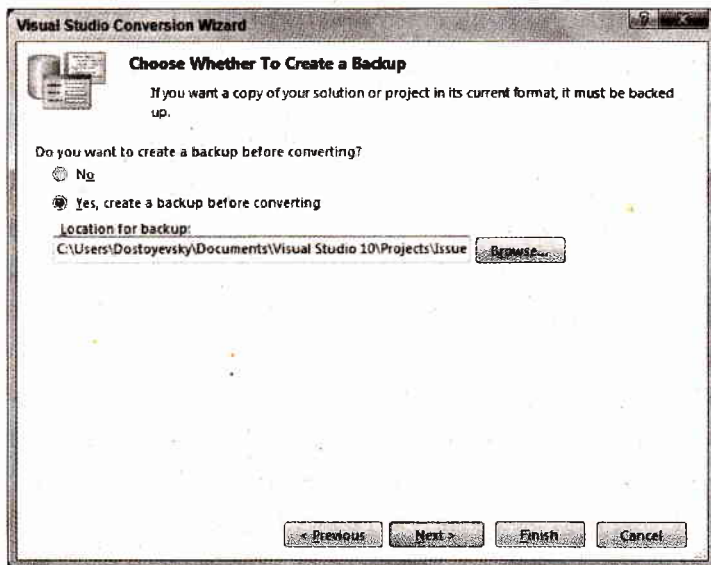


图 16-3

还可以在转换过程的最后一步生成一个转换报告。这个报告可以在 Visual Studio 的文档窗口中直接查看，如图 16-4 所示(进行了一个简单的转换)。

因为本书适合于专业人员使用，所以不会详细介绍 Visual Studio 2010 中的每个功能和菜单项。用户应自己熟悉该 IDE。介绍 Visual Studio 的主要目的是让用户熟悉构建和调试 C# 应用程序所涉及的概念，这样才能更好地使用 Visual Studio 2010。图 16-5 显示了 Visual Studio 2010 的外观(注意，Visual Studio 的外观是高度可定制的，在启动该开发环境时，看到的可能是位置或内容不同的窗口)。

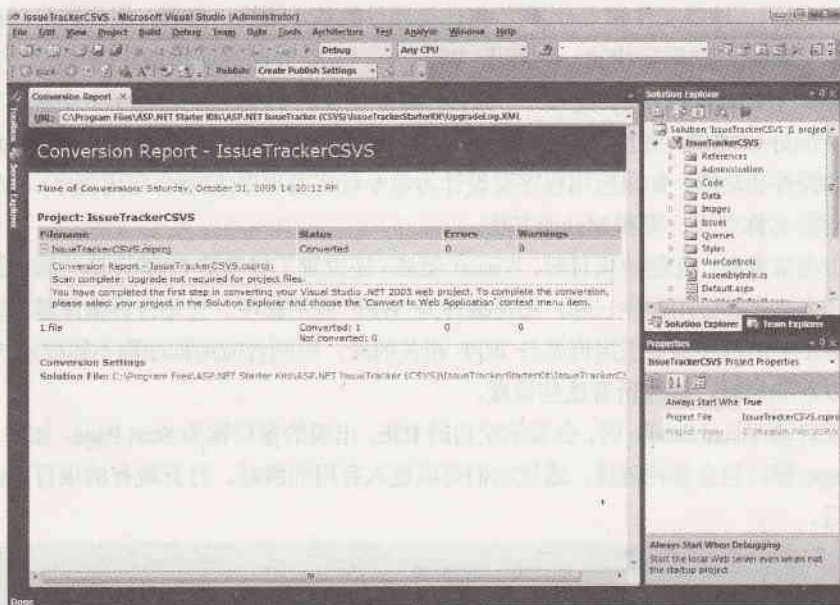


图 16-4

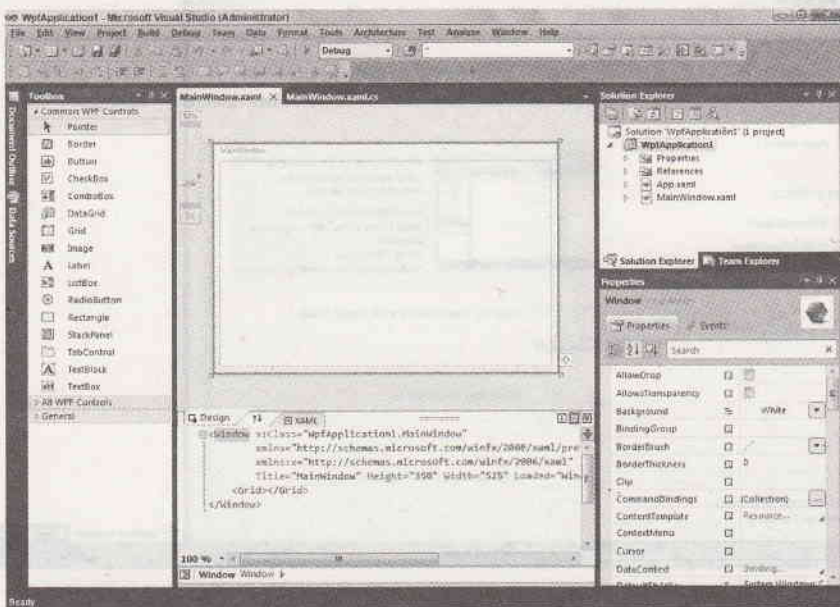


图 16-5

下面几节将创建、编写和调试项目，查看 Visual Studio 在每个阶段所能完成的操作。

16.1.1 创建项目

安装好 Visual Studio 2010 后，就可以开始编写第一个项目了。在 Visual Studio 中，很少从一个空白文件开始，从头输入 C# 代码，就像本书前面的章节那样(当然，如果确实要从头开始编写代码，或者创建一个包含许多项目的解决方案，该 IDE 也提供了空应用程序项目选项)。

编写项目的方式一般是先告诉 Visual Studio 要创建什么类型的项目，然后 Visual Studio 会自动

生成文件和 C# 代码, 这些文件和 C# 代码给出该类项目的基本架构。接着, 用户就可以在其中添加自己的代码了。例如, 如果要构建一个基于 Windows GUI 界面的应用程序(在 .NET 中, 这称为 Windows 窗体), Visual Studio 就会建立一个文件, 其中包含的 C# 源代码创建了一个基本窗体。这个窗体可以与 Windows 通信, 并接收事件。它还可以最大化、最小化或重新设置大小, 用户只需在其中添加需要的控件和功能。如果应用程序要设计为命令行实用程序(控制台应用程序), Visual Studio 就会提供基本的名称空间、类和 Main() 方法。

最后一步也很重要, 在创建项目时, Visual Studio 还设置了提供给 C# 编译器的编译选项——表示项目是编译为命令行应用程序、库, 还是编译为 WPF 应用程序。它还告诉编译器需要引用的基类库(WPF GUI 应用程序需要引用许多与 WPF 相关的库, 控制台应用程序则不需要)。当然如果必要, 用户可以在编辑时, 修改所有这些设置。

在第一次启动 Visual Studio 时, 会显示空白的 IDE, 出现的窗口称为 Start Page, 如图 16-6 所示。这个 Start Page 窗口包含各种链接, 通过它们可以进入有用的网站, 打开现有的项目, 或者同时启动一个新项目。

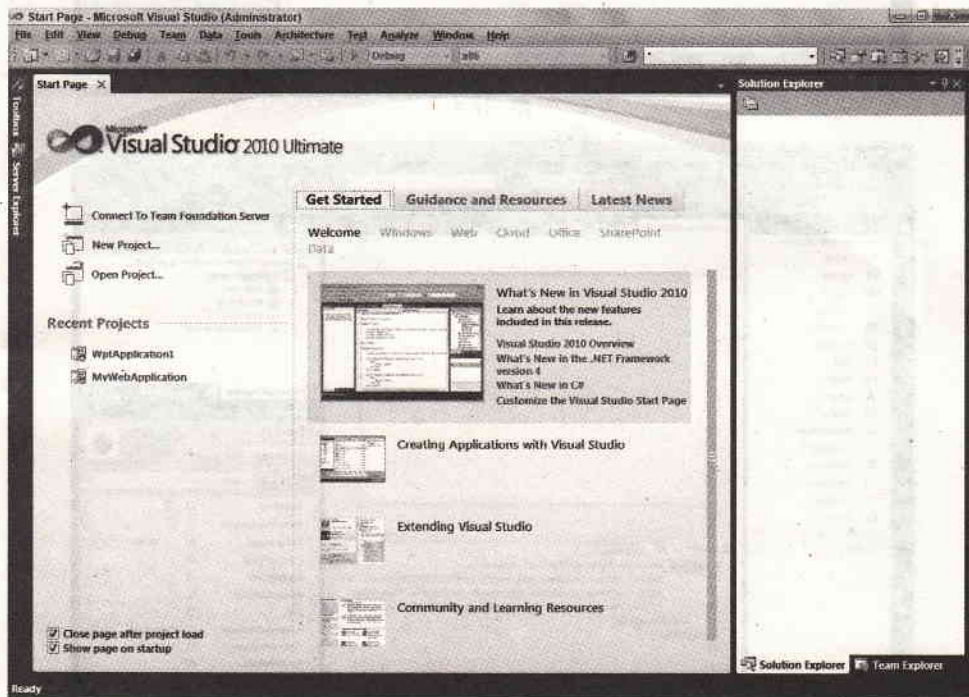


图 16-6

图 16-6 显示了使用 Visual Studio 2010 打开的 Start Page 类型的窗口, 其中有一个最近编辑的项目列表。单击其中的一个项目就可以打开它。

1. 选择项目类型

创建新项目时, 可以从 Visual Studio 的菜单中选择 File | New Project 命令, 打开 New Project 对话框, 如图 16-7 所示, 其中给出了可以创建的各种项目。

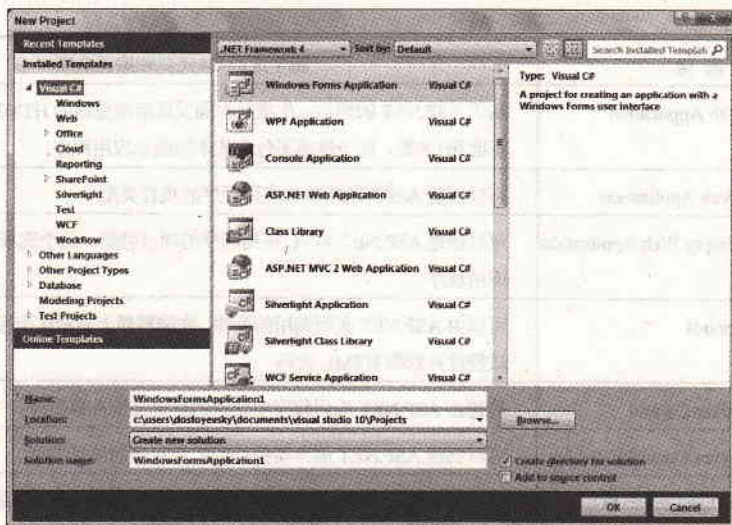


图 16-7

使用该对话框，可以选择 Visual Studio 为用户生成的某种初始架构文件和代码、编译选项，以及编译代码所使用的编译器：C# 2010、Visual Basic 2010 或 C++ 2010 编译器。从这里可以立即看出，Microsoft 为 .NET 提供了多种语言集成。本例选择了 C# 控制台应用程序。



这里不打算介绍不同类型的项目的所有选项。在 C++ 方面，Visual Studio 可以创建所有旧的 C++ 项目类型——MFC 应用程序、ATL 项目等。在 Visual Basic 2008 方面，选项有一些变化，例如，可以创建 Visual Basic 命令行应用程序(控制台应用程序)、.NET 组件(类库)或者 Windows 窗体控件(Windows 窗体控件库)等，但不能创建基于 COM 的旧风格的控件(.NET 控件可以取代这种 ActiveX 控件)。

表 16-1 列出了 Visual C# Projects 下所有可用的选项。注意，在 Other Projects 选项下还有一些比较专业的 C# 模板项目。

表 16-1

如果选择	将生成的 C# 代码和编译选项
Windows Forms Application	响应事件的基本空窗体
Class Library	可以由其他代码调用的 .NET 类
WPF Application	响应事件的基本空窗体。这个项目类型类似于 Windows Forms Application 项目类型(Windows Forms)，这个 Windows Application 项目类型允许构建基于 XAML 的智能客户解决方案
WPF Browser Application	类似于 WPF 的 Windows Application，但这个变体类型可以构建面向浏览器的基于 XAML 的应用程序
ASP.NET Web Application	基于 ASP.NET 的网站：生成从页面发送到浏览器的 HTML 响应的 ASP.NET 页面和 C# 类。这个选项包含一个基本的演示应用程序

(续表)

如果选择	将生成的 C# 代码和编译选项
Empty ASP.NET Web Application	基于 ASP.NET 的网站：生成从页面发送给浏览器的 HTML 响应的 ASP.NET 页面和 C# 类。这个选项不包含基本的演示应用程序
ASP.NET MVC 2 Web Application	可以创建 ASP.NET MVC 应用程序的项目类型
ASP.NET MVC 2 Empty Web Application	可以创建 ASP.NET MVC 应用程序的项目类型。这个选项不包含基本的演示应用程序
ASP.NET Server Control	可以由 ASP.NET 页面调用的控件，在浏览器上显示这个控件时，可生成给出该控件外观的 HTML 代码
ASP.NET AJAX Server Control	构建在 ASP.NET 应用程序中使用的自定义服务器控件
ASP.NET AJAX Server Control Extender	可以创建 ASP.NET 服务器控件的扩展程序的项目类型
ASP.NET Dynamic Data Linq to SQL Web Application	可以构建通过 Linq to SQL 使用 ASP.NET Dynamic Data 的 ASP.NET 的项目类型
ASP.NET Dynamic Data Entities Web Application	可以构建通过 Linq to Entities 使用 ASP.NET Dynamic Data 的 ASP.NET 的项目类型
Silverlight Application	可以构建 Silverlight 应用程序的项目类型
Silverlight Navigation Application	可以构建 Silverlight 应用程序的项目类型，这个应用程序以一个核心 Silverlight 应用程序作为开始，可以根据自己的需要进行扩展
Silverlight Class Library	可以创建 Silverlight 类库的项目类型
WPF Custom Control Library	可以在 WPF 应用程序中使用的自定义控件
WPF User Control Library	用 WPF 建立的用户自定义控件库
Windows Forms Control Library	用于创建在 Windows Forms 应用程序中使用的控件的项目
Syndication Service Library	允许构建和提供联合服务的 WCF 项目
Console Application	在命令行提示符上或控制台窗口中运行的应用程序
WCF Service Application	用于 WCF 服务的项目类型
Windows Service	在 Windows 操作系统的后台上运行的服务
Enable Windows Azure Tools	可以加载基于 Azure 的工具，以集成计算解决方案
Reports Application	用于创建带有 Windows 用户界面和报表的项目
Crystal Reports Application	该项目用于创建带有 Windows 用户界面和水晶报表示例的 C# 应用程序
Activity Designer Library	为使用 Windows Workflow 提供 Activity Designer 模板的项目
Activity Library	提供空白的 Workflow Activity Library 的项目。这个项目可以创建活动库，以后在工作流中重用于构建块
Workflow Console Application	提供与 Windows Workflow 一起使用的基本控制台应用程序的项目
WCF Service Library	这个项目可以创建 WCF 服务类库(.dll)，其端点通过 XML 配置文件来控制
WCF Workflow Service Application	可以创建基于 WCF 的分布式通信应用程序，该应用程序使用 Windows Workflow
Office	这组项目可以构建面向 Microsoft Office(Word、Excel、PowerPoint、InfoPath、Outlook、Visio 和 SharePoint)的应用程序或插件

如前所述,这并不是.NET Framework 4 项目的完整列表,但这是一个很好的开始。这个项目列表增加了许多内容,主要是面向 WPF、WCF 和 WF 的最新项目。本书后面的章节将介绍这些新功能。请参阅第 35 章、第 43 章和第 44 章。

还要注意,可以使用新的项目模板,因为可以通过 New Project 对话框在线搜索模板。

2. 回顾新建的控制台项目

在上述对话框中选择 Console Application 选项,单击 OK 按钮,Visual Studio 就会提供几个文件,包括一个源代码文件 Program.cs,其中包含了最初的架构代码。图 16-8 显示了 Visual Studio 编写的代码。

可以看出,这是一个 C# 程序,但它实际上没有做任何工作,只是包含了任何 C# 可执行程序所必需的基本项:一个名称空间和一个包含 Main() 方法的类,其中 Main() 方法是程序的入口点(严格来说,名称空间是不必要的,但不声明名称空间是一种不好的编程习惯)。按 F5 键,或者选择 Debug 菜单中的 Start,这段代码就可以编译和运行。在这样做之前,在程序中添加一行代码,让应用程序完成某些工作:

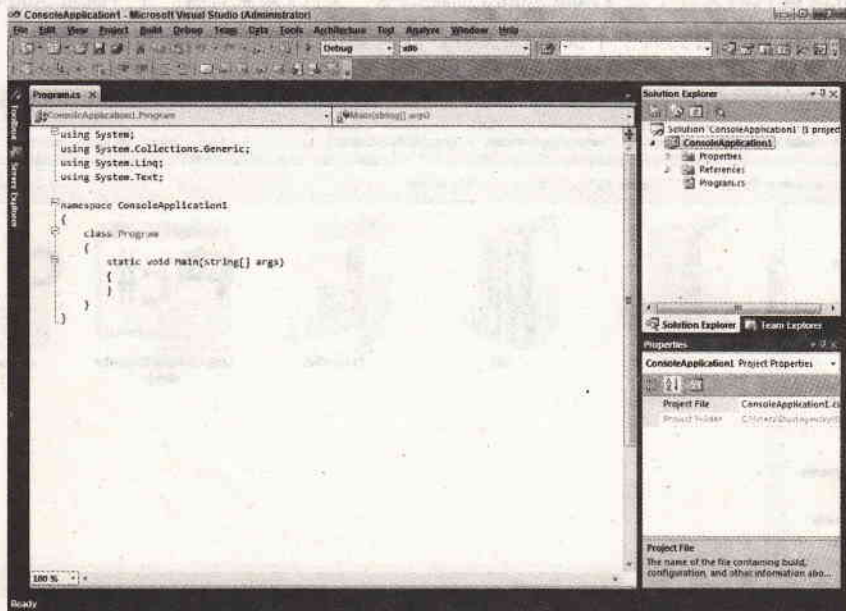


图 16-8

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the authors of Professional C#");
}
```

如果编译并运行了该项目,就会显示一个控制台窗口,但该窗口几乎立即消失了,用户几乎看不到输出的消息。原因是在创建该项目时,Visual Studio 记住了用户指定的设置,所以会把它编译并运行控制台应用程序。然后,Windows 知道它需要运行一个控制台应用程序,但没有从中运行该程序的控制台窗口。所以,Windows 就创建一个控制台窗口,并运行该程序。只要程序退出,Windows 就认为它不再需要该控制台窗口,因此就立即删除它。这些都是非常逻辑化的操作,但如

果希望能看到项目的输出结果，这些操作对用户就没有什么帮助。

要避免这个问题，可以在代码中 `Main()` 方法返回前插入下述代码：

```
static void Main(string[] args)
{
    Console.WriteLine("Hello from all the folks at Wrox Press");
    Console.ReadLine();
}
```

这样，代码运行后，会显示其输出结果，之后执行 `Console.ReadLine()` 语句，此时用户按回车键后，程序退出。这表示在用户按回车键之前，控制台窗口一直会挂起。

注意这仅是从 Visual Studio 中测试运行控制台应用程序的一个问题。如果编写的是一个 WPF 或 Windows Forms 应用程序，该应用程序显示的窗口会自动停留在屏幕上，直到用户显式程序为止。同样，如果从命令行提示符运行一个控制台应用程序，就没有窗口消失的问题。

3. 查看其他项目文件

`Program.cs` 源代码文件不是 Visual Studio 创建的唯一文件。如果查看一下 Visual Studio 在其中创建项目的文件夹，就会看到其中不仅有 C# 文件，还有如图 16-9 所示的完整目录结构。

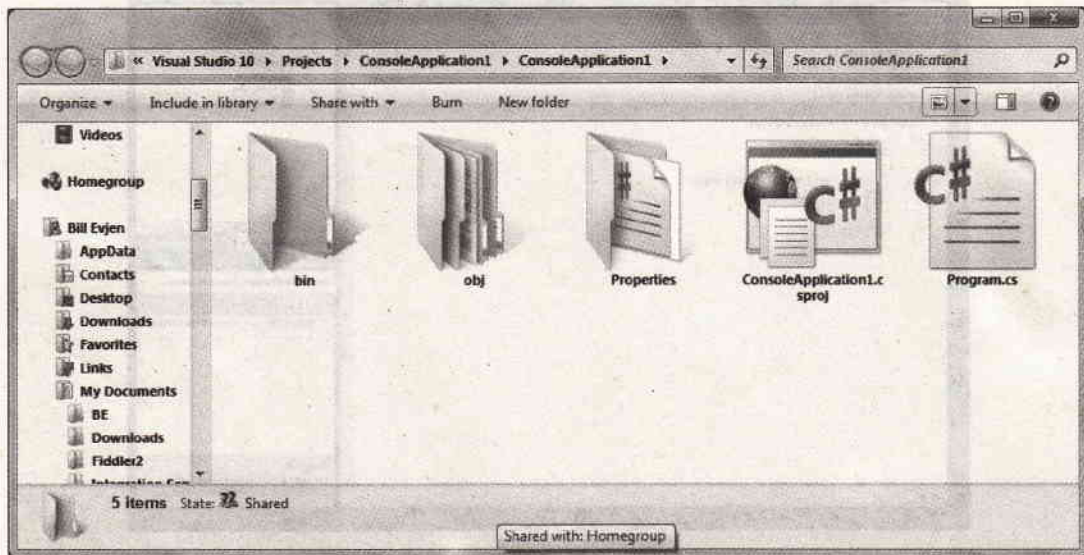


图 16-9

文件夹 `bin` 和 `obj` 存储编译好的文件和中间文件，`obj` 的子文件夹存储各种临时或中间文件，`bin` 的子文件夹存储编译好的程序集。



传统上，Visual Basic 开发人员只编写代码，然后运行它。代码在发布前，必须编译成可执行文件，但 Visual Basic 在调试时隐藏了这个编译过程。而在 C# 中，这个过程比较明显：要运行代码，必须先编译(或生成)它，即在某处创建一个程序集。

还有一个包含 `AssemblyInfo.cs` 文件的 `Properties` 文件夹。在项目的主文件夹 `ConsoleApplication1` 中, 剩余的文件都是由 Visual Studio 建立的。它们包含项目的信息(例如, 它包含的文件), 这样, Visual Studio 就知道如何编译项目, 在下次打开该项目时, 知道如何读取它。

16.1.2 解决方案和项目的区别

项目和解决方案的一个重要区别是:

- 项目是一组要编译到单个程序集(在某些情况下, 是单个模块)中的所有源代码文件和资源。例如, 项目可以是类库, 或 Windows GUI 应用程序。
- 解决方案是构成某个软件包(应用程序)的所有项目组成的集。

为了说明这个区别, 考虑一下在发布一个项目(该项目包含多个程序集)时的情况。例如, 其中可能有一个用户界面、自定义控件和其他组件, 它们都作为应用程序的库文件一起发布。不同的管理员甚至还有不同的用户界面。应用程序的不同部分都包含在一个独立的程序集中, 因此, 在 Visual Studio 看来, 它们都是独立的项目。但我们要同时编写这些项目, 使它们彼此连接起来。所以, 在 Visual Studio 中把它们当作一个单元来编辑就很重要。在 Visual Studio 中, 可以把所有的项目看作一个解决方案, 把该解决方案当作是它可以读入的单元, 并允许用户在其上工作。

前面大致讨论了如何创建一个控制台项目。实际上, 在前面的例子中, Visual Studio 创建的是一个解决方案, 尽管这个特定的解决方案只包含一个项目。可以在 Visual Studio 的 `Solution Explorer` 窗口中查看它, 该窗口包含一个定义解决方案的树型结构, 如图 16-10 所示。

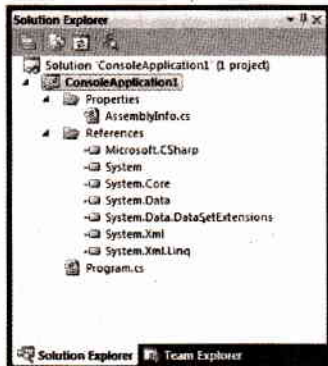


图 16-10

图 16-10 说明了项目包含源文件 `Program.cs` 和另一个 C#源文件 `AssemblyInfo.cs`(在文件夹 `Properties` 中), `AssemblyInfo.cs` 包含程序集的描述信息且能够指定版本信息(该文件详见第 18 章)。Solution Explorer 也指定了项目根据名称空间引用的程序集。展开 Solution Explorer 窗口中的 `Reference` 文件夹, 就可以看到它。

如果没有改变 Visual Studio 中的任何默认设置, Solution Explorer 就在屏幕的右上角。如果看不到它, 就可以选择 `View` 菜单中的 `Solution Explorer` 命令。

解决方案用扩展名为 `.sln` 的文件来表示, 在本例中, 就是 `ConsoleApplication1.sln`。该项目由项目的主文件夹中的各个其他文件来表示。如果试图使用 Notepad 编辑这些文件, 就会发现它们大多数都是纯文本文件, 为了与 .NET 和可能依赖于开放标准的 .NET 工具对应的原则保持一致, 它们大都是 XML 格式。



C++开发人员应认识到, Visual Studio 解决方案对应于旧的 C++项目工作区(存储在.dsw文件中), Visual Studio 项目对应于旧的 C++项目(.dsp文件)。另一方面, Visual Basic 开发人员应注意, 解决方案对应于旧的 Visual Basic 项目组(.vbg文件), .NET 项目对应于旧的 Visual Basic 项目(.vbp文件)。Visual Studio 与旧 Visual Basic IDE 的区别是, Visual Studio 总是自动创建一个解决方案。在 Visual Studio 6 中, Visual Basic 开发人员最初会得到一个项目, 然而, 如果要得到项目组, 就必须在 IDE 中显式指定。

1. 给解决方案添加另一个项目

下面几节将说明 Visual Studio 如何处理 Windows 应用程序和控制台应用程序。为此, 本节将创建一个 Windows 项目 BasicForm, 并把它添加到当前的解决方案 ConsoleApplication1 中。



最终我们将得到包含一个 Windows 应用程序和一个控制台应用程序的解决方案。这种情况并不常见——我们一般会在解决方案中包含一个应用程序和多个库, 但这个解决方案可以给出更多的代码。例如, 如果用户编写的实用程序需要作为 Windows 应用程序和命令行实用程序来运行, 就可以创建这样的解决方案。

新建项目可以使用两种方式: 一是进入 File 菜单, 选择 New Project 命令(如前所述)。二是在 File 菜单中选择 Add | New Project 命令。如果从 File 菜单中选择 New Project 选项, 会打开熟悉的 Add New Project 对话框, 但这次 Visual Studio 要在已有的 ConsoleApplication1 项目位置中新建项目, 如图 16-11 所示。

如果选中这个选项, 就会添加一个新项目, ConsoleApplication1 解决方案现在就应包含一个控制台应用程序和一个 Windows 应用程序。



为了保持 Visual Studio 的语言无关性, 新项目不一定是 C#项目。在同一个解决方案中, 可以有 C#项目、Visual Basic 项目和 C++项目。但这里仍使用 C#, 因为这是一本介绍 C#的书。

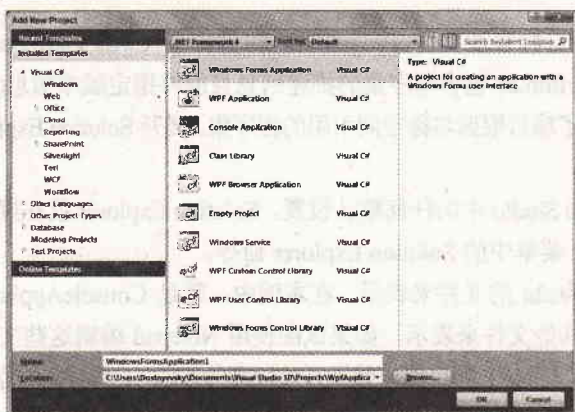


图 16-11

当然,这意味着 ConsoleApplication1 对于这个解决方案,不再是一个合适的名称了。右击该解决方案的名称,从弹出的上下文菜单中选择 **Rename** 命令,改变其名称。如果把它重命名为 DemoSolution。Solution Explorer 窗口就应如图 16-12 所示。

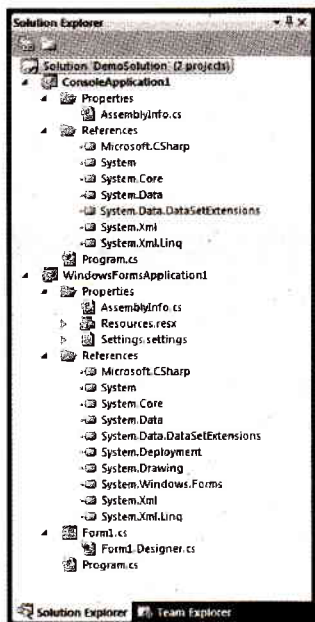


图 16-12

从这里可以看出, Visual Studio 会自动让新增的 Windows 项目引用那些对 Windows 窗体的功能非常重要的某些附加基类。

如果查看一下 Windows 资源管理器,就会发现解决方案的文件名已经改为 DemoSolution.sln。一般情况下,如果要重命名任何文件,最好在 Solution Explorer 中进行,因为 Visual Studio 会自动更新其他项目文件中对该文件的任何引用。如果使用 Windows 资源管理器重命名文件,就会中断解决方案,因为 Visual Studio 不能定位它需要读入 IDE 的所有文件。用户必须手动编辑项目和解决方案文件,以更新文件引用。

2. 设置启动项目

如果在解决方案中有多个项目,就必须确保该解决方案在某一刻只运行一个项目。在编译解决方案时,将编译其中的所有项目。但在按 F5 键或选择 **Start** 菜单时,必须告诉 Visual Studio 先运行哪个项目。如果有一个可执行文件,它调用了几个库,显然就应先运行这个可执行文件。对于本例,项目中有两个独立的可执行文件,就必须逐个调试它们。

可以告诉 Visual Studio 应运行哪个项目,方法是在 Solution Explorer 窗口中右击该项目,在弹出的上下文菜单中选择 **Set as Startup Project** 命令。这就告诉 Visual Studio 哪个项目是当前的启动项目,因为它在 Solution Explorer 窗口中用黑体显示,在图 16-12 上,就是 ConsoleApplication1。

16.1.3 Windows 应用程序代码

在 Visual Studio 第一次创建应用程序时, Windows 应用程序包含的启动代码要比控制台应用程

序多, 因为创建一个窗口是一个比较复杂的过程。第 39 章将详细讨论 Windows 应用程序的代码, 这里只给出 WindowsApplication1 项目中 Form1 类的代码, 说明自动生成的代码有多少。

16.1.4 项目的浏览和编码

本节将介绍在给项目添加代码时, Visual Studio 所提供的一些功能。

1. 可折叠的编辑器

Visual Studio 的一个重要特性是, 它把可折叠的编辑器当作默认的代码编辑器, 如图 16-13 所示。

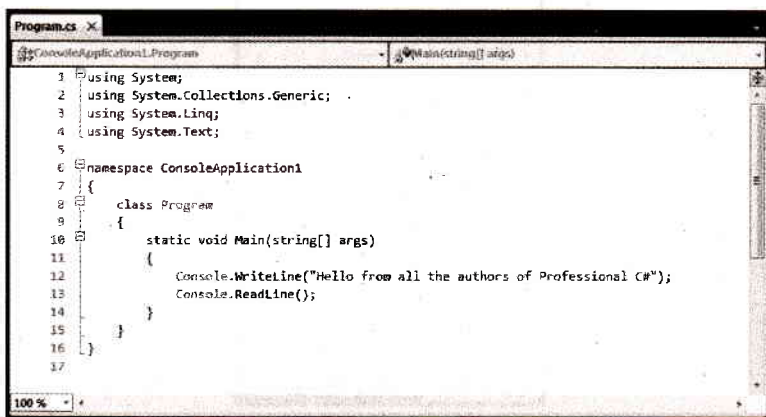


图 16-13

图 16-13 显示了前面生成的控制台应用程序的代码。但要注意窗口左边的小减号图标, 它们标记了编辑器认为是新代码块(或文档注释)的起点。单击这些图标, 可以关闭对应的代码块视图, 就像关闭树形控件中的节点一样, 如图 16-14 所示。

这意味着, 在编辑时, 可以只考虑要查看的那些代码块, 隐藏目前不想查看的代码块。而且, 如果不喜欢编辑器隐藏代码块的方式, 就可以用 C# 预处理器指令 `#region` 和 `#endregion` (详见本书前面的章节) 指定折叠代码块的另一种方式。例如, 假定要折叠 `Main()` 方法中的代码, 可以先添加如图 16-15 所示的代码。

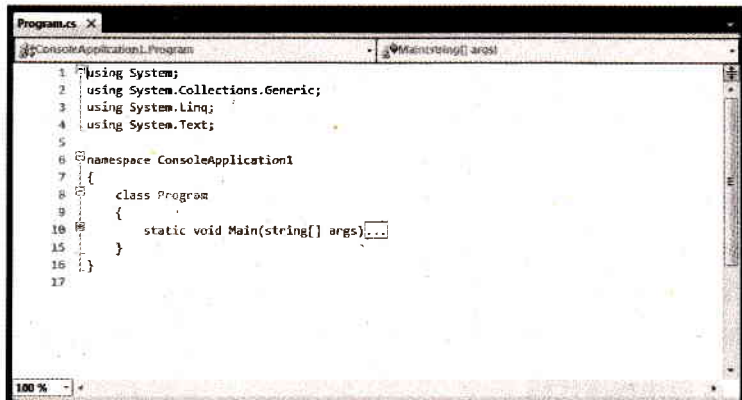


图 16-14

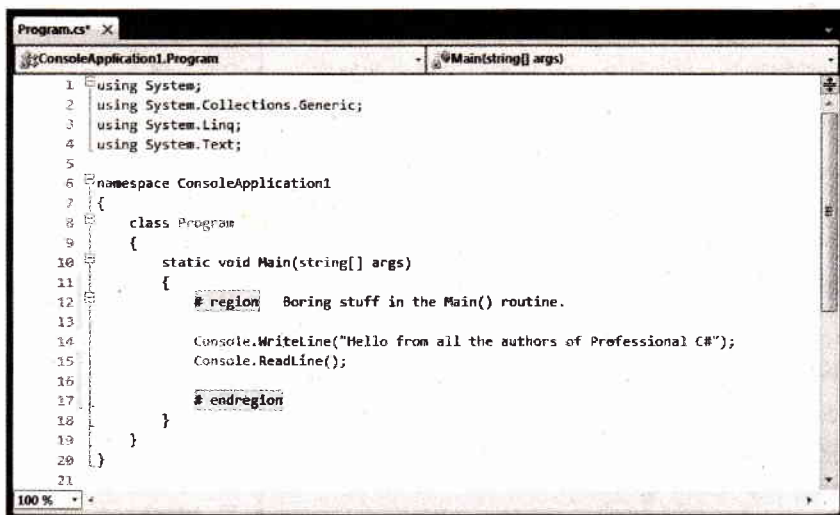


图 16-15

代码编辑器会自动检测#region 块，通过#region 指令放置一个新的减号图标，如图 16-15 所示，以便关闭该代码区域。把这个代码块放在一个区域中，就可以用在#region 指令中指定的注释标记该区域，让编辑器关闭该代码块，如图 16-16 所示。但编译器会忽略这些指令，像往常一样编译 Main() 方法。

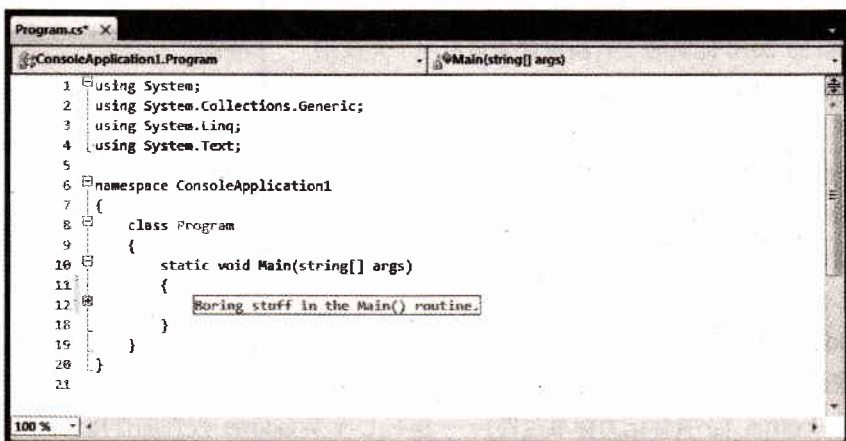


图 16-16

除了可折叠的编辑器对应的功能之外，Visual Studio 的代码编辑器还拥有 Visual Studio 6 的全部功能，特别是它支持 IntelliSense 功能，这不仅减少了输入代码的工作量，还可以确保使用正确的参数。C++开发人员会注意到，Visual Studio 的 IntelliSense 功能比 Visual Studio 6 版本的更强大，速度也更快。IntelliSense 功能自从 Visual Studio 2005 以来也得到了进一步的改进。它更加智能，可以记住用户喜欢的选项，并从这些选项开始，而不是从 IntelliSense 功能提供的有时很长的列表的开头开始。

代码编辑器也对代码进行一些语法检查，在编译代码前用短波浪线划出大多数语法错误。把鼠标悬停在有下划线的文本上面，就会弹出一个小文本框，解释什么错误。这个功能 Visual Basic 开发人员已使用了多年，所谓设计期间的调试功能，现在 C#和 C++开发人员也可以使用它。

2. 其他窗口

除了代码编辑器外, Visual Studio 还提供了许多其他窗口, 这些窗口允许用户以不同的角度查看项目。

本节的其余部分将介绍许多其他的窗口。如果某个窗口在屏幕上不可见, 就可以进入 View 菜单, 选择合适窗口的名称。要显示设计视图和代码编辑器, 可以在 Solution Explorer 窗口中右击文件名, 然后从弹出的上下文菜单中选择 View Designer 或 View Code 命令, 也可以从 Solution Explorer 窗口顶部的工具栏中选择相应菜单项。设计视图和代码编辑器共用同一个选项卡式窗口。

(1) Design View 窗口

如果设计一个用户界面应用程序, 如 Windows 应用程序、Windows 控件库或者 ASP.NET 应用程序, 就可以使用 Design View 窗口, 它会显示窗体的整体外观。Design View 窗口一般和 Toolbox 窗口一起使用。Toolbox 窗口包含许多可以拖放到程序中的 .NET 组件, 如图 16-17 所示。

Visual Studio 2010 提供的工具箱包含大量可用于开发的组件。组件的种类在某种程度上取决于用户所编辑项目的类型——例如, 在编辑 DemoSolution 解决方案中的 Windows FormsApplication1 项目时, 可以使用的组件种类就比编辑 ConsoleApplication1 项目时多。最重要的组件种类如下:

- **数据访问组件**——可以连接数据源并管理它们包含的数据的类。这类组件可处理 Microsoft SQL Server、Oracle 和 OleDb 数据源。
- **Windows Forms 控件(标记为常用控件)**——表示可视化控件的类, 如文本框、列表框或树型视图, 用于处理胖客户端应用程序。
- **Web Forms 控件(标记为标准控件)**——基本上与 Windows 控件的作用一样的类, 但用于 Web 浏览器, 把模拟控件的 HTML 输出结果发送给浏览器(只有使用 ASP.NET 应用程序时才能看到该结果)。
- **其他组件**——在计算机上执行各种有用任务的 .NET 类, 如连接目录服务或事件日志。

也可以把自己的自定义组件类别添加到工具箱中, 方法是右击任意类别, 从弹出的上下文菜单中选择 Add Tab 命令。从该上下文菜单中选择 Choose Items 命令, 就可以把其他工具放在工具箱中。这非常适合于添加自己喜欢的 COM 组件和 ActiveX 控件。在默认情况下, COM 组件和 ActiveX 控件不会显示在工具箱中。如果要添加一个 COM 控件, 就可以单击它, 并把它拖放到项目上, 就像操作 .NET 控件一样。Visual Studio 会自动添加所有必需的 COM 交互操作性代码, 以便项目调用该控件。此时, 添加到项目中的实际上是 Visual Studio 在后台创建的一个 .NET 控件, 它是所选 COM

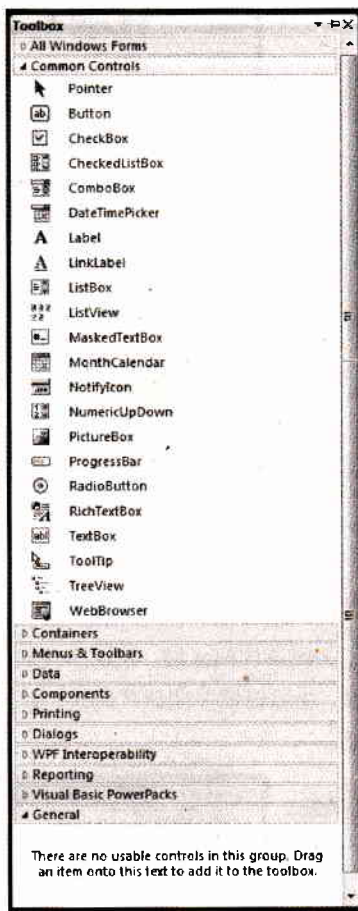


图 16-17

控件的容器。

为了说明工具箱如何工作，把一个文本框放在基本窗体项目上。首先单击工具箱中的 `TextBox` 控件，再单击一次，把它放在设计视图的窗体中(也可以直接把控件拖放到设计界面上)。设计视图如图 16-18 所示，该图也是编译并运行 `WindowsFormsApplication1` 项目的结果。

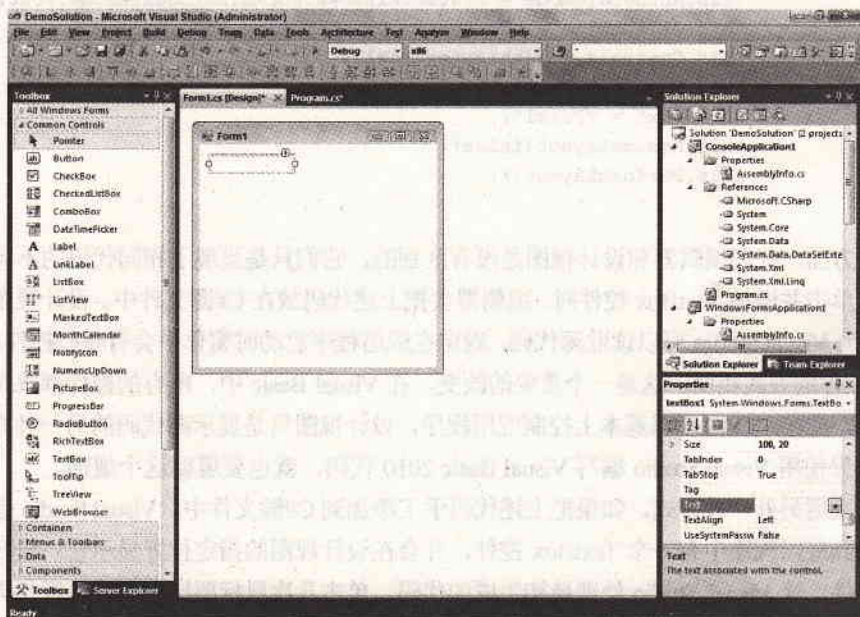


图 16-18

在窗体的代码视图上，Visual Studio 2010 并没有像 IDE 的以前版本那样，直接实例化放在窗体上的 `TextBox` 对象。单击并展开 Visual Studio Solution Explorer 窗口中 `Form1.cs` 文件旁边的加号，会看到一个文件 `Form1.Designer.cs`，它用于窗体及其中控件的设计。在这个类文件中，`Form1` 类有一个新的成员变量：

```
partial class Form1
{
    private System.Windows.Forms.TextBox textBox1;
```

在 `InitializeComponent()` 方法中还有一些需要初始化的代码，该方法从 `Form1` 类的构造函数中调用：

```
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.SuspendLayout();
    //
    // textBox1
    //
    this.textBox1.Location = new System.Drawing.Point(0, 0);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(100, 20);
```

```

this.textBox1.TabIndex = 0;
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 265);
this.Controls.Add(this.textBox1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
this.PerformLayout();
}

```

在某一方面，代码编辑器和设计视图是没有区别的，它们只是呈现了相同代码的不同视图。在设计视图上单击并添加 **TextBox** 控件时，编辑器会把上述代码放在 C# 源文件中。设计视图反映了这个变化，因为 **Visual Studio** 可以读取源代码，确定在应用程序启动时窗体中会有哪些控件。与 **Visual Basic** 查看控件的方式相比，这是一个重要的改变，在 **Visual Basic** 中，所有的控件都放在可视化的设计视图中。现在，C# 源代码基本上控制应用程序，设计视图只是显示源代码的另一种方式。顺便提一下，如果使用 **Visual Studio** 编写 **Visual Basic 2010** 代码，就也要遵循这个规则。

还可以采用另外一种方式，如果把上述代码手工添加到 C# 源文件中，**Visual Studio** 也会自动从代码中检测到应用程序中有一个 **TextBox** 控件，并会在设计视图的指定位置显示它。最好可视化地添加这些控件，让 **Visual Studio** 处理最初生成的代码，单击几次鼠标要比输入好几行代码快得多，也不容易出错！

可视化地添加这些控件的另一个原因是为了确定应用程序中有这些控件，**Visual Studio** 确定需要使相关的代码遵循某些条件——手动编写的代码可能不遵循这些条件。特别是 **InitializeComponent()** 方法包含初始化 **TextBox** 控件的代码，在它的注释中警告用户不要修改代码。这是因为 **Visual Studio** 要查找这个方法，以确定应用程序在启动时有哪些控件。如果在代码的其他地方创建和定义了一个控件，**Visual Studio** 不知道有这个控件，就不能在设计视图或其他某些有用的窗口中编辑它。

实际上，无论有什么警告，都可以在 **InitializeComponent()** 方法中修改代码，但应非常小心。例如，修改一些属性的值一般不会出什么问题，如让某个控件显示不同的文本，或者改变它的大小。实际上，开发人员工作室非常擅长于处理在这个方法中添加的任何其他代码。但要注意，如果对 **InitializeComponent()** 方法进行了过多的修改，**Visual Studio** 就有可能识别不出使用代码添加的控件。需要强调的是，在编译代码时，这不会影响到应用程序，但它可能禁用 **Visual Studio** 为这些控件提供的某些编辑功能。因此，如果要进行任何其他重要的初始化，那么最好在 **Form1** 类的构造函数或其他方法中进行。

(2) Properties 窗口

这是从旧 **Visual Basic IDE** 继承而来的另一个窗口。本书的第 I 部分说过，.NET 类可以实用属性。实际上，如第 39 章(讨



图 16-19

论 Windows 窗体的建立)所述,表示窗体和控件的.NET 基类有许多定义其操作或外观的属性。例如,Width、Height、Enabled(用户是否可以在该控件中输入信息)和 Text(控件所显示的文本),Visual Studio 能识别其中的许多属性。对于 Visual Studio 能通过读取源代码检测到的控件,Properties 窗口可以显示和编辑大多数属性的初始值,如图 16-19 所示。



Properties 窗口也可以显示事件。单击窗口顶部的闪电图标,就可以查看 IDE 中当前选中的事件,或者查看在 Properties 窗口的下拉列表中选择的事件。

在 Properties 窗口的顶部,有一个列表框,从中可以选择要查看的控件。本章的这个例子选择的是 Form1 控件,即 WindowsFormsApplication1 项目的主窗体类,将其文本编辑为“Basic Form — Hello!”。如果此时查看源代码,就会看到刚才的操作实际上是通过一个友好的用户界面编辑了源代码:

```
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleModeMode.Font;
this.ClientSize = new System.Drawing.Size(284, 265);
this.Controls.Add(this.textBox1);
this.Name = "Form1";
this.Text = "Basic Form — Hello";
this.ResumeLayout(false);
this.PerformLayout();
```

并不是 Properties 窗口中的所有属性都会在源代码中显式指定。对于没有显式指定的属性,Visual Studio 会显示在创建窗体时给它们设置的默认值,这些默认值实际上是在初始化窗体时设置的。显然,如果在 Properties 窗口中修改这些属性的值,源代码中就会出现一条显式地设置该属性的语句,反之亦然。有趣的是,如果属性是从其初始值改变而来,这个属性在 Properties 窗口的列表框中就会显示为黑体。有时双击 Properties 窗口中的属性,会返回其初始值。

Properties 窗口提供了一种查看控件或窗口的外观和属性的简便方式。



Properties 窗口实现为一个 System.Windows.Forms.PropertyGrid 实例,该实例在内部使用第 14 章介绍的反射技术,来标识要显示的属性和属性值。

(3) Class View 窗口

与 Properties 窗口不同,Class View 窗口最初是从 C++(和 J++)开发环境继承而来的一个窗口,如图 16-20 所示。Visual Studio 实际上并没有把 Class View 窗口看作是一个窗口,而是 Solution Explorer 窗口的一个附加选项卡。默认情况下,Class View 窗口不会显示在 Visual Studio 的 Solution Explorer 窗口中。要打开 Class View 窗口,应选择 View | Class View 命令。如图 16-20 所示,Class View 窗口显示了代码中名称空间和类的层次结构,它给出了一个树形视图,用户可以展开该视图,以查看哪些名称空间包含了什么类,哪些类包含了什么成员。

Class View 窗口的一个功能是,如果右击在源代码中可以访问的任何项的名称,弹出的上下文菜单就会提供一个 Go To Definition 选项,单击它,就可以访问代码编辑器中该项的定义。还可以在 Class View 窗口中双击该项(实际上是在源代码编辑器中右击需要的项,从弹出的上下文菜单中选择相同的选项)来完成同样的操作。上下文菜单还允许给类添加字段、方法、属性或索引器。这意味着

可以在一个对话框中指定相关成员的详细信息，系统会自动添加对应的代码。这对于字段和方法可能不太有效，因为它们可以快速添加到代码中，但对于属性和索引器它就非常有用，因为它可以减少大量的输入工作。

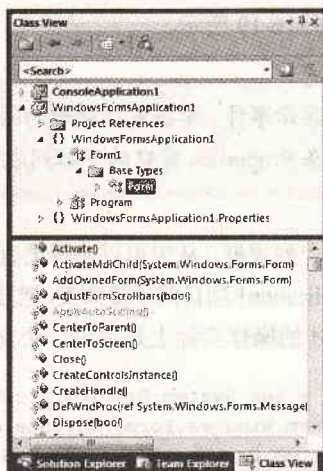


图 16-20

(4) Object Browser 窗口

在 .NET 环境中编程的一个重要优点是可以确定从程序集中引用的基类和任何其他库中有什么方法和其他代码项。这个功能通过 Object Browser 窗口来获得。在 Visual Studio 2010 的 View 菜单中选择 Object Browser 命令，就可以访问这个窗口。

Object Browser 窗口非常类似于 Class View 窗口，它也显示一个树型视图，该树型视图给出了应用程序的类结构，允许查看每个类的成员。用户界面则略有不同，因为它在一个单独的窗格中显示类的成员，而不是在树型视图中显示。但真正的区别是它不仅可以查看项目中的名称空间和类，还可以查看项目所引用的所有程序集中的名称空间和类。图 16-21 显示了利用 Object Browser 窗口查看 .NET 基类中的 SystemException 异常类的情况。

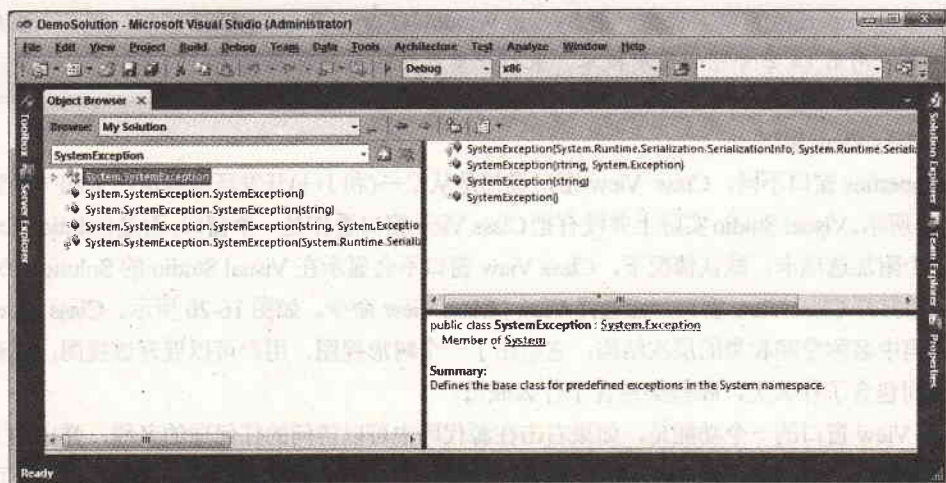


图 16-21

在 Object Browser 窗口中必须注意的一点是,它先按照类所在的程序集对类进行分组,再按照名称空间对类进行分组。但是,因为基类的名称空间常常分布在多个程序集中,所以在定位某个特定类时可能会遇到麻烦,除非知道该类在哪个程序集中。

Object Browser 窗口可以查看 .NET 对象。如果由于某些原因要查看已安装的 COM 对象,就会发现以前在 C++ 的 IDE 中使用的 OLEView 工具仍然可用,它在文件夹 C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin 中,该文件夹还有几个其他类似的实用程序。

(5) Server Explore 窗口

Server Explore 窗口可以用于确定编码时网络中计算机各个方面的情况,如图 16-22 所示。

从上面的屏幕截图中可以看出,可以通过 Server Explore 窗口访问数据库连接、服务信息和事件日志等。

Server Explore 窗口与 Properties 窗口链接在一起,所以,如果打开 Services 节点,单击某个特定服务,该服务的属性就会显示在 Properties 窗口中。

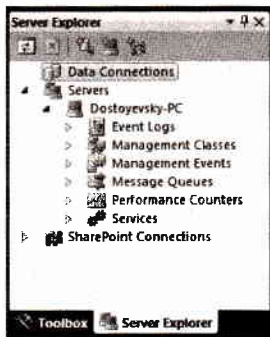


图 16-22

3. Pin 按钮

在浏览 Visual Studio 时,前面介绍的许多窗口包含一些有趣的功能,很容易让人回想起工具栏的一些功能。除了代码编辑器外,它们都是固定的。另一个功能是在这些窗口处于固定状态时,在每个窗口右上角的最小化按钮的旁边就会出现一个大头针图标,这个图标的工作方式就跟大头针一样,把已打开的窗口固定。在窗口被固定时(大头针垂直显示),它们的操作就像一般的窗口一样。但当它们没有被固定时(大头针水平显示),只要它们获得焦点,就仍然会打开;而失去焦点时(用户单击了其他地方),它们就会无声无息地退到整个 Visual Studio 应用程序的主边界上(打开和关闭该按钮时,计算机的速度也有快慢变化)。

固定或不固定窗口是充分利用屏幕有限空间的另一种方式。以前在 Windows 中这个功能用得不多,但在一些第三方应用程序(如 PaintShop Pro)中就使用了类似的概念。固定的窗口在许多基于 Unix 的系统上已经有了一定的应用。

16.1.5 生成项目

Visual Studio 不仅可以编写项目,正是这个 IDE 实际上管理着项目的整个生命周期,包括生成或编译解决方案。本节介绍 Visual Studio 为生成项目提供的选项。

1. 生成、编译和产生项目

在介绍各种生成选项前，首先澄清一些术语。在把源代码变成某类可执行的代码时，常常会使用3个不同的术语：编译、生成和产生。这3个术语的出现是因为直到最近，把源代码变成可执行代码的过程需要多个步骤(在C++中仍是这样)。这在很大程度上是因为一个程序常常包含许多源文件。

例如，在C++中，每个源文件都需要单独编译。这样就生成了对象文件，每个对象文件都包含一些可执行代码，但每个对象文件都仅与一个源文件相关。为了生成可执行代码，这些对象文件必须链接在一起，这个过程就称为链接。这个组合过程通常称为(至少在Windows平台上)生成代码。但是，在C#中，编译器更为专业，能够把所有的源文件当作一个块来读取和处理。因此，就没有独立的链接阶段，在C#中，“编译(compile)”和“生成(build)”可以互换。

术语“产生(make)”的基本含义与“生成(build)”相同，但在C#环境中一般不使用。该术语来源于旧的大型计算机系统，在该系统上，当项目由许多源文件组成时，就编写一个独立的文件，其中包含的指令可以指示编译器如何生成项目：包括哪些文件，链接什么库等，这个文件一般称为产生文件(make file)，该文件仍然是Unix系统上的标准。产生文件在Windows上一般不需要，但如果用户需要，也可以编写它们(或者让Visual Studio生成它们)。

2. 调试版本和发布版本

有C++背景的开发人员都知道不同版本的概念，但Visual Basic开发人员就不一定知道了。在调试程序时，可执行代码进行的操作一般与准备发布该软件时所进行的操作大不相同。在准备发布软件时，除了代码正常工作外，可执行文件的大小应尽可能小，运行速度应尽可能快。但这些要求与调试代码时的要求并不相容。原因如下：

(1) 优化

要获得高性能，部分依赖于编译器对代码进行的许多优化，即编译器在编译过程中一直在监视源代码，找出某些代码，以一种不影响全局效果的方式修改某些细节信息，使其效率更高。例如，如果编译器遇到下面的源代码：

```
double InchesToCm(double Ins)
{
    return Ins*2.54;
}

// later on in the code

Y = InchesToCm(X);
```

就可以用下面的代码替换它：

```
Y = X * 2.54;
```

或者它把下面的代码：

```
{
    string Message = "Hi";
    Console.WriteLine(Message);
}
```

替换成:

```
Console.WriteLine("Hi");
```

因此, 在过程中不必声明不必要的对象引用。

不能准确地确定 C# 编译器进行了什么优化工作, 以及上面的两个例子在某些特定程序中是否有可能出现, 因为这类信息没有加以说明(对于托管语言(如 C#), 上述优化很可能在 JIT 编译期间进行, 而不是在 C# 编译器把源代码编译成程序集时进行)。从商业角度来看, 编写编译器的公司通常不会对编译器使用的技巧进行过多的说明。这里还要强调, 优化不影响源代码, 它们只影响可执行代码的内容。但是, 通过上面的例子, 您应明白优化的内涵。

问题是, 像上面例子中的优化可以使代码运行得更快, 但它们对于调试, 就没什么帮处。假定在第一个例子中, 要在 `InchesToCm()` 方法内部设置一个断点, 看看其中发生了什么。如果可执行代码实际上没有 `InchesToCm()` 方法, 因为编译器已删除了它, 该怎么办? 如果在 `Message` 变量上设置了一个监视点, 但在编译好的代码中根本没有这个变量, 该怎么办?

(2) 调试器符号

在调试时, 常常需要查看变量的值, 因此需要通过该变量在源代码中的名称来指定它们。问题是可执行代码一般不包含这些变量的名称——编译器用内存地址取代了它们。NET 对这种情形进行了一定的改进。程序集中的某些对象是与其名称一起存储的, 但只有小部分对象是这样, 如公共类和方法。这些名称仍在程序集进行 JIT 编译时被删除。如果在调试器检查可执行代码时, 要求调试器给出变量 `HeightInInches` 的值, 也不能得到我们希望的结果, 它只能给出地址, 根本就没有 `HeightInInches` 引用。因此, 为了正确地调试, 用户必须在可执行代码中添加额外的调试信息。这些信息包括变量名和代码行号, 让调试器查找与源代码指令对应的可执行机器汇编语言指令。但不能在发布版本中放置这些信息, 因为这是商业机密(调试信息可以让其他人更容易反汇编源代码), 而且会增大可执行文件。

(3) 额外的源代码调试命令

在调试时, 我们经常会遇到这样一个相关的问题: 代码中有额外的命令行显示与调试相关的重要信息。显然, 在发布软件前, 必须从可执行代码中完全删除这些相关命令。可以手动完成这个任务, 但如果仅是以某种方式标记这些语句, 让编译器在编译代码时忽略它们, 再发布软件, 不是更容易吗? 本书的第 I 部分已经讨论过, 在 C# 中, 可以定义一个合适的处理器符号, 再把它和 `Conditional` 特性结合在一起使用, 这就是所谓的条件编译。

与最终发布的产品相比, 即使把这些因素都加上, 编译所有的商业软件和调试它们也有细微差别。Visual Studio 可以把这些都考虑进去, 因为如前所述, 它存储了编译代码时应该传递给编译器的所有选项的详细信息。为了支持不同类型的生成文件, Visual Studio 只需要存储多组这类信息。不同组的生成信息就称为配置。在创建项目时, Visual Studio 会自动提供两种配置, 分别是调试和发布:

- 调试配置通常会指定不进行任何优化, 在可执行代码中给出额外的调试信息, 编译器假定给出了调试预处理器符号 `Debug`, 除非在源代码中显式指定了 `#undefined`。
- 发布配置通常指定编译器要优化编译过程, 在可执行代码中没有额外的调试信息, 编译器假定没有给出任何调试预处理器符号。

也可以定义自己的配置。例如, 要建立专业级版本和企业级版本, 可能就需要定义自己的配置,

以发布两个版本的软件。过去, 因为 Windows NT 支持 Unicode 字符编码, 而 Windows 95 不支持, 所以通常 C++ 项目有一个 Unicode 配置和一个 MBCS(多字节字符集)配置。

3. 选择配置

一个很明显的问题是, 因为 Visual Studio 存储了多个配置的信息, 在生成一个项目时, 该如何确定使用哪个配置? 答案是总是有一个活动配置, 在要求 Visual Studio 生成项目时, 就使用这个配置(注意配置是每个项目的配置, 而不是解决方案的配置)。

在默认情况下, 创建一个项目时, 调试配置就是活动配置。单击 **Build** 菜单选项, 选择 **Configuration Manager** 项, 就可以改变活动配置。还可以通过 Visual Studio 的主工具栏中的下拉菜单来改变活动配置。

4. 编辑配置

除了选择活动配置外, 还可以查看和编辑配置。为此, 需要在 **Solution Explorer** 窗口中选择相应的项目, 然后从 **Project** 菜单中选择 **Properties** 命令, 打开一个非常复杂的对话框(在 **Solution Explorer** 窗口中右击项目名, 在弹出的上下文菜单中选择 **Properties** 命令, 也可以打开这个对话框)。

该对话框包含一个树型视图, 在该树型视图中可以选择许多不同的区域, 以查看或编辑。这里不详细介绍所有的区域, 只介绍其中两个最重要的区域。

图 16-23 显示了某个特定应用程序的可用属性的选项卡视图。这个屏幕截图显示了本章前面创建的 **ConsoleApplication1** 项目的一般应用程序设置。

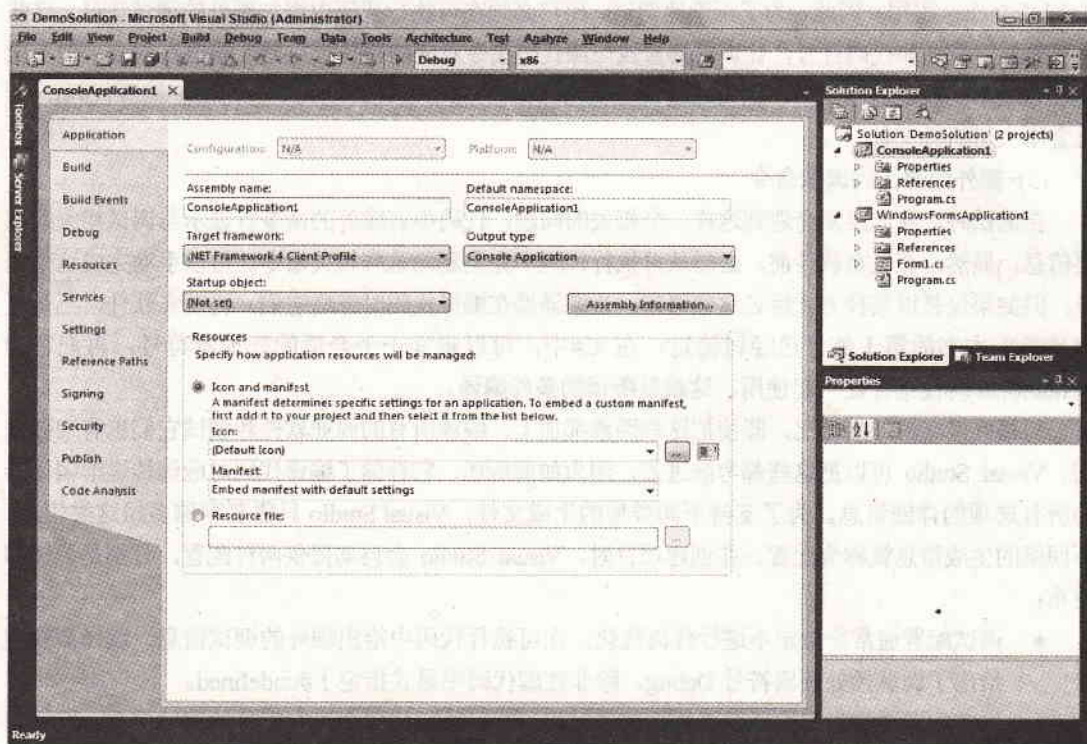


图 16-23

注意，可以选择要生成的程序集的名称和程序集的类型。其中的 **Output Type** 是 **Console Application**、**Windows Application** 和 **Class Library**。当然，还可以改变程序集的类型(尽管这还有争议，但在 Visual Studio 最初生成项目时，可能会奇怪为什么不能选择正确的项目类型呢)。

图 16-24 显示了生成文件的配置属性。注意，对话框顶部的列表框允许指定要查看的配置。此时可以查看调试配置——假定编译器定义了 **DEBUG** 和 **TRACE** 预处理器符号。如上所述，在调试配置中，代码没有优化，并会生成额外的调试信息。

一般情况下，用户不需要调整配置设置。如果需要使用它们，了解不同配置属性的区别是非常有用的。

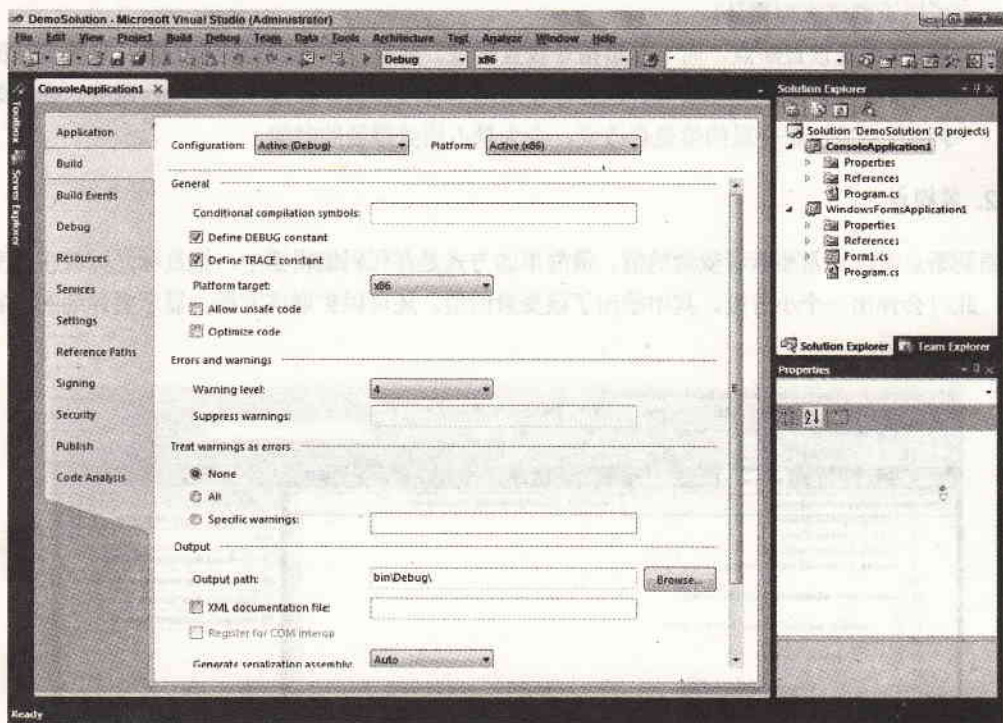


图 16-24

16.1.6 调试代码

在长时间讨论生成项目和生成配置后，我们还没有用大量的篇幅讨论代码的调试。原因是在 Visual Studio 中调试的规则和过程——设置断点和查看变量的值——与在 Visual Studio 6 IDE 中没有太大的区别。下面将简要介绍 Visual Studio 提供的调试功能，主要讨论对于某些开发人员是新内容的部分，我们还将论述如何处理异常，因为它们会给调试带来问题。

与 .NET 出现以前的语言一样，在 C# 中，调试所涉及的主要技术是设置断点，使用它们在代码的执行过程中检查某处发生的情况。

1. 断点

在 Visual Studio 中，可以在执行的代码中给任意一行设置断点。最简单的方式是在代码编辑器中单击该行，即在文档窗口左边的阴影区域中单击该行(或者选择该行，按 F9 键)，这样，就在该行

设置了一个断点，只要代码执行到该行，就会中断，把控制权交给调试器。在 Visual Studio 的以前版本中，断点在代码编辑器中用该行左边的一个大圆点表示。Visual Studio 则把该行的文本和背景用另一种颜色来突出显示。再次单击该圆点，就会删除断点。

如果程序并不适合于每次执行一行就中断一次，也可以设置条件断点。为此，可以单击 **Debug | Windows | Breakpoints** 命令，这弹出一个对话框，该对话框要求用户给出要设置的断点的详细信息，可以使用的选项有：

- 指定只有在对设置了断点的代码执行一定次数后，才能中断程序的执行。
- 指定只有执行某行一定的次数后，断点才起作用，例如，执行该行 20 次后，断点才起作用（可用于调试大型循环）。
- 给相关变量设置断点，而不是给指令设置断点。此时，监视的是变量的值，只要变量的值发生了改变，就会触发断点。但是，使用这个选项会显著减慢代码的运行速度。在每条指令执行完后检查变量的值是否改变，会大量占用处理器的时间。

2. 监视点

遇到断点时，通常要查看变量的值。最简单的方式是在代码编辑器中，把鼠标光标放在该变量名上，此时会弹出一个方框，其中给出了该变量的值。还可以扩展该方框，显示更详细的内容，如图 16-25 所示。

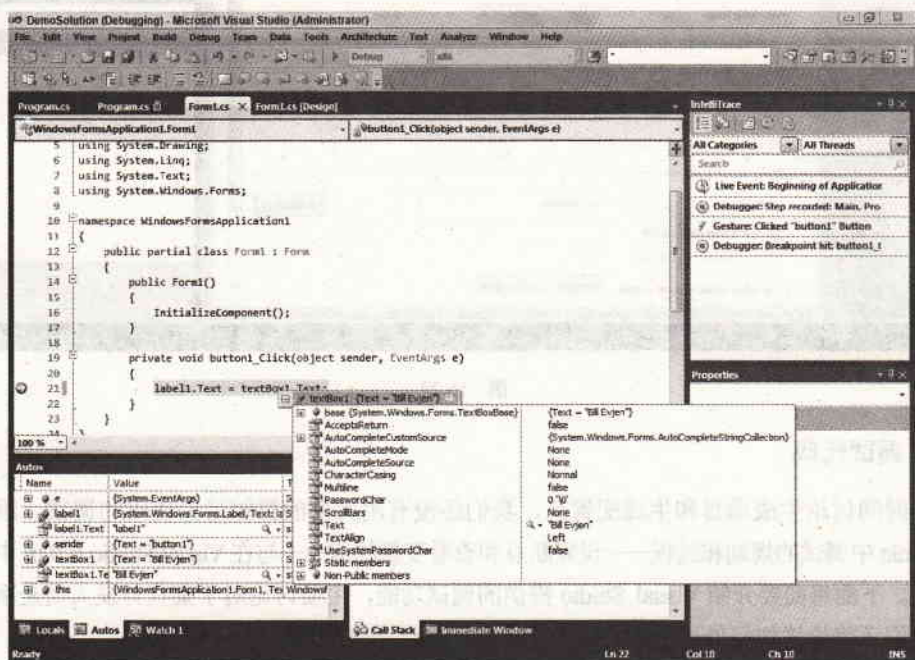


图 16-25

然而，也可以使用 **Autos** 窗口来查看变量的内容，它是一个选项卡式窗口，程序只有在调试器中运行时，该窗口才会出现，如图 16-26 所示。如果该窗口没有打开，就可以选择 **Debug | Windows | Autos** 命令。

类变量或结构变量的旁边有一个“+”图标，单击它可以展开变量，查看其字段的值。

这个窗口的 3 个选项卡主要用于监视不同的变量：

- Autos 监视程序运行时最后访问的几个变量。
- Locals 监视当前执行的方法中可访问的变量。
- Watch 监视用户在 Watch 窗口中输入的变量名时显式指定的任何变量。

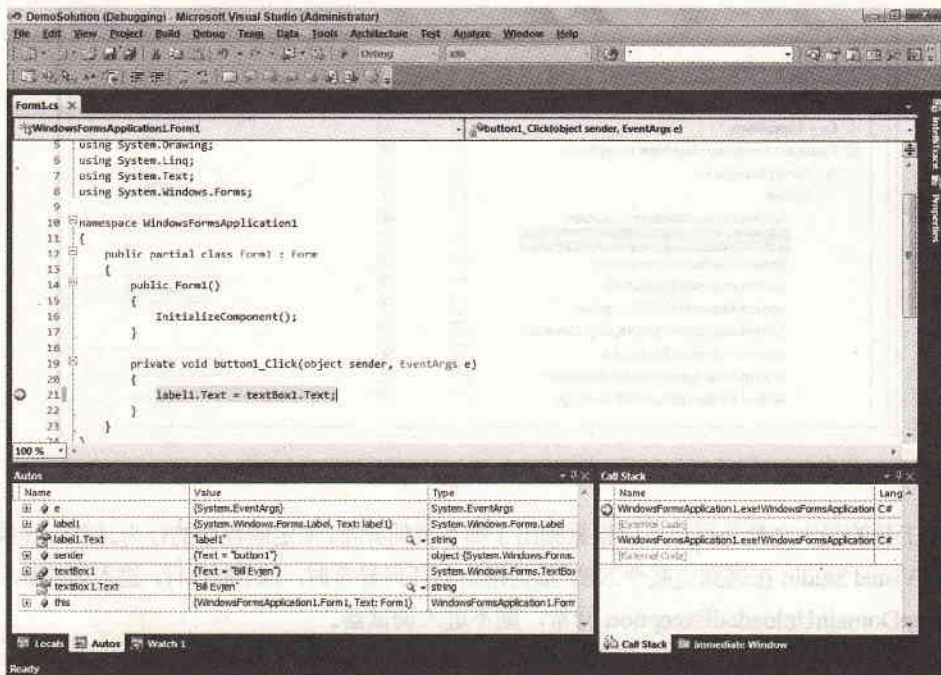


图 16-26

3. 异常

在发布应用程序时，异常可以确保在该应用程序中以合适的方式处理错误情况。它们可以保证应用程序正常运行，不会给用户显示许多技术性的对话框。但在调试时，异常并没有那么强大的功能，这个问题有两个方面：

- 如果在调试时产生一个异常，则在调试时常常不希望由程序自动处理——尤其是如果自动处理异常就意味着退出，并终止程序的执行！我们要让调试器查找到发生异常的原因。当然，问题是如果要编写出健壮的可以防范错误的好代码，程序就应能自动处理几乎所有的异常——包括要检测的错误！
- 如果产生了一个没有编写对应的处理程序的异常，.NET 运行库还是会查找该异常的处理程序。此时它发现没有这样的处理程序，因而终止程序。这里没有调用栈，不能查看任何变量的值，因为它们都已超出了作用域。

当然，可以在 catch 块中设置断点，但这常常没有什么帮助，因为在执行 catch 块时，按照定义，程序流会退出相应的 try 块。这样，要查看的变量值就超出了作用域，找不出错误发生的原因。甚至不能查看栈跟踪，找出 throw 语句退出程序时执行了哪个方法，因为该方法已交出了控制权。当然，在 throw 语句上设置断点可以解决这个问题，但如果代码中有许多 throw 语句时，该如何安全地编码？如何告诉编译器是哪条 throw 语句抛出了异常？

实际上, Visual Studio 给这些问题提供了一个非常好的解决方案。查看一下 Debug 主菜单, 其中有一个 Exceptions 菜单项, 选择它会打开 Exceptions 对话框, 如图 16-27 所示, 指定抛出异常时要执行什么操作。可以选择继续执行或者停止, 并开始调试代码, 此时程序停止执行, 调试器单步执行 throw 语句。

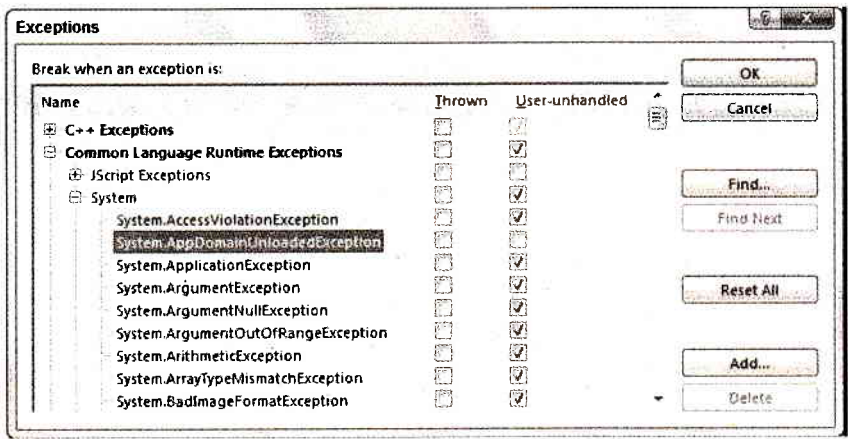


图 16-27

这个工具的强大之处是用户可以根据抛出了哪个类的异常, 来定制程序的行为。例如, 在图 16-27 上, 告诉 Visual Studio 在遇到由某个 .NET 基类抛出的任何异常时, 就中断执行, 进入调试器, 但如果抛出了 AppDomainUnloadedException 异常, 则不进入调试器。

Visual Studio 知道 .NET 基类中所有的异常类, 也知道在 .NET 环境外部抛出的许多异常。Visual Studio 不会自动识别用户编写的自定义异常类, 但用户可以把自已的异常类手动添加到该列表中, 指定哪个异常会立即停止执行程序。为此, 只需单击图 16-27 中的 Add 按钮(在从树型结构选择一个顶级节点时, 该按钮就可用), 输入异常类的名称即可。

16.2 重构工具

许多开发人员在第一次为应用程序开发功能后, 一旦该功能实现之后, 他们就会改写应用程序, 使之更易管理, 可读性更高。这个过程称为重构。重构就是改写代码, 提高可读性、性能, 提供类型安全性, 使应用程序更好地遵循标准 OO(面向对象)编程实践的过程。

因此, Visual Studio 2010 的 C# 环境现在包含一组重构工具, 这些工具位于 Visual Studio 菜单的 Refactoring 选项下。为了说明这些工具的作用, 下面在 Visual Studio 中新建一个 Car 类:

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

namespace ConsoleApplication1
{
    public class Car
    {
        public string _color;
```

```

public string _doors;

public int Go()
{
    'int speedMph = 100;
    return speedMph;
}
}

```

现在,假定重构时希望修改代码,把 `_color` 和 `_door` 变量封装到.NET 的公共属性中。Visual Studio 2010 的重构功能允许在文档窗口中右击这些属性,选择 **Refactor | Encapsulate Field** 命令。打开 **Encapsulate Field** 对话框,如图 16-28 所示。

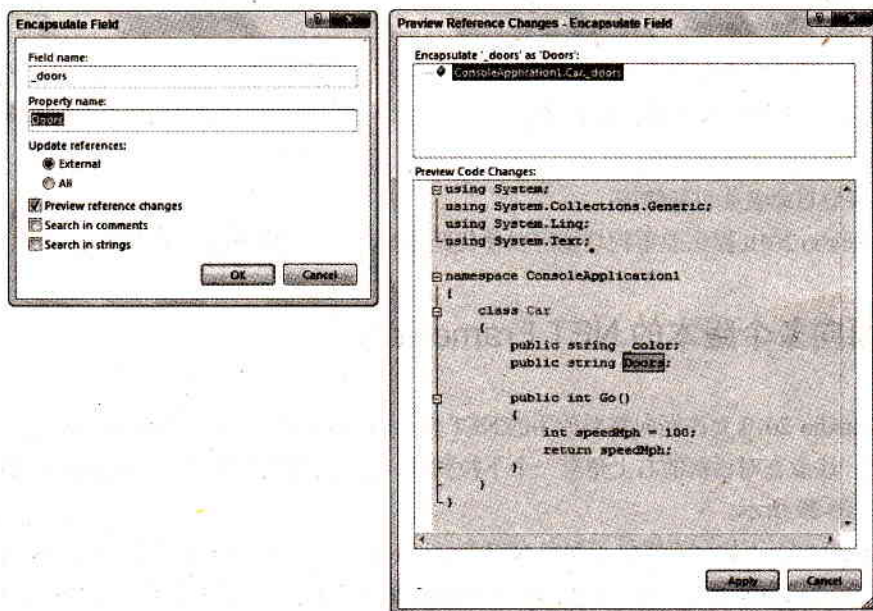


图 16-28

在这个对话框中,提供属性名,单击 **OK** 按钮,就把选中的公共字段转换为一个私有字段,并把它封装到一个.NET 公共属性中。单击 **OK** 按钮后,改写后代码如下所示(改写了两个字段):

```

namespace ConsoleApplication1
{
    public class Car
    {
        private string _color;

        public string Color
        {
            get { return _color; }
            set { _color = value; }
        }
        private string _doors;

        public string Doors
        {

```

```

    get { return _doors; }
    set { _doors = value; }
}

public int Go()
{
    int speedMph = 100;
    return speedMph;
}
}
}

```

可以看出, 这些向导不但能重构一个页面上的代码, 也能重构整个应用程序的代码。它们还可以完成如下任务:

- 给方法、局部变量、字段等重命名
- 从选中的代码中提取方法
- 根据一组已有的类型成员提取接口
- 把局部变量提升为参数
- 对参数重命名或重新排序

Visual Studio 2010 提供的重构功能可以使代码更简洁、可读性更高, 结构化更强。

16.3 面向多个版本的.NET Framework

Visual Studio 2010 允许面向需要使用的.NET Framework 版本。打开 New Project 对话框, 准备新建项目时, 注意在对话框的右上角有一个下拉列表, 它允许选择要使用的 Framework 版本。这个对话框如图 16-29 所示。

在图 16-29 中, 下拉列表提供了面向.NET Framework 2.0、3.0、3.5 和 4 的功能。使用升级对话框把 Visual Studio 2008 解决方案升级到 Visual Studio 2010 时, 重点是要理解, 只是把解决方案升级为使用 Visual Studio 2010, 而不是把项目升级到.NET Framework 4 上。项目仍在以前使用的.NET Framework 版本上工作, 但现在可以使用新的 Visual Studio 2010 处理项目了。

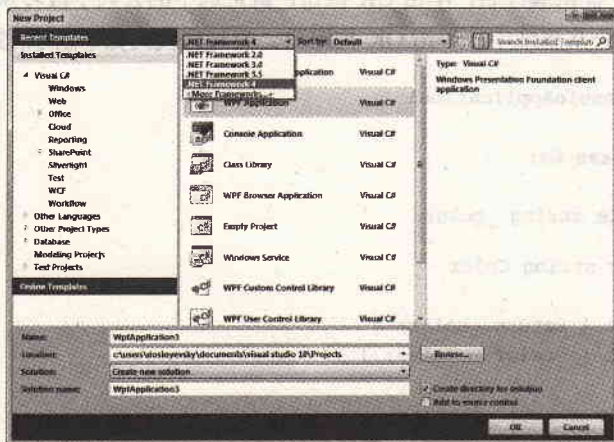


图 16-29

如果要改变解决方案使用的 .NET Framework 版本, 就可以右击该项目, 选择该解决方案的属性。如果处理的是 ASP.NET 项目, 就会打开如图 16-30 所示的对话框。

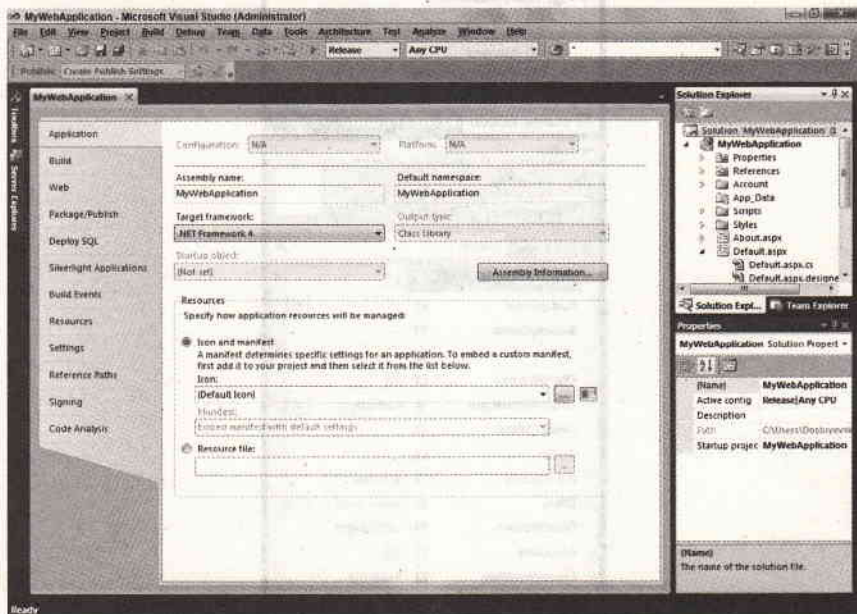


图 16-30

在这个对话框中, Application 选项卡提供了改变应用程序使用的 .NET Framework 版本的功能。

16.4 WPF、WCF、WF 等

在默认情况下, Visual Studio 2005 不允许构建面向 .NET Framework 3.0 的应用程序, 这在 VS2005 的生命周期中已过时了。Visual Studio 2005 的默认安装仅面向 .NET Framework 2.0。要开始使用面向 .NET Framework 3.0 的新技术, 需要安装另外几个软件包。

.NET Framework 3.0 允许使用类库来构建新的应用程序类型, 例如, 使用 Windows Presentation Foundation(WPF)、Windows Communication Foundation(WCF)、Windows Workflow Foundation(WF) 和 Windows CardSpace 的应用程序。

Visual Studio 2010 的面向 Framework 功能允许构建使用 .NET Framework 3.0、3.5 或 4 的这些应用程序类型。

16.4.1 在 Visual Studio 2010 中构建 WPF 应用程序

.NET Framework 4 给 Visual Studio 带来的一个优秀功能是增加了 WPF Application 项目类型(在 Windows 类别中)。选择这个项目类型会创建一个 MainWindow.xaml 文件和一个 MainWindow.xaml.cs 文件。这个项目类型在 Solution Explorer 窗口中默认创建的内容如图 16-31 所示(带有可搜索的 Properties 窗口)。

在 Visual Studio 2010 中, 可用的选项在文档窗口中。创建这个项目后, 文档窗口的默认视图如图 16-32 所示。



图 16-31

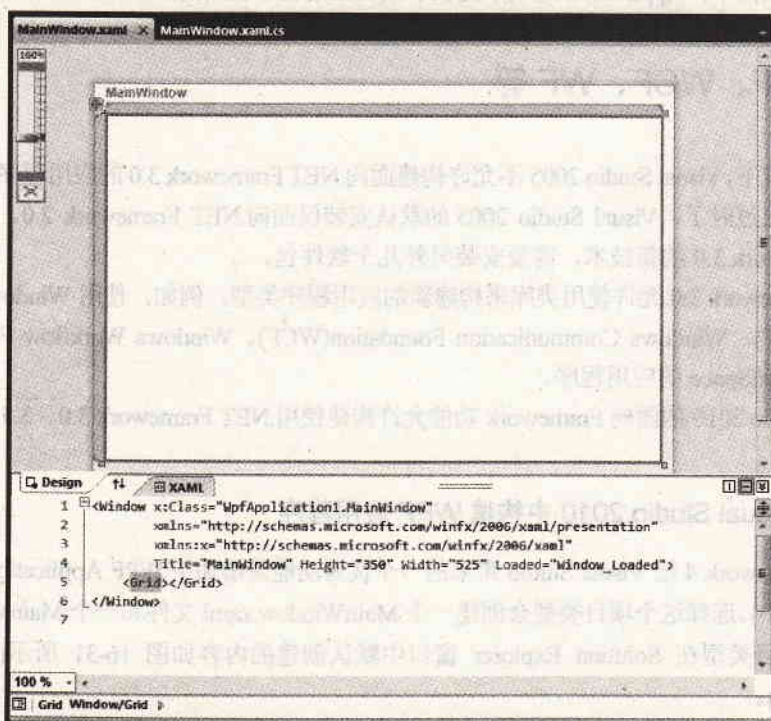


图 16-32

文档窗口有两个视图：设计视图和 XAML 视图。在设计视图中进行修改，会使 XAML 视图出现相应的变化，反之亦然。与传统的 Windows 窗体应用程序一样，WPF 应用程序也可以使用包含在 Visual Studio 的工具箱中的控件。控件对应的这个新工具箱如图 16-33 所示。

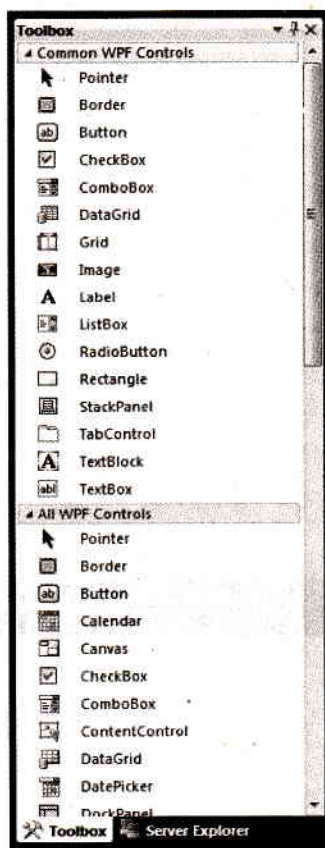


图 16-33

16.4.2 在 Visual Studio 2010 中构建 WF 应用程序

另一种完全不同的应用程序样式(在 Visual Studio 中构建应用程序时)是 Windows Workflow 应用程序类型。例如，从 New Project 对话框的 Workflow 部分中选择 Workflow Console Application 项目类型，就会创建一个控制台应用程序，它的 Solution Explorer 视图如图 16-34 所示。

在构建使用 Windows Workflow Foundation 的应用程序时，要注意它非常依赖于设计视图。仔细查看工作流(如图 16-35 所示)，会发现它包含多个顺序步骤，甚至包含基于条件(如 if-else 语句)的操作。



图 16-34

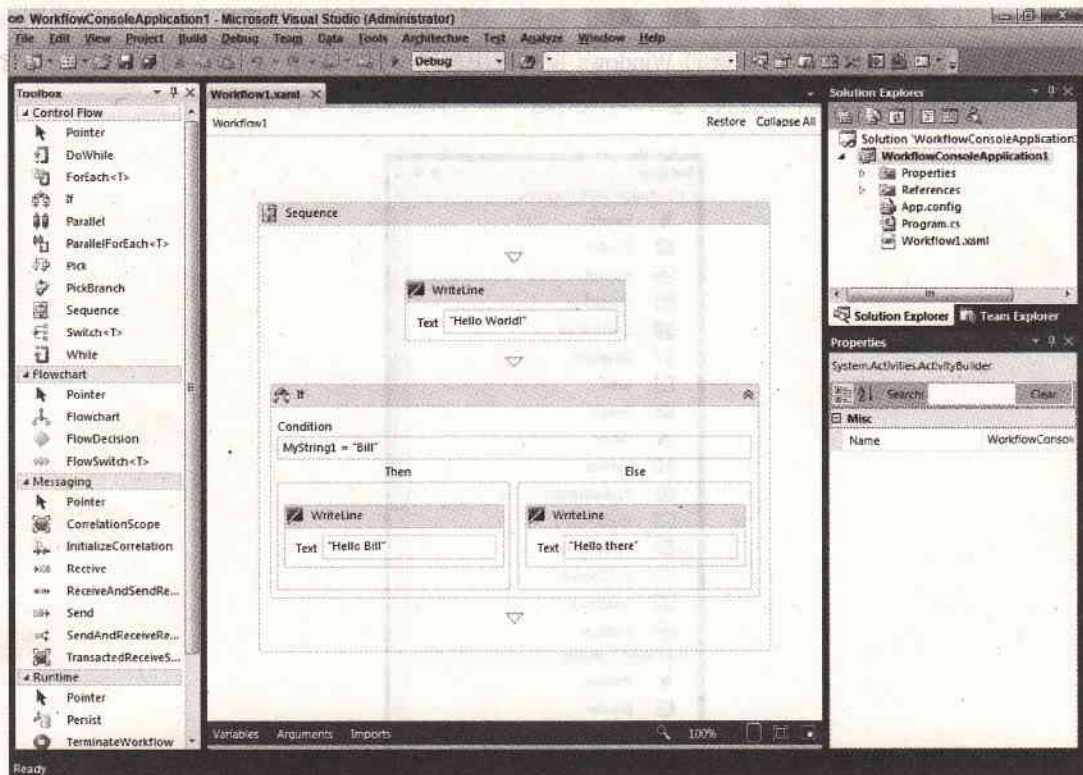


图 16-35

16.5 小结

本章介绍了.NET 环境中一个最重要的编程工具 Visual Studio 2010。本章的大部分内容说明如何使用这个工具编写 C# 代码(以及 C++ 和 Visual Basic 代码)。

Visual Studio 2010 是编程领域中最简单的开发环境之一。Visual Studio 很容易进行 RAD(Rapid Application Development, 快速应用程序开发), 同时还可以深入了解如何创建应用程序的机制。本章详细介绍了如何使用 Visual Studio 完成各种任务, 包括重构代码, 多目标、读取 Visual Studio 6 项目、调试程序, 以及许多可用于 Visual Studio 的窗口。

本章还介绍了可通过 .NET Framework 4 创建的最新项目。这些项目类型关注的是 Windows Presentation Foundation、Windows Communication Foundation 和 Windows Workflow Foundation。

第 17 章将详细讨论部署。

第 17 章

部 署

本章内容:

- 部署要求
- 简单的部署情况
- 基于 Windows 安装程序的项目
- ClickOnce 技术

编译源代码并完成测试后, 开发过程并没有结束。在这个阶段, 需要把应用程序提供给用户。无论是 ASP.NET 应用程序、智能客户端应用程序还是用 Compact Framework 构建的应用程序, 软件都必须部署到目标环境中。.NET Framework 使部署工作比以前容易得多, 因为不再需要注册 COM 组件, 也不需要编写新的注册表配置单元。

本章将介绍可用于应用程序部署的选项, 包括 ASP.NET 应用程序和智能客户端应用程序的部署选项。

17.1 部署的规划

部署常常是开发过程之后的工作, 如果不精心规划, 就可能会导致严重的问题。为了避免在部署过程中出错, 应在最初的设计阶段就对部署过程进行规划。任何部署问题, 如服务器的容量、桌面的安全性或从哪里加载程序集等, 都应从一开始就纳入设计, 这样部署过程才会比较顺利。

另一个必须在开发过程早期解决的问题是, 在什么环境下测试部署。应用程序代码的单元测试和部署选项的测试可以在开发人员的系统中进行, 而部署必须在类似于目标系统的环境中测试。这非常重要, 可以消除目标计算机上不存在的依赖项。例如, 第三方的库很早就安装在项目的开发人员的计算机上, 但目标计算机可能没有安装这个库。在部署软件包中它很容易忘记包含这个库。在开发人员的系统上进行的测试不可能发现这个错误, 因为库已经存在了。归档依赖关系可以帮助消除这种潜在的错误。

部署过程对于大型应用程序可能非常复杂。提前规划部署, 在实现部署过程时可以节省时间和精力。

选择合适的部署选项, 必须像开发系统的其他方面那样特别关注和认真规划。选择错误的选项会使把软件交付给用户的过程充满艰难险阻。

17.1.1 部署选项

本节概述.NET 开发人员可以使用的部署选项。其中大多数选项将在本章后面详细论述。

- **Xcopy**——Xcopy 实用工具允许把一个程序集或一组程序集复制到应用程序文件夹中,从而减少了开发时间。由于程序集是自我包含的(即,元数据描述了包含在程序集中的内容)因此不需要在注册表中注册。

每个程序集都跟踪它需要执行的其他程序集。默认情况下,程序集会在当前的应用程序文件夹中查找依赖项。把程序集移动到其他文件夹的过程将在本章后面讨论。

- **发布 Web 站点**——在发布 Web 站点时,会编译整个站点,然后把它复制到指定的位置。预编译之后,所有的源代码都会从最终的输出中删除,找出和处理所有编译错误。
- **部署项目**——Visual Studio 2010 可以为应用程序创建安装程序。基于 Microsoft Windows Installer 技术有 4 种选择:
 - 创建合并模块;
 - 为客户端应用程序创建安装程序;
 - 为 Web 应用程序创建安装程序;
 - 为基于智能设备(Compact Framework)的应用程序创建安装程序。
- 还可以创建 cab 文件。部署项目为安装过程提供了极大的灵活性和可定制性。这 4 种部署方式对于大型应用程序都十分有用。
- **ClickOnce 技术**——ClickOnce 技术可以构建自动升级的、基于 Windows 的应用程序。ClickOnce 允许把应用程序发布到 Web 站点、文件共享、甚或 CD 上。在对应用程序进行升级并生成新版本后,开发小组可以把它们发布到相同的位置或站点上。最终用户在使用应用程序时,程序会检查是否有更新版本,如果有,就进行更新。

17.1.2 部署要求

最好看一下基于.NET 的应用程序的运行要求。在执行任何托管的应用程序之前,CLR 对目标平台都有一定的要求。

首先必须满足的要求是操作系统。目前,下面的操作系统可以运行基于.NET 的应用程序:

- Windows XP SP3
- Windows Vista SP1
- Windows 7

其次,必须支持下面的服务器平台:

- Windows 2003 Server SP2 系列
- Windows 2008 Server 系列(不支持 Server Core Role)

再次,必须支持下面的体系结构:

- x86
- x64
- ia64(一些特性不支持)

其他要求有 Windows Internet Explorer 5.01 版本或更高版本,MDAC 2.8 版本或更高版本(如果应

用程序需要访问数据)和用于 ASP.NET 应用程序的 Internet 信息服务(Internet Information Services, IIS)。

在部署 .NET 应用程序时,还必须考虑硬件要求。硬件的最低要求是:客户端和服务器都是奔腾 400MHz, 且有 96MB RAM。

要获得最佳性能,应增加 RAM, RAM 越大, .NET 应用程序运行得就越好。对于服务器应用程序更是如此。

17.1.3 部署 .NET 运行库

使用 .NET 开发应用程序时,需要依赖 .NET 运行库。这似乎很明显,但有时可以忽略这一点。表 17-1 列出了必须发布的版本号和文件名:

表 17-1

.NET 版本	文 件 名
2.0.50727.42	dotnetfx.exe
3.0.4506.30	Dotnetfx3.exe(包括 x86 和 x64)
3.5.21022.8	Dotnetfx35.exe(包括 x86、x64 和 ia64)
4.0.0.0	Dotnetfx40.exe(包括 x86、x64 和 ia64)

17.2 简单的部署选项

如果在应用程序的初始设计阶段考虑了部署,部署就只是简单地把一组文件复制到目标计算机上。对于 Web 应用程序,就只需使用 Visual Studio 2010 中的一个菜单选项。本节就讨论这种简单的部署情况。

为了了解如何设置各种部署选项,必须有一个要部署的应用程序。从 www.wrox.com 上下载的示例包含 4 个项目:

- ClientWinForms
- ClientWPF
- WebClient
- AppSupport

ClientWinForms 和 ClientWPF 项目是分别使用 WinForms 和 WPF 的智能客户端应用程序,WebClient 项目是一个简单的 Web 应用程序,AppSupport 项目是一个类库,它包含一个简单的类,该类返回一个包含当前日期和时间的字符串。

示例应用程序使用 AppSupport 项目用一个包含当前日期的字符串填写一个标签。为了使用这些示例,首先加载并构建 AppSupport 项目。然后在其他两个应用程序中,设置对新构建的 AppSupport.dll 的引用。

下面是 AppSupport 程序集的代码:



可从
wrox.com
下载源代码

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

```

namespace AppSupport
{
    public class DateService
    {
        public string GetLongDateInfoString()
        {
            return string.Concat("Today's date is ", DateTime.Now.ToLongDateString());
        }

        public string GetShortDateInfoString()
        {
            return string.Concat("Today's date is ", DateTime.Now.ToShortDateString());
        }
    }
}

```

代码段 Dateservice.cs

这个简单的程序集足以说明可用的部署选项。

17.2.1 Xcopy 部署

Xcopy 部署就是把一组文件复制到目标计算机上的一个文件夹中，然后在客户端上执行应用程序。这个术语来自于 DOS 命令 `xcopy.exe`。无论程序集的数目是多少，如果把文件复制到同一个文件夹中，应用程序就会执行，不需要编辑配置设置或注册表。

为了理解 Xcopy 部署的工作原理，执行下面的步骤：

- (1) 打开示例下载文件中的 ClientWinForms 解决方案(ClientWinForms.sln)。
- (2) 把目标改为 Release，进行完整的编译。
- (3) 然后，使用“我的电脑”或文件管理器导航到项目文件夹\ClientWinForms\bin\Release，双击 ClientWinForms.exe，运行应用程序。

(4) 现在单击对应的按钮，会看到当前日期显示在两个文本框中。这将验证应用程序是否能正常运行。当然，这个文件夹是 Visual Studio 放置输出的地方，所以应用程序能正常工作。

(5) 新建一个文件夹，命名为 ClientWinFormsTest。把这两个文件从 Release 文件夹复制到这个新文件夹中，然后删除 Release 文件夹。再次双击 ClientWinForms.exe 文件，验证它是否正常工作。

这就是它的所有工作；Xcopy 部署只需把程序集复制到目标计算机上，就可以部署功能完善的应用程序。这里使用的示例非常简单，但这并不意味着这个过程对较复杂的应用程序无效。实际上，使用这种方法对可以部署的程序集的大小或数目没有限制。不想使用 Xcopy 部署的原因是它不能把程序集放在全局程序集缓存(GAC)中，或者不能在“开始”菜单中添加图标。如果应用程序仍依赖于某种类型的 COM 库，就不能很容易地注册 COM 组件。

17.2.2 Xcopy 和 Web 应用程序

Xcopy 部署也可以用于 Web 应用程序，但文件夹结构有点不同。我们必须建立 Web 应用程序的虚拟目录，并配置合适的用户权限。这个过程通常需要使用 IIS 管理工具来完成。

在建立虚拟目录后，Web 应用程序文件就可以复制到虚拟目录中。复制 Web 应用程序的文件有点困难，需要考虑几个配置文件和页面使用的图像。

17.2.3 发布 Web 站点

Web 项目的另一个部署选项是发布 Web 站点。发布 Web 站点就是预编译整个站点，并把编译好的版本放在指定的位置。该位置可以是共享文件、FTP 位置，或可以通过 HTTP 访问的任何其他位置。编译过程会从程序集中去除所有的源代码，为部署创建 DLL。这也包括 ASPX 源文件中的标记。ASPX 文件并不包含一般的标记，而是包含程序集的一个指针。每个 ASPX 文件都与一个程序集相关。无论使用代码隐藏模型还是单个文件模型，这个过程都会正常工作。

发布 Web 站点的优点是速度快，安全性高。速度有所提高，是因为所有的程序集都已编译。否则，第一次访问页面时会会有一个延迟，因为要编译和缓存页面和依赖代码。安全性有所提高，是因为不部署源代码。另外，在部署前所有的源代码都进行了预编译，找出了所有的编译错误。

使用 Build | Publish Web Site 菜单项就可以发布 Web 站点。在 Publish Method 下拉菜单中，可以选择 MSDeploy Publish、FTP 或文件系统。FTP 选项要求指定 FTP 地址和必要的登录凭证。文件系统选项要求指定目标位置的路径。MSDeploy Publish 选项比较有趣。

可以在项目属性页面的 Package and Publish 选项卡中给 Web 站点的部署定义许多属性。可以指定应包含什么文件，包括调试信息。还可以在 Deploy Sql 选项卡中包含数据库信息。这些信息包括连接字符串、模式信息和数据库脚本选项。

也可以在这里选择打包选项。之后会生成包含安装 Web 站点所需所有内容的 zip 文件。除了 zip 文件之外，还会生成另外 3 个文件：

- Projectname.deploy.cmd——MSDeploy 用于安装 Web 站点的命令文件。
- Projectname.SetParameters.xml——这个 XML 文件包含了传递给 Web 部署工具的不同参数，可用于部署到不同的服务器或不同的环境中。
- Projectname.SourceManifest.xml——Visual Studio 2010 创建软件包所使用的设置。

Projectname 是要打包的项目名。软件包可以使用 Visual Studio 或通过 MSBuild 来部署。

17.3 Visual Studio 2010 安装和部署项目

Xcopy 部署使用起来很简单，但有时它缺乏一些功能。为了克服这个缺点，Visual Studio 2010 提供了 6 个安装程序项目类型。其中 4 个类型基于 Windows Installer 技术，表 17-2 列出了这些项目类型。

表 17-2

项 目 类 型	说 明
Setup Project	用于安装客户端应用程序、中间层应用程序和作为 Windows 服务运行的应用程序
Web Setup Project	用于安装基于 Web 的应用程序
Merge Module Project	创建.msm 合并模块，这些模块可以和其他基于 Windows Installer 的安装应用程序一起使用
CAB Project	创建.cab 文件，通过旧式的部署技术进行发布
Setup Wizard	帮助创建部署项目
Smart Device CAB Project	Pocket PC、Smartphone 和其他基于 CE 的应用程序的 CAB 项目

Setup Project 和 Web Setup Project 非常相似。主要的区别是使用 Web Setup, 项目会部署到 Web 服务器上的一个虚拟目录中, 而使用 Setup Project, 项目会部署到文件夹结构中。这两个项目类型都基于 Windows Installer, 拥有基于 Windows Installer 的安装程序的所有功能。

在创建包含在多个部署项目中的组件或功能库时, 一般使用 Merge Module Project。通过创建合并模块, 可以设置专用于组件的配置项, 而无需在主部署项目的创建过程中担心它们。

Cab Project 类型仅为应用程序创建.cab 文件。.cab 文件由旧式的安装技术以及一些基于 Web 的安装过程使用。Setup Wizard 项目类型逐步完成创建部署项目的过程, 在此过程中向用户询问特定的问题。

下面几节讨论如何创建这些部署项目, 可以改变哪些设置和属性, 以及可以增加什么定制内容。

17.3.1 Windows Installer

Windows Installer 是一个服务, 它负责管理在大多数 Windows 操作系统上安装、更新、修复和删除应用程序。它是 Windows ME、Windows 2000、Windows XP、Windows Vista 和 Windows 7 的一部分, 可以用于 Windows 95、Windows 98 和 Windows NT 4.0。Windows Installer 的当前版本是 4.5。

Windows Installer 在数据库中跟踪应用程序的安装。在卸载应用程序时, Windows Installer 很容易跟踪和删除已添加的注册表设置、复制到硬盘上的文件, 以及已添加的桌面图标和“开始”菜单图标。如果某个特定文件仍被另一个应用程序引用, 安装程序就会把它保留在硬盘上, 从而不会使其他的应用程序中断。数据库还可以修复应用程序。如果注册表设置或与应用程序相关的 DLL 被破坏或不小心中除了, 就可以修复安装。在修复过程中, 安装程序会从上一次安装中读取数据库, 并重制该安装过程。

Visual Studio 2010 中的部署项目可以创建 Windows 安装软件包。部署项目允许访问大多数需要访问的内容, 以便安装给定的应用程序。但是, 如果需要更多的控制, 就应查看 Windows Installer SDK, 它是 Platform SDK 的一部分, 其中包含了为应用程序创建自定义安装软件包的文档。下面几节将使用 Visual Studio 2010 部署项目创建这些安装软件包。

17.3.2 创建安装程序

为客户端应用程序或 Web 应用程序创建安装软件包并不困难。第一个任务是标识应用程序需要的所有外部资源, 包括配置文件、COM 组件、第三方库、控件和图像。前面说过, 在项目文档中应包含一个依赖项列表。这正是这个文档的用武之地。Visual Studio 2010 可以询问程序集, 检索该程序集的依赖项, 但我们仍需要审核这些内容, 以防遗漏。

另一个问题是, 在整个过程的什么时候创建安装软件包。如果设置了一个自动构建过程, 就可以把安装软件包的构建包含在项目成功构建的过程中。在耗时而复杂的大型项目中, 过程的自动进行会大大减少出错的可能性。

我们可以把部署项目包含在项目解决方案中。在 Solution Property Pages 对话框中有针对 Configuration Properties 的一个设置。使用这个设置可以选择要为各种构建配置包含的项目。如果选择 Release builds 而不是 Debug builds 下面的 Build 复选框, 安装软件包就只在创建发布版本时创建。这也是在下面示例中使用的过程。图 17-1 显示了 ClientWPF 解决方案的 Solution Property Pages 对话框。其中显示了 Debug 配置, 没有给安装项目选中 Build 复选框。

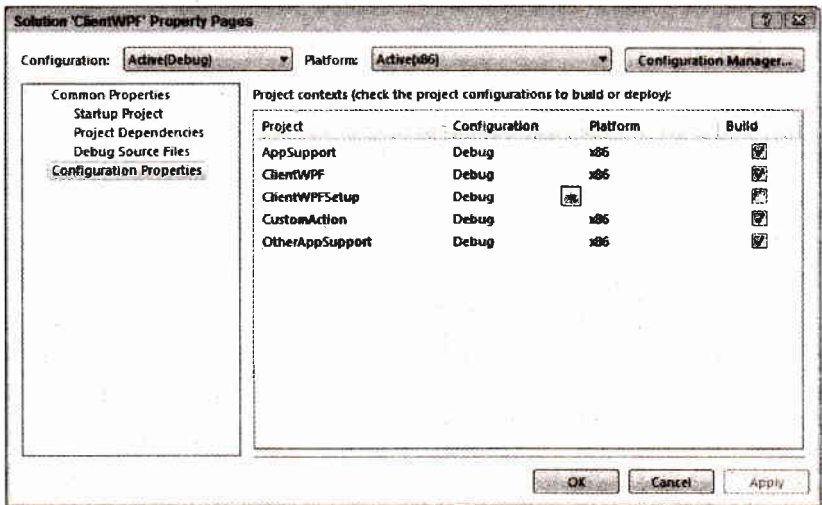


图 17-1

1. 简单的客户端应用程序

在下面的示例中，要为 ClientWinForms 解决方案创建一个安装程序(它包含在下载示例中，其中还包含已完成的安装程序项目)。这个示例将说明如何在独立的解决方案中创建安装程序项目。后面将介绍在原来的解决方案中包含安装程序项目的 ClientWPF。

在开始创建部署项目之前，要确保部署的应用程序有一个发布版本。

接着，在 Visual Studio 2010 中新建一个项目。在 New Project 对话框中，选择左边的 Setup and Deployment Projects，再选择右边的 Setup Project，给它指定一个名称(例如，ClientWinFormsSetup)。

在 Solution Explorer 窗口中，单击项目，再单击 Properties 窗口，就会看到一个属性列表。这些属性将在应用程序的安装过程中显示。其中一些属性还会显示在控制面板中“添加/删除程序”对话框对应的小程序中。因为用户在安装过程中可以看到大多数属性(或者在“添加/删除程序”窗口中查看安装过程时可以看到它们)，所以正确地设置它们会使应用程序更专业。

这个属性列表非常重要，特别是在应用程序要进行商业化部署时，更是如此。表 17-3 描述了这些属性和要输入的值。

表 17-3

项 目 属 性	说 明
AddRemoveProgramIcon	显示在“添加/删除程序”对话框中的图标
Author	应用程序的编写者。这个属性设置通常与 manufacturer 的相同，它显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Contact 字段中
Description	这是一个形式自由的文本字段，描述要安装的应用程序或组件。这些信息显示在 msi 软件包的 Properties 对话框中的 Summary 页面上，以及“添加/删除程序”对话框的 SupportInfo 页面上的 Comment 字段中

(续表)

项 目 属 性	说 明
DetectNewerInstalledVersion	这是一个布尔值, 设置为 <code>true</code> 时, 将检查是否已安装了应用程序的更新版本, 如果是, 就停止安装过程
InstallAllUsers	这是一个布尔值, 设置为 <code>true</code> 时, 将为计算机的所有用户安装应用程序。设置为 <code>false</code> 时, 就只有当前用户能访问应用程序
Keywords	可用于在目标计算机上搜索 <code>.msi</code> 文件。这些信息显示在 <code>msi</code> 软件包的 <code>Properties</code> 对话框中的 <code>Summary</code> 页面上
Localization	用于字符串资源和注册表设置的地域。这会影响安装程序的用户界面
Manufacturer	生产应用程序或组件的公司的名称。一般与 <code>Author</code> 属性中指定的信息相同。这些信息显示在 <code>msi</code> 软件包的 <code>Properties</code> 对话框中的 <code>Summary</code> 页面上, 以及“添加/删除程序”对话框的 <code>SupportInfo</code> 页面上的 <code>Publisher</code> 字段中, 用作应用程序的默认安装路径的一部分
ManufacturerURL	一个 Web 站点的 URL, 该站点与要安装的应用程序或组件相关
PostBuildEvent	在构建结束后执行的命令
PreBuildEvent	在构建开始前执行的命令
ProductCode	应用程序或组件的唯一的字符串 GUID。Windows Installer 使用这个属性标识应用程序的后续升级或安装
ProductName	描述应用程序的名称, 在“添加/删除程序”对话框中用作应用程序的描述, 还用作默认安装路径的一部分: <code>C:\Program Files\Manufacturer\ProductName</code>
RemovePreviousVersions	一个布尔值, 如果设置为 <code>true</code> , 就检查计算机上是否安装了应用程序的以前版本。如果是, 就调用以前版本的卸载功能, 之后继续安装。这个属性使用 <code>ProductCode</code> 和 <code>UpgradeCode</code> 确定是否卸载。 <code>UpgradeCode</code> 应相同, 而 <code>ProductCode</code> 应不同
RunPostBuildEvent	运行 <code>PostBuildEvent</code> 的时间, 其选项有 <code>On successful build</code> 或 <code>Always</code>
SearchPath	一个字符串, 表示依赖的程序集、文件或合并模块的搜索路径。在开发人员的计算机上构建安装程序对应的软件包时使用该属性
Subject	与应用程序相关的其他信息。这些信息显示在 <code>msi</code> 软件包的 <code>Properties</code> 对话框中的 <code>Summary</code> 页面上
SupportPhone	为应用程序或组件提供支持的电话号码。这些信息显示在“添加/删除程序”对话框的 <code>SupportInfo</code> 页面上的 <code>Support Information</code> 字段中
SupportURL	为应用程序或组件提供支持的 URL。这些信息显示在“添加/删除程序”对话框的 <code>SupportInfo</code> 页面上的 <code>Support Information</code> 字段中
TargetPlatform	支持 Windows 的 32 或 64 位版本
Title	安装程序的标题。它显示在 <code>msi</code> 软件包的 <code>Properties</code> 对话框中的 <code>Summary</code> 页面上
UpgradeCode	一个字符串 GUID, 表示同一应用程序的不同版本共享的标识符。对于应用程序的不同版本或不同语言版本, 不应修改 <code>UpgradeCode</code> , 该属性由 <code>DetectNewerInstalledVersion</code> 和 <code>RemovePreviousVersions</code> 使用
Version	安装程序、 <code>.cab</code> 文件或合并模块的版本号。注意它不是要安装的应用程序的版本号

设置完属性后,就可以开始添加程序集了。在这个示例中,唯一要添加的程序集是主可执行文件 `ClientWinForms.exe`。为此,可以在 **Solution Explorer** 窗口中右击项目,或从 **Project** 菜单中选择 **Add** 命令,此时有 4 个选项:

- **Project Output**——下一个示例探讨这个选项
- **File**——用于添加 `readme` 文本文件或不在构建过程中添加的任何其他文件
- **Merge Module**——独立创建的合并模块
- **Assembly**——使用这个选项可以选择要安装的程序集

本例选择 **Assembly**, 打开 **Component Selector** 对话框, 该对话框类似于给项目添加引用的对话框。浏览至应用程序的 `\bin\release` 文件夹, 选择 `ClientWinForms.exe`, 在 **Component Selector** 对话框中单击 **OK** 按钮, 现在可以看到 `ClientWinForms.exe` 在部署项目的 **Solution Explorer** 窗口中。在 **Detected Dependencies** 部分, 可以看到 **Visual Studio** 要求 `ClientWinForms.exe` 给出它依赖的程序集。在本例中, 会自动包括 `AppSupport.dll`。继续这个过程, 直到应用程序中的所有程序集都显示在部署项目的 **Solution Explorer** 窗口中为止。

接着, 需要确定把程序集部署到什么地方。在默认情况下, 在 **Visual Studio 2010** 中会显示文件系统编辑器, 这个文件系统编辑器分为两个窗格, 左边的窗格显示目标计算机上文件系统的层次结构, 右边的窗格则显示选中文件夹的详细视图。文件夹名称可能不是我们希望看到的, 但这些文件夹用于目标计算机, 例如, 文件夹 `User's Programs Menu` 映射为目标客户端上包含用户的 `Programs Menu` 的文件夹, 它随着所使用的 **Windows** 版本的不同而不同。

此时可以添加其他文件夹, 或者特定的文件夹或自定义文件夹。要添加特定的文件夹, 应确保目标计算机上的文件系统在左边的窗格上突出显示, 然后选择主菜单中的 **Action** 菜单。 **Add Special Folder** 菜单选项提供了可以添加的文件夹列表。例如, 如果要在 **Application** 文件夹下添加一个文件夹, 就可以在编辑器的左边窗格中选择 **Application** 文件夹, 再选择 **Action** 菜单。这时就会出现一个可以新建文件夹的 **Add** 菜单。给新文件夹重命名, 就会在目标计算机上创建它。

我们要添加的一个特定文件夹是 **GAC** 文件夹。如果有几个不同的应用程序使用 `AppSupport.dll`, 就可以把它安装到 **GAC** 文件夹中。为了把程序集添加到 **GAC** 文件夹中, 程序集必须有一个强名。把程序集添加到 **GAC** 文件夹的过程就是从 **Special Folder** 菜单中添加 **GAC** 文件夹, 再把要放在 **GAC** 文件夹中的程序集从当前文件夹拖放到 **Global Assembly Cache** 文件夹中。如果试图对没有强名的程序集进行这个操作, 部署项目就不能编译。

如果选择 **Application** 文件夹, 在右边的窗格上, 刚才添加的程序集就会自动添加到 **Application** 文件夹中。还可以把程序集移动到其他文件夹中, 但程序集必须能找到对方(有关探测的更多信息请参阅第 18 章)。

如果要在用户的桌面上或“开始”菜单中添加应用程序的快捷方式, 就应把该快捷方式拖放到适当的文件夹中。要创建桌面快捷方式, 就应进入 **Application** 文件夹, 在编辑器的右边窗格上选择该应用程序, 再进入 **Action** 菜单, 选择 **Create Shortcut** 菜单项, 创建应用程序的快捷方式。在创建好快捷方式后, 把它拖放到 `User's Desktop` 文件夹中。现在安装应用程序时, 快捷方式就会显示在桌面上。一般情况下, 应由用户决定是否需要应用程序的桌面快捷方式。

要求用户输入信息并执行相应步骤的过程将在本章后面介绍。在“开始”菜单中创建菜单项的过程与此相同。另外, 如果查看刚才创建的快捷方式的属性, 就可以配置快捷方式的基本属性, 如参数和要使用的图标。应用程序图标是默认图标。

在构建部署项目之前，需要检查一些项目属性。如果选择 **Project** 菜单，再选择 **ClientWinFormsSetup Properties** 命令，就会打开 **Project Property Pages** 对话框，其中的属性是针对当前配置的。在 **Configuration** 下拉框中选择配置后，就可以修改表 17-4 中的属性。

表 17-4

属 性	说 明
Output filename	在编译项目时生成的.msi 或.msm 文件的名称
Package files	这个属性允许指定如何打包文件。其选项有： As loose uncompressed files——所有的部署文件都在同一个目录下存储为.msi 文件； In setup file——文件打包到.msi 文件中(默认设置)； In cabinet files——文件打包到同一目录下的一个或多个.cab 文件中。选择这个选项时，CAB size 选项就变成可用的
Prerequisites URL	可以指定在哪里查找.NET Framework 或 Windows Installer 等必备的程序。单击 Settings 按钮会显示一个对话框，其中包含安装过程中可以使用的列表项，例如 Windows Installer 4.5、.NET Framework 和 SQL Express 2008。还有一个选项，可以选择从哪里下载必备的程序
Compression	这个属性指定所包含文件的压缩样式。其选项如下： Optimized for speed——用于较大的文件，安装时间较短(默认设置) Optimized for size——用于较小的文件，安装时间较长 None——不压缩
CAB size	Package file 属性设置为 In cabinet files 时，就启用这个属性。它不仅可以创建一个 CAB 文件，还可以设置每个.cab 文件占用的最大内存
Authenticode signature	在选中这个属性时，部署项目输出的签名就使用 Authenticode 标记，不选中默认设置

在设置完项目属性后，就应构建部署项目，为 **ClientWinForms** 应用程序创建安装程序。在构建项目后，就可以在 **Solution Explorer** 窗口中右击项目名，测试安装程序了，此时可以在弹出的上下文菜单中访问 **Install** 和 **UnInstall** 选项。如果一切正常，就可以成功安装和卸载 **ClientWinForms** 应用程序。

2. 同一个解决方案项目

上一个示例成功地创建了一个部署软件包，但有几个缺点。例如，在新程序集添加到原始应用程序中时会发生什么情况？部署项目不会自动识别任何改动的地方，而必须添加新程序集，再验证新的依赖项已包含进来。在较小的应用程序(如本例)中，这没有什么大不了。

但在处理包含几十个甚至上百个程序集的应用程序时，这就可能很难维护了。**Visual Studio 2010** 为解决这个潜在的问题提供了一个简单的方法，即把部署项目包含在应用程序的解决方案中，这样就可以把主项目的输出当作部署程序集了。下面以 **ClientWPF** 为例来说明。

在 **Visual Studio 2010** 中打开 **ClientWPF** 解决方案，使用 **Solution Explorer** 窗口添加一个新项目，选择 **Deployment and Setup Projects**，再选择 **Setup Project**，之后按照上一节介绍的步骤进行。可以把这个项目命名为 **ClientWPFSetup**。在前面的示例中，通过从 **Project** 菜单中选择 **Add | Assemblies** 命

令；添加程序集，这次从 Project 菜单中选择 Add | Project Output 命令，这会打开 Add Project Output Group 对话框，如图 17-2 所示。

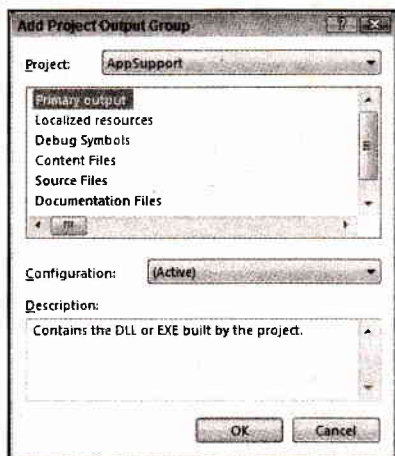


图 17-2

对话框的顶部有一个下拉列表框，其中显示了当前解决方案中的所有项目。选择主启动项目，然后从下面的列表中选择要包含在项目中的项，选项有 Documentation、Primary Output、Localized resources、Debug Symbols、Content Files 和 Source Files。

首先选择 **Primary Output**，这包括构建应用程序时的输出和所有依赖项。对话框中还有一个下拉列表框，其中列出了有效的配置：**Debug**、**Release** 以及自己添加的任何自定义配置。这还确定了选取什么输出。对于部署，应使用 **Release** 配置。

在完成这些选择后，在 **Solution Explorer** 窗口中就会给部署项目添加一个新条目。该条目的名称是 **Primary output form ClientWPF (Release .NET)**。另外，**AppSupport.dll** 文件将出现在依赖项列表中。与以前一样，不需要搜索依赖程序集。

此时，仍在应用上一节讨论的所有不同的项目属性。可以修改 **Name**、**Manufacturer**、**.cab** 文件大小和其他属性。在设置好属性后，建立解决方案的发布版本，并测试安装程序。一切应如期正常工作。

为了理解把部署软件包添加到应用程序解决方案中的优点，下面把一个新项目添加到解决方案中。在本例中该项目名为 **OtherAppSupport**。在该项目中，有一个简单的测试方法，它返回字符串 **"This is the message"**。在 **ClientWPF** 中设置一个对新增项目的引用，建立解决方案的另一个发布版本。部署项目会自动选择新的程序集，无需我们做任何工作。还可以右击安装程序项目的 **Detected Dependencies** 部分，选择 **Refresh Dependencies** 命令。如果使用独立的部署项目，就不会选取新添的项目。

3. 简单的 Web 应用程序

为 Web 应用程序创建安装软件包与创建客户端安装软件包没有什么不同。下载的示例包含一个 **WebClient** 应用程序，它也利用了 **AppSupport.dll** 程序集。可以用与创建客户端部署项目相同的方式创建部署项目，即创建独立的部署项目，或在原解决方案中创建部署项目。这个示例在原解决方案中构建部署项目。

启动 **WebClient** 解决方案，添加一个新的 **Deployment and Setup** 项目。这次要确保在 **Templates**

窗口中选择 **Web Setup Project**。如果查看该项目的属性视图，就会看到 Web 应用程序拥有与客户端应用程序相同的所有属性。唯一新增的属性是 **RestartWWWService**。这是一个布尔值，用于在安装过程中重新启动 IIS。如果使用 ASP.NET 组件，而且没有替换某些 ATL 或 ISAPI DLL，就不需要修改该属性。

如果查看文件系统编辑器，就会注意其中只有一个文件夹。**Web Application** 文件夹就是我们的虚拟目录。默认情况下，目录名就是部署项目名，它位于 Web 根目录下。表 17-5 解释了可以在安装程序中设置的属性。上一节讨论的属性未包括在内。

表 17-5

属 性	说 明
AllowDirectoryBrowsing	布尔值，如果设置为 true ，就允许以 HTML 格式列出虚拟目录中的文件和子文件夹。该属性映射到 IIS 的 Directory Browsing 属性
AllowReadAccess	布尔值，如果设置为 true ，就允许用户读取或下载文件。该属性映射到 IIS 的 Read 属性
AllowScriptSourceAccess	布尔值，如果设置为 true ，就允许用户访问源代码，包括脚本。该属性映射到在 IIS 中访问的 Script 资源
AllowWriteAccess	布尔值，如果设置为 true ，就允许用户修改可写文件中的内容。映射到 IIS 的 Write 属性
ApplicationProtection	确定运行在服务器上的应用程序的保护级别。有效值如下： Low ——应用程序与 Web 服务运行在同一个进程中 Medium ——应用程序运行在同一个进程中，但不与 Web 服务运行在同一个进程中 High ——应用程序运行在它自己的进程中 该属性映射到 IIS 的 Application Protection 属性。如果 IsApplication 属性为 false ，则不起作用
AppMappings	列出应用程序名以及与应用程序相关的文档或数据文件。该属性映射到 IIS 的 Application Mappings 属性
Condition	Windows Installer 条件，必须满足该条件，才能安装需要的项
DefaultDocument	用户第一次浏览站点时的默认文档或启动文档
ExecutePermissions	用户执行应用程序必须拥有的许可级别。有效值如下： None ——只能访问静态内容 ScriptOnly ——只能访问脚本，包括 ASP ScriptAndExecutables ——可以访问所有文件 该属性映射到 IIS 的 Execute Permissions 属性
Index	布尔值，如果设置为 true ，就允许给 Microsoft Indexing 服务的内容建立索引。该属性映射到 IIS 的 Index this resource 属性
IsApplication	布尔值，如果设置为 true ，就让 IIS 为文件夹创建应用程序根目录
LogVisits	布尔值，如果设置为 true ，就可以把对 Web 站点的访问记录到日志文件中。该属性映射到 IIS 的 Log visits 属性
Property	可以在安装期间访问的指定属性
VirtualDirectory	应用程序的虚拟目录，这相对于 Web 服务器

注意,大多数属性都是 IIS 的属性,可以在 IIS 管理工具中设置。所以有如下逻辑假设:为了在安装程序中设置这些属性,安装程序在运行时需要拥有管理员权限。这里进行的设置可能会降低安全性,所以对做出的修改要进行很好的归档。

除了这些属性之外,创建部署软件包的过程非常类似于前面的客户示例。两个项目的主要区别是允许在安装过程中修改 IIS。可以看出,我们对 IIS 环境有很大的控制权。

4. Web 服务器上的客户

另一个安装情况是从 Web 站点上运行安装程序,或从 Web 站点上运行应用程序。如果必须把应用程序部署给大量用户,这就是两个很有吸引力的选项。从 Web 站点上部署,就不需要分配介质,如 CD-ROM、DVD,甚或软盘。从 Web 站点甚至共享网络上运行应用程序,就根本不需要发布安装程序。

从 Web 站点上运行安装程序相当简单,使用 Web Bootstrapper 项目编译选项即可,此时需要提供安装文件夹的 URL,在这个文件夹中,安装程序会查找需要的.msi 和其他必要的文件。设置好这个选项,并编译部署软件包后,就可以把它复制到在 Setup folder URL 属性中指定的 Web 站点中。这样当用户导航到这个文件夹时,就可以运行安装程序,或者先下载再运行它。在这两种情况下,用户都必须连接到同一个站点,才能完成安装。

17.4 ClickOnce

ClickOnce 是一种允许应用程序自动升级的部署技术。应用程序发布到共享文件、Web 站点或 CD 这样的媒介上。之后,ClickOnce 应用程序就可以自动升级,而无需用户的干涉。

ClickOnce 还解决了安全权限问题。一般情况下,要安装应用程序,用户需要有管理权限。而利用 ClickOnce,用户只要有运行应用程序所需的最低绝对权限,就可以安装和运行应用程序。

17.4.1 ClickOnce 操作

ClickOnce 应用程序有两个基于 XML 的清单文件,其中一个是应用程序的清单,另一个是部署清单。这两个文件描述了部署应用程序所需的所有信息。

应用程序清单包含应用程序的相关信息,例如,需要的权限、要包括的程序集和其他依赖项。部署清单包含了应用程序的部署信息。应用程序清单的位置之类的信息包含在部署清单中。这些清单的完整模式在.NET SDK 文档中。

ClickOnce 有一些限制。例如,程序集不能添加到 GAC 文件夹中。表 17-6 比较了 ClickOnce 和 Windows Installer。

表 17-6

特 征	ClickOnce	Windows Installer
应用程序的安装位置	ClickOnce 应用程序缓存	Program Files 文件夹
给多个用户安装	否	是
安装共享文件	否	是

(续表)

特 征	ClickOnce	Windows Installer
安装驱动程序	否	是
安装到 GAC 文件夹中	否	是
在“启动”组中添加应用程序	否	是
在 Favorites 菜单中添加应用程序	否	是
注册表文件类型	否	是
访问注册表	否。有访问 HKLM 的 Full Trust 权限	是
文件的二进制修补	是	否
根据需要安装程序集	是	否

在一些情况下，使用 Windows Installer 比较好，但 ClickOnce 也适用于许多应用程序。

17.4.2 发布 ClickOnce 应用程序

ClickOnce 需要知道的全部信息都包含在两个清单文件中。为 ClickOnce 部署发布应用程序的过程就是生成清单，并把文件放在正确的位置。清单文件可以在 Visual Studio 2010 中生成。还可以使用一个命令行工具 `mage.exe`，它还有一个带 GUI 的版本 `mageUI.exe`。

在 Visual Studio 2010 中创建清单文件有两种方式。在 Project Properties 对话框的 Publish 选项卡底部有两个按钮，一个是 Publish Wizard 按钮，另一个是 Publish Now 按钮。Publish Wizard 按钮要求回答几个关于应用程序的部署问题，然后生成清单文件，把所有需要的文件复制到部署位置。Publish Now 按钮使用在 Publish 选项卡中设置的值，创建清单文件，并把文件复制到部署位置。

为了使用命令行工具 `mage.exe`，必须传递各个 ClickOnce 属性的值。使用 `mage.exe` 可以创建和更新清单文件。在命令提示符中输入 `mage.exe-help`，就会显示传递所需值的语法。

`mage.exe` 的 GUI 版本(`mageUI.exe`)类似于 Visual Studio 2010 中的 Publish 选项卡。使用 GUI 工具可以创建和更新同时是应用程序清单和部署清单的文件。

ClickOnce 应用程序会显示在控制面板中“添加/删除程序”对话框对应的小程序中，这与其他安装的应用程序一样。一个主要区别是用户可以选择卸载应用程序或回滚到以前的版本。ClickOnce 在 ClickOnce 应用程序缓存中保存以前的版本。

17.4.3 ClickOnce 设置

两个清单文件都有几个属性。最重要的属性是应用程序应从什么地方部署。必须指定应用程序的依赖项。Publish 选项卡中有一个 Application Files 按钮，单击它会打开一个对话框，用来输入应用程序需要的所有程序集。单击 Prerequisite 按钮会显示一个与应用程序一起安装的常见必备程序列表。可以选择从发布应用程序的位置上安装必备程序，也可以从供应商的 Web 站点上安装必备程序。

单击 Update 按钮会显示一个对话框，其中包含了如何更新应用程序的信息。当有应用程序的新版本时，可以使用 ClickOnce 更新应用程序。其选项包括：每次启动应用程序时检查是否有更新版本，或在后台检查更新版本。如果选择后台选项，就可以输入两次检查的指定间隔时间。此时可以使用允许用户拒绝或接收更新版本的选项。它可用于在后台进行强制更新，这样用户就不知道进行了更新。下次运行应用程序时，会使用新版本替代旧版本。还可以使用另一个位置存储更新文件。

这样, 原始安装软件包在一个位置, 用于给新用户安装应用程序, 而所有的更新版本放在另一个位置上。

安装应用程序时, 可以让它以在线模式或离线模式下运行。在离线模式下, 应用程序可以从“开始”菜单中运行, 就好像它是用 Windows Installer 安装的。在线模式表示应用程序只能在有安装文件夹的情况下运行。

17.4.4 ClickOnce 文件的应用程序缓存

用 ClickOnce 发布的应用程序不能安装在 Program Files 文件夹中, 它们会放在应用程序缓存中, 应用程序缓存驻留在当前用户的 Documents and Settings 文件夹的 Local Settings 子文件夹下。控制部署的这个方面意味着, 可以把应用程序的多个版本同时放在客户机上。如果应用程序设置为在线运行, 就会保留用户访问过的每个版本。对于设置为本地运行的应用程序, 会保留当前版本和以前的版本。

所以, 把 ClickOnce 应用程序回滚到以前的版本是一个非常简单的过程。如果用户进入控制面板中“添加/删除程序”对话框对应的小程序, 则所显示的对话框将允许删除 ClickOnce 应用程序或回滚到以前的版本。管理员可以修改清单文件, 使之指向以前的版本。之后, 下次用户运行应用程序时, 会检查是否更新版本。应用程序不查找要部署的新程序集, 而是还原以前的版本, 但不需要用户的交互。

17.4.5 应用程序的安全性

通过 Internet 或内联网部署的应用程序, 其安全性或可信赖设置比安装到本地驱动器上的应用程序低。例如, 默认情况下, 如果应用程序从 Internet 上启动或部署, 它就位于 Internet Security 区域。也就是说, 除了其他内容之外, 它不能访问文件系统。如果应用程序是从共享文件中安装的, 它就在 Intranet 区域中运行。

如果 ClickOnce 应用程序需要的信赖度高于默认值, 它就会提示用户获得运行应用程序所需的权限。这些权限在应用程序清单的 `trustInfo` 元素中设置。只需要授予这个设置中要求的权限。因此, 如果应用程序需要文件访问权限, 就不需要授予 Full Trust 权限, 只要文件访问权限即可。

另一个选项是使用 Trusted Application Deployment。Trusted Application Deployment 是基于整个企业授予权限的方式, 不需要提示用户。给每台客户机标识一个信任许可颁发者, 这可以通过公钥加密法完成。一般一个公司只有一个信任许可颁发者。一定要把颁发者的私钥放在安全的地方。

信任许可从颁发者申请。所申请的信任级别是信任许可配置的一部分。还必须给许可颁发者提供用于应用程序签名的公钥。所创建的许可包含用于应用程序签名的公钥和许可颁发者的公钥。这个信任许可会嵌入到部署清单中。最后一步是用自己的密钥对部署清单签名。现在应用程序就可以部署了。

在客户打开部署清单时, Trust Manager 会确定 ClickOnce 应用程序是否有较高的信任级别。首先查看颁发者的许可。如果它是有效的, 就比较许可中的公钥和用于应用程序签名的公钥。如果它们匹配, 就给应用程序授予需要的权限。

17.5 Visual Studio 2010 高级选项

前面讨论的安装过程功能非常强大，可以完成许多工作。在安装过程中还可以控制许多方面。例如，可以使用 Visual Studio 2010 中的各种编辑器建立条件安装，或者添加注册表键和自定义对话框。SampleClientSetupSolution 示例就启用了所有这些高级选项。

17.5.1. 文件系统编辑器

文件系统编辑器允许指定组成应用程序的各种文件和程序集部署到目标计算机的什么地方。默认情况下，会显示一组标准的部署文件夹。使用该编辑器可以添加任意多个自定义文件夹和特定文件夹。还可以给应用程序添加桌面快捷方式和“开始”菜单快捷方式。组成部署的任何文件都必须在文件系统编辑器中引用。

17.5.2. 注册表编辑器

注册表编辑器允许给注册表添加键和数据。在第一次显示该编辑器时，会显示一组标准的主键：

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

HKEY_CURRENT_USER 和 HKEY_LOCAL_MACHINE 在 Software/[manufacturer] 键中包含了额外的条目，其中 Manufacturer 是在部署项目的 Manufacturer 属性中输入的信息。

要添加额外的键和值，应在编辑器的左边突出显示一个主键，从主菜单中选择 Action 命令，再选择 New。选择要添加的键或值类型。重复这个步骤，直到完成了需要注册表的所有设置为止。如果在左边的窗格中选择了 Registry on Target Machine 选项，再选择 Action 菜单，就会看到一个 Import 选项，该选项允许导入已定义的 *.reg 文件。

要为键创建默认值，必须先为该键输入对应值。然后在右边窗格或值窗格上选择值名称。从 File 菜单中选择 Rename 命令，并删除该名称。按回车键，值名称就替换为(Default)。

还可以在该编辑器中为子键和值设置一些属性。前面还没有讨论的唯一属性是 DeleteAt- Uninstall。设计良好的应用程序应在卸载过程中删除由该应用程序添加的所有键。默认设置是不删除键。

注意，维护应用程序设置的首选方法是使用基于 XML 的配置文件。与注册表项相比，这些文件提供了更多灵活性，更容易还原和备份。

17.5.3. 文件类型编辑器

文件类型编辑器用于建立文件和应用程序之间的关系。例如，在双击具有.doc 扩展名的文件时，就会在 Word 中打开该文件。使用这个编辑器可以为应用程序创建这种关系。

为了添加关系，执行下面的步骤：

- (1) 从 Action 菜单中选择 File Types on Target Machine 命令。
- (2) 然后选择 Add File Type 命令。在 Properties 窗口中，可以设置关系的名称。
- (3) 在 Extension 属性中添加应与应用程序相关的文件扩展名。不要输入句点，可以用分号隔开多个扩展名，例如，ex1;ex2。

(4) 在 Command 属性中选择带有省略号的按钮。

(5) 接着选择要与特定的文件类型相关的文件(一般是可执行文件)。注意任何一个扩展名都应只与一个应用程序相关。

默认情况下,编辑器会显示&Open as the Document Action。我们还可以添加其他选项。编辑器中显示的动作顺序就是在用户右击文件类型时在弹出的上下文菜单中显示的动作顺序。注意第一项总是默认动作。可以为动作设置 Argument 属性,这是用于启动应用程序的命令行参数。

17.5.4. 用户界面编辑器

有时在安装过程中要求用户提供更多的信息。用户界面编辑器可用于为一组预定义的对话框指定属性。该编辑器分为两个部分 Install 和 Admin。一个用于标准安装,另一个用于管理员的安装。每个部分又分为 3 个子部分: Start、Progress 和 End。这些子部分表示安装过程的 3 个基本阶段,如图 17-3 所示。

表 17-7 列出了可以添加到项目中的对话框类型。

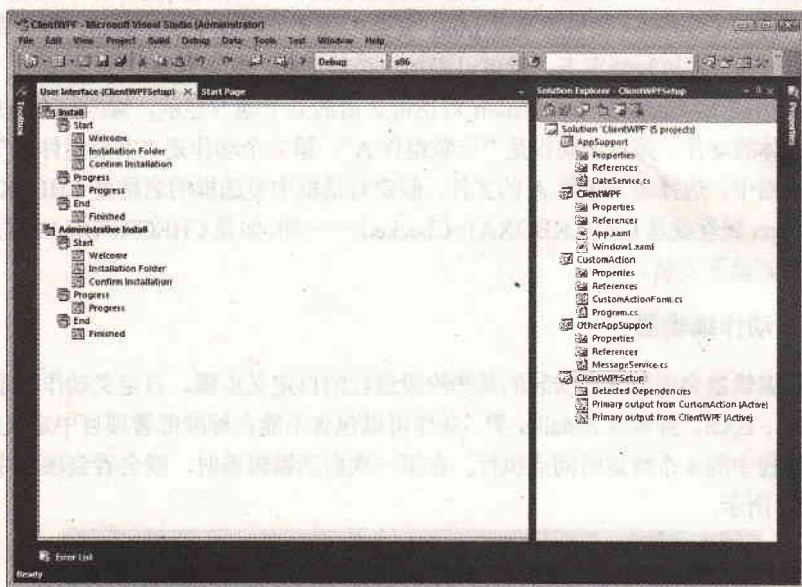


图 17-3

表 17-7

对 话 框	说 明
Checkboxes	至多包含 4 个复选框。每个复选框都包含 Lable、Value 和 Visible 属性
Confirm Installation	允许用户在安装开始之前确认各个设置
Customer Information	包含集团名称、公司名称和序列号的编辑字段。公司名称和序列号是可选的
Finished	在安装过程的最后显示
Installation Address	用于 Web 应用程序,显示一个对话框,让用户选择另一个安装 URL
Installation Folder	用于客户端应用程序,显示一个对话框,让用户选择另一个安装文件夹
License Agreement	显示许可协议,该协议位于 LicenseFile 属性指定的文件中

(续表)

对 话 框	说 明
Progress	在安装过程中显示一个进度指示器，说明当前的安装状态
Radio Buttons	至多包含 4 个单选按钮，每个单选按钮都包含 Lable 和 Value 属性
Read Me	显示 readme 信息，该信息包含在 ReadMe 属性指定的文件中
Register User	执行一个在注册过程中指导用户操作的应用程序，该应用程序必须在安装项目中提供
Splash	显示一个位图图像
TextBoxes	至多包含 4 个文本框，每个文本框都包含 Lable、Value 和 Visible 属性
Welcome	包含两个属性：WelcomeText 和 CopyrightWarning，它们都是字符串属性

这些对话框还包含一个设置横幅位图的属性，大多数对话框还包含一个设置横幅文本的属性。还可以在编辑器窗口中向上或向下拖动对话框，改变它们的显示顺序。

既然可以获得一些信息，现在的问题就是如何使用它们。此时就要使用项目中大多数对象都包含的 Condition 属性。Condition 属性必须是 true，安装步骤才能继续下去。例如，假定安装程序包含 3 个可选的安装组件。在这种情况下，就可以添加一个对话框，其中包含 3 个复选框。该对话框应在 Welcome 对话框之后 Confirm Installation 对话框之前的某个地方显示。修改每个复选框的 Label 属性，来描述具体的动作。第一个动作是“安装组件 A”，第二个动作是“安装组件 B”，依次类推。在文件系统编辑器中，选择表示组件 A 的文件。假定对话框中复选框的名称是 CHECKBOXA1，那么文件的 Condition 属性就是 CHECKBOXA1=Checked——即，如果 CHECKBOXA1 复选框被选中，就安装文件；否则就不安装。

17.5.5. 自定义动作编辑器

自定义动作编辑器允许定义在安装的某些阶段进行的自定义步骤。自定义动作应事先创建好，它可以包含 DLL、EXE、脚本或 Installer 类。动作可以包含不能在标准部署项目中定义的特定步骤。动作应在部署过程中的 4 个特定时间点执行。在第一次启动编辑器时，就会看到项目中的这 4 个时间点，如图 17-4 所示。

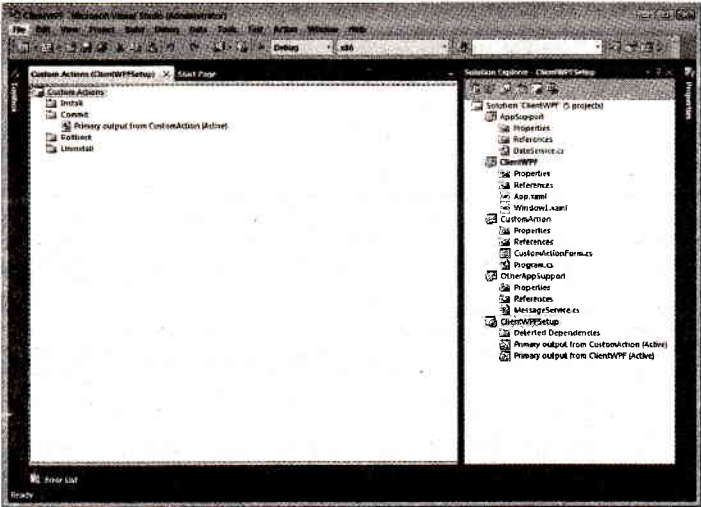


图 17-4

- **Install**——动作在安装阶段的最后执行
- **Commit**——动作在安装完成后执行，不记录错误
- **Rollback**——动作在回滚阶段完成后执行
- **Uninstall**——动作在卸载完成后执行

要添加动作，首先选择在其中要执行动作的安装阶段。再从 **Action** 菜单中选择 **Add Custom Action** 菜单选项，打开 **File System** 对话框，这表示包含动作的组件必须是部署项目的一部分。因为动作在要部署的目标计算机上执行，所以它应列在文件系统编辑器中。

在添加完动作后，可以选择表 17-8 中列出的一个或多个属性。

表 17-8

参 数	命令行参数
Condition	Windows Installer 条件，若要执行动作，该条件必须为 true
CustomDataAction	可用于动作的自定义数据
EntryPoint	包含动作的自定义 DLL 的入口点。如果动作包含在可执行文件中，这个属性就不起作用
InstallerClass	布尔值，如果设置为 true，就指定该动作是一个 .NET 类 <code>ProjectInstaller</code>
Name	动作的名称，默认为动作的文件名
SourcePath	动作在开发人员的计算机上的路径

因为动作是在部署项目外部开发的代码，所以可以给应用程序自由添加专业的外观。但要注意这些动作都在相关的阶段完成后发生。如果选择 **Install** 阶段，动作就会在安装阶段完成后执行。如果要在该过程之前执行动作，就应创建一个启动条件。

17.5.6. Launch Conditions 编辑器

Launch Conditions 编辑器允许指定在安装继续之前必须满足的一些条件。启动条件可以分为不同类型的条件。基本启动条件是 **File Search**、**Registry Search** 和 **Windows Installer Search**。在编辑器第一次启动时，会看到两组：**Search Target Machine** 和 **Launch Conditions**，如图 17-5 所示。一般需要进行搜索，并根据该搜索的成功或失败来执行某个条件。这通过设置搜索的 **Property** 属性来实现。可以在安装过程中访问 **Property** 属性，也可以在其他动作的 **Condition** 属性中也可以检查该属性。还可以在编辑器中添加某个启动条件。在这个条件中，把 **Condition** 属性设置为搜索的 **Property** 属性值。在该条件中，可以指定一个 URL，用于下载所搜索的文件、注册表键或安装组件。注意在图 17-5 中，默认添加了一个 .NET Framework 条件。

File Search 搜索某个文件或某类文件。可以设置与文件相关的许多不同属性，来确定如何搜索文件，这些属性包括文件名、文件夹位置、各种日期值，版本信息和大小。还可以设置要搜索的子文件夹数目。

Registry Search 允许搜索键和值，还允许设置待搜索的根键。

Windows Installer Search 搜索指定的安装组件。这个搜索由 GUID 执行。

Launch Conditions 编辑器提供了两个预打包的启动条件：一个是 .NET Framework 启动条件，它允许搜索运行库的特定版本；另一个启动条件是搜索 MDAC 的特定版本，该搜索使用注册表搜索，来查找相关的 MDAC 注册表项。

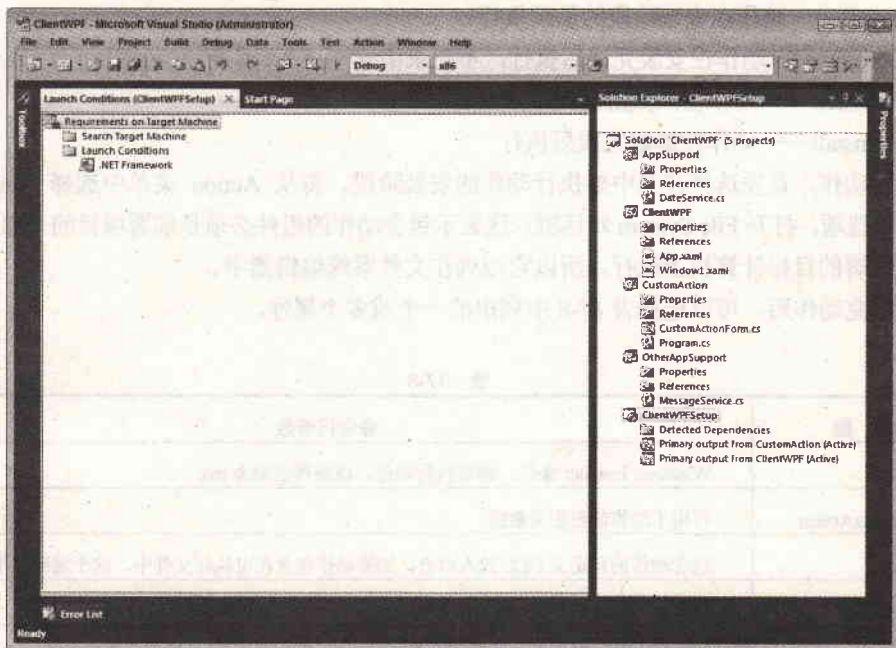


图 17-5

17.6 小结

部署软件对桌面软件的开发人员比较困难。随着 Web 站点越来越复杂，部署基于服务器的软件也变得更困难。本章探讨了 Visual Studio 2010 和 .NET Framework 3.5 版本的部署选项和功能；使部署更容易完成，错误也较少。

在阅读完本章后，您应能创建部署软件包，它可以解决几乎所有的部署问题。客户端应用程序可以在本地部署，或通过 Internet 或内联网来部署。本章还介绍了部署项目的扩展特性和部署项目的配置方式。我们可以使用 No Touch Deployment 和 ClickOnce 部署应用程序。ClickOnce 的安全特性为部署客户端应用程序提供了一种安全、有效的方式。使用部署项目安装 Web 应用程序还可以更轻松地完成 IIS 的配置。如果预编译应用程序，则发布 Web 站点还有额外的好处。

第Ⅲ部分

基 础

- 第 18 章 程序集
- 第 19 章 检测
- 第 20 章 线程、任务和同步
- 第 21 章 安全性
- 第 22 章 本地化
- 第 23 章 System.Transactions
- 第 24 章 网络
- 第 25 章 Windows 服务
- 第 26 章 互操作性
- 第 27 章 核心 XAML
- 第 28 章 Managed Extensibility Framework
- 第 29 章 文件和注册表操作

第 18 章

程 序 集

本章内容:

- 程序集概述
- 创建程序集
- 使用应用程序域
- 共享程序集
- 版本问题

程序集是.NET 用于部署和配置单元的术语。本章主要讨论什么是程序集,如何使用它们,它们的功能为什么这么强大。

本章将讲述如何动态地创建程序集,如何把程序集加载到应用程序域中,如何在不同的应用程序之间共享程序集。本章还会介绍版本问题,这是共享程序集的一个重要方面。

18.1 程序集的含义

程序集是.NET 应用程序的部署单元。.NET 应用程序包含一个或多个程序集。通常扩展名是 EXE 或 DLL 的.NET 可执行程序称为程序集。程序集和本地 DLL 或 EXE 有什么区别? 它们的文件扩展名虽然相同,但.NET 程序集包含元数据,这些元数据描述了程序集中定义的所有类型及其成员的信息,即方法、属性、事件和字段。

.NET 程序集的元数据还提供了程序集中文件的相关信息、版本信息和所使用的程序集的信息。.NET 程序集是以前为本地 DLL 的 DLL hell 提供的解决方案。

程序集是自我描述的安装单元,由一个或多个文件组成。程序集可以是包括元数据的 DLL 或 EXE,它也可以由多个文件组成,例如,资源文件、模块和 EXE。

程序集可以是私有或共享的。在简单的.NET 应用程序中,最好使用私有程序集工作。私有程序集没有特殊的管理、注册和版本设置等问题,只有用户自己的应用程序在使用私有程序集时才有版本问题。其他应用程序不受影响,因为它们有自己的程序集副本。在这种应用程序中使用的私有组件应与应用程序一起安装。因为私有程序集位于应用程序所在的目录或子目录下,所以应用程序不会有版本冲突问题。其他应用程序都不会重写私有的程序集。当然,仍可以使用私有程序集的版本号。这非常有助于代码的修改,但它不是.NET 所必需的。

在使用共享程序集时,几个应用程序都使用同一个程序集,且与它有一定的依赖关系。共享程序集减少了磁盘和内存空间的需求。使用共享程序集时,要遵循许多规则。共享程序集必须有一个版本号和一个唯一的名称,通常它安装在全局程序集缓存(global assembly cache, GAC)中。GAC 允许共享系统上同一个程序集的不同版本。

18.1.1 程序集的功能

程序集的功能可以总结如下:

- 程序集是自描述的。不再需要考虑注册表键、从其他地方获得类型库等问题。程序集包含描述程序集的元数据。元数据包括从程序集中导出的类型和一个清单。下一节将介绍清单。
- 版本的相互依赖性在程序集的清单中进行了记录。任何被引用的程序集的版本存储在程序集的清单中,这样就很容易确定因错误的版本号而引起的部署失败了。以后使用的引用程序集版本可以由开发人员和系统管理员配置。本章后面将介绍可用的版本策略及其工作方式。
- 程序集可以并行加载。从 Windows 2000 开始,就可以获得并行功能,其中同一个 DLL 的不同版本可以在系统上同时使用。您检查过目录<windows>\winsxs 吗? .NET 允许同一个程序集的不同版本在一个进程中使用!那么这有什么用呢?如果程序集 A 引用共享程序集 Shared 的版本 1,程序集 B 引用共享程序集 Shared 的版本 2,而用户同时使用程序集 A 和程序集 B,则应用程序需要使用共享程序集 Shared 的这两个版本,在.NET 中,应加载和使用两个版本。.NET 4 运行库允许一个进程中有多个 CLR 版本(2 和 4)。例如,这样就可以加载有不同 CLR 要求的插件。在同一个进程的不同 CLR 版本中,对象之间没有直接通信的.NET 方式,但可以使用其他技术,如 COM。
- 应用程序使用应用程序域来确保其独立性。使用应用程序域,许多应用程序就可以独立地运行在一个进程中。一个应用程序中的错误不会直接影响同一个进程中的其他应用程序。
- 安装非常简单,只需复制一个程序集中的所有文件,一条 xcopy 命令就足够了。这个特性称为 ClickOnce 部署。但是在一些情况下不能进行 ClickOnce 部署,而需要正常的 Windows 安装。第 17 章讨论过应用程序的部署。

18.1.2 程序集的结构

程序集由描述它的程序集元数据、描述导出类型和方法的类型元数据、MSIL 代码和资源组成。所有这些部分都在一个文件中,或者分布在几个文件中。

在第一个例子中,程序集元数据、类型元数据、MSIL 代码和资源都在一个文件 Component.dll 中,如图 18-1 所示,这个程序集由一个文件组成。

第二个例子介绍的是分布在 3 个文件中的一个程序集,如图 18-2 所示。Component.dll 包含程序集元数据、类型元数据和 MSIL 代码,但不包含资源。这个程序集使用了一张图片 picture.jpeg,该图片没有嵌在 Component.dll 中,而是在程序集元数据中引用。程序集元数据还引用了一个模块 util.netmodule,该模块自身只包含一个类的类型元数据和 MSIL 代码,不包含程序集元数据,所以这个模块自身没有版本信息,也不能单独安装。第二个例子中的这 3 个文件构成了一个程序集,这个程序集是一个安装单元,还可以在另一个文件中放置程序集清单。

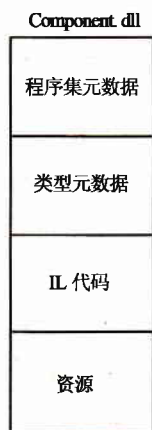


图 18-1

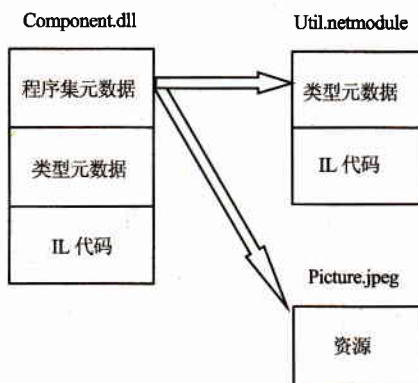


图 18-2

18.1.3 程序集清单

程序集的一个重要部分是程序集清单，它是元数据的一部分，描述了程序集和引用它所需的所有信息，并列出了它所有的依赖关系。清单由以下部分组成：

- 标识(名称、版本、区域性和公钥)。
- 属于该程序集的一个文件列表。一个程序集至少要有一个文件，也可以包含许多个文件。
- 被引用程序集的列表。在程序集清单中说明了在程序集中使用的所有程序集，这些引用信息包括版本号和公钥。公钥用于唯一地标识程序集。本章后面将讨论公钥。
- 一组许可请求——运行这个程序集需要的许可。许可详见第 21 章。
- 导出的类型，假定它们在一个模块中定义，该模块从程序集中引用，程序集就包含它们；否则它们就不是程序集清单的一部分。模块是可重用的单元。类型描述在程序集中存储为元数据，使用属性和方法可以从这些元数据中获得结构和类，它替代了以前用 COM 描述类型的类型库。使用 COM 客户端很容易在程序集清单的外部生成一个类型库。反射机制使用已导出类型的信息，便于后面绑定到类。有关反射的内容，请参见第 14 章。

18.1.4 名称空间、程序集和组件

也许您目前会混淆名称空间、类型、程序集和组件。名称空间如何与程序集的概念相匹配？名称空间完全独立于程序集。在一个程序集中可以有不同的名称空间，同一个名称空间也可以分布在多个程序集上。名称空间只是类型名的一种扩展，它属于类型名的范畴。

例如，除了许多其他名称空间外，程序集 `mscorlib` 和 `System` 都包含名称空间 `System.Threading`。尽管程序集包含相同的名称空间，但没有相同的类名。

18.1.5 私有程序集和共享程序集

程序集可以是共享的，也可以是私有的。私有程序集或者位于应用程序所在的同一个目录下，或者位于其子目录中。使用私有程序集时，不需要考虑与其他类的命名冲突或版本问题。在构建过程中引用的程序集会复制到应用程序的目录下。私有程序集是构建程序集的一般方式，特别是在同一个公司中构建应用程序和组件的时候，就更是如此。



尽管私有程序集可能仍有命名冲突(应用程序可能包含多个程序集,这些程序集本来有冲突,或者一个私有程序集中的名称与应用程序使用的一个共享程序集中的名称冲突),但命名冲突会大大减少。如果使用了多个私有程序集或使用了其他应用程序中的共享程序集,最好利用恰当命名的名称空间和类型,使命名冲突降低到最少。

在使用共享程序集时,必须遵循一些规则。程序集必须是唯一的,因此,必须有一个唯一的名称(称为强名)。强名的一部分是一个强制的版本号。当组件由另一个开发商构建,而不是应用程序的开发商构建时,以及一个大型应用程序分布在几个子项目中时,常常需要使用共享程序集。另外,一些技术(如.NET Enterprise Services)需要在特定的情形下使用共享程序集。

18.1.6 附属程序集

附属程序集是只包含资源的程序集,它尤其适用于本地化。因为程序集有一种相关的文化,所以资源管理器会查找包含特定文化资源的附属程序集。



附属程序集的更多信息可参见第 22 章。

18.1.7 查看程序集

程序集可以使用命令行实用工具 ildasm 来查看,这是一个 MSIL 反汇编程序。从命令行中启动 ildasm,把程序集作为其参数,或者选择 File | Open 命令,就可以打开程序集。

图 18-3 是 ildasm 打开的一个示例程序 SharedDemo.dll,我们后面会构建这个程序。ildasm 显示了程序集清单,以及 Wrox.ProCSharp.Assemblies.Sharing 名称空间中的 SharedDemo 类型。打开该程序集清单时,就可以看到版本号、程序集特性和被引用的程序集及其版本。打开类的方法,就可以查看 MSIL 代码。



除了 ildasm 之外,.NET Reflector 是另一个用于分析程序集的强大工具。.NET Reflector 可以搜索类型和成员,调用的图和被调的图,并将 IL 代码反编译为 C#、C++ 或 Visual Basic。这个工具可以从 <http://www.redgate.com/products/reflector> 上下载。

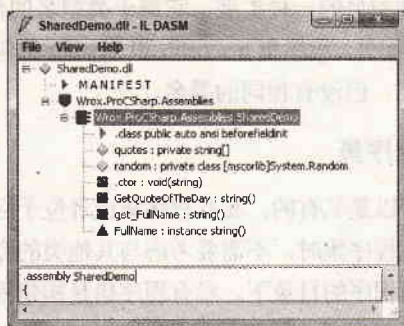


图 18-3

18.2 创建程序集

前面学习了程序集的含义,下面要构建一些程序集。当然,本书前面已经构建了一些程序集,因为.NET 可执行程序也是一个程序集。下面要介绍程序集的特定选项。

18.2.1 创建模块和程序集

在 Visual Studio 中,所有的 C# 项目类型都会创建一个程序集。无论是选择 DLL 项目类型,还是 EXE 项目类型,都会创建一个程序集。使用命令行 C# 编译器 `csc`,也可以创建模块。模块是一个没有程序集特性的 DLL(所以它不是程序集,但可以在以后把它添加到程序集中)。命令

```
csc /target:module hello.cs
```

创建模块 `hello.netmodule`,可以使用 `ildasm` 查看这个模块。

模块也有一个清单,但在该清单中没有 `assembly` 条目(除了引用的外部程序集之外),因为模块没有程序集特性。不能用模块来配置版本或许可,而只能在程序集范围内进行。在模块的清单中可以找到程序集的引用。使用 `csc` 的 `/addmodule` 选项,可以把模块添加到已有的程序集中。

为了比较模块和程序集,下面创建一个简单的 A 类,并用下面的命令编译它:

```
csc /target:module A.cs
```

编译器生成 `A.netmodule` 文件,它不包括程序集的信息(使用 `ildasm` 可以查看清单信息)。模块的清单显示了被引用的程序集 `mscorlib` 和 `module` 条目,如图 18-4 所示。

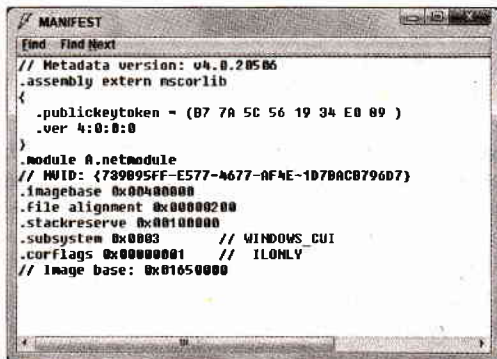


图 18-4

下面创建一个程序集 B,它包括模块 `A.netmodule`。不需要用一个源文件来生成这个程序集,生成该程序集的命令如下:

```
csc /target:library /addmodule:A.netmodule /out:B.dll
```

在使用 `ildasm` 查看程序集时,只能找到一个清单。在清单中,引用了程序集 `mscorlib`。接着看看带有散列算法和版本的程序集部分。算法的数量决定了用于创建程序集的散列代码的算法类型。在通过编程创建程序集时,可以选择该算法。清单包含属于该程序集的所有模块的一个列表。在图 18-5 中,可以看出 `file A.netmodule` 属于该程序集的一部分,从模块中导出的类是程序集清单的一部分,从程序集本身导出的类则不是,如图 18-5 所示。

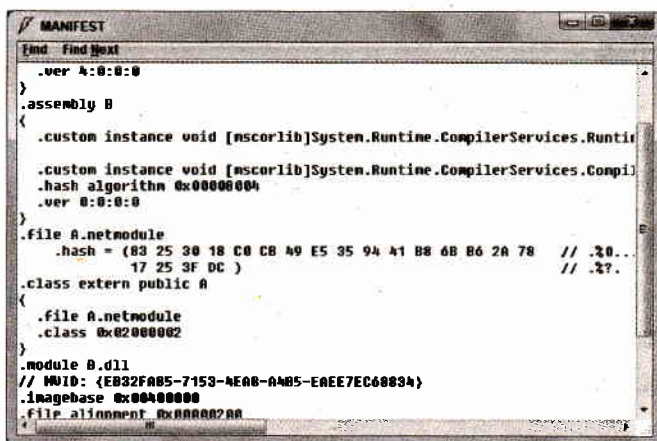


图 18-5

模块的作用是什么？模块可以更快地启动程序集，因为并不是所有的类型都在一个文件中。模块只在需要时加载。使用模块的另一个原因是，要用多种编程语言来创建一个程序集。一个模块用 Visual Basic 编写，另一个模块用 C# 编写，并且这两个模块都包括在一个程序集中。

18.2.2 程序集的特性

在创建 Visual Studio 项目时，会自动生成源文件 AssemblyInfo.cs，这个文件在 Solution Explorer 窗口的 Properties 对话框中。在该文件中，可以使用一般的源代码编辑器配置程序集的特性。下面是从项目模板中生成的一个文件：

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
//
[assembly: AssemblyTitle("DomainTest")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("DomainTest")]
[assembly: AssemblyCopyright("Copyright © 2010")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed
// to COM
[assembly: Guid("ae0acc2c-0daf-4bb0-84a3-f9f6ac48bfe9")]
//
```

```
// Version information for an assembly consists of the following four
// values:
//
//     Major Version
//     Minor Version
//     Build Number
//     Revision
//
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

这个文件用于配置程序集清单。编译器读取程序集特性，把特定的信息插入到程序集清单中。
assembly:前缀把属性标记为程序集级别特性。与其他特性相反，程序集级别特性与特定的语言元素无关，用于程序集特性的参数是名称空间 System.Reflection、System.Runtime.CompilerServices 和 System.Runtime.InteropServices 中的类。

 第 14 章介绍了特性和如何创建和使用自定义特性的内容。

表 18-1 是 System.Reflection 名称空间中定义的程序集特性列表。

表 18-1	
程序集的特性	说 明
AssemblyCompany	指定公司名
AssemblyConfiguration	指定构建信息，例如零售或调试信息
AssemblyCopyright 和 AssemblyTrademark	包含版权和商标信息
AssemblyDefaultAlias	如果程序集名不容易理解(例如，动态地创建程序集名时的 GUID)，就可以使用该特性。使用这个特性可以指定别名
AssemblyDescription	描述程序集或产品。如果查看可执行文件的属性，这个值就会显示为 Comments
AssemblyProduct	指定了程序集所属产品的名称
AssemblyTitle	给程序集提供一个友好的名称。该名称可以包含空格。使用文件属性时，这个值就显示为 Description
AssemblyCulture	定义程序集的文化。这个特性对附属程序集很重要
AssemblyInformationalVersion	在引用程序集时，这个特性不用于版本检查，它仅用于版本信息。该特性非常适合于指定使用多个程序集的应用程序的版本。打开可执行程序属性，这个值就显示为 Product Version
AssemblyVersion	这个特性给出了程序集的版本号。本章后面讨论版本问题
AssemblyFileVersion	这个特性定义了文件的版本。这个值显示在 Windows 文件属性窗口中，但它对 .NET 的行为没有影响

下面是配置这些特性的一个示例：

```
[assembly: AssemblyTitle("Professional C#")]
[assembly: AssemblyDescription("Sample Application")]
[assembly: AssemblyConfiguration("Retail version")]
[assembly: AssemblyCompany("Wrox Press")]
[assembly: AssemblyProduct("Wrox Professional Series")]
[assembly: AssemblyCopyright("Copyright (C) Wrox Press 2010")]
[assembly: AssemblyTrademark("Wrox is a registered trademark of " +
    "John Wiley & Sons, Inc.")]
[assembly: AssemblyCulture("")]

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

在 Visual Studio 2008 中, 可以用项目属性、应用程序设置和程序集信息配置这些特性, 如图 18-6 所示。

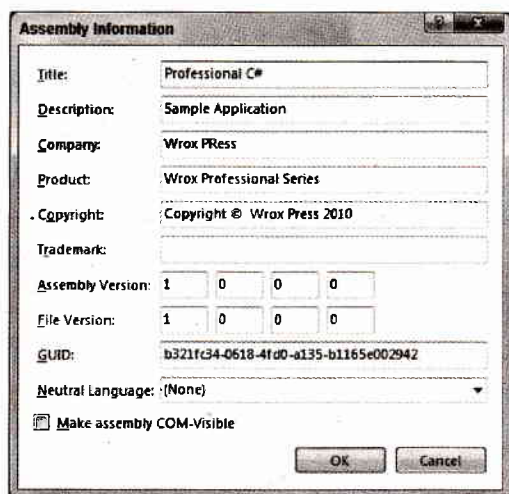


图 18-6

18.2.3 动态加载和创建程序集

在开发期间, 添加对程序集的引用, 使之包含在程序集引用中, 该类型的程序集就可用于编译器。在运行期间, 只要实例化了一种类型的程序集, 或者使用了该类型的一个方法, 就会加载所引用的程序集。除了使用这种自动操作之外, 还可以通过编程加载程序集。为此, 可以使用 `Assembly` 类的静态方法 `Load()`。这个方法是重载的, 其中可以使用 `AssemblyName` 给它传递程序集的名称或字节数组。

还可以快速创建程序集, 如下面的例子所示。这个例子演示了把 C# 代码输入文本框后, 启动 C# 编译器, 就会动态地创建一个新程序集, 并调用编译的代码。

要动态地编译 C# 代码, 可以使用 `Microsoft.CSharp` 名称空间中的 `CSharpCodeProvider` 类。使用这个类可以编译代码, 从 DOM 树、文件和源代码中生成程序集。

该应用程序的 UI 是使用 WPF 创建的, 如图 18-7 所示。窗口由一个要输入 C# 代码的文本框、一个按钮和一个 `TextBlock` WPF 控件组成, `TextBlock` 横跨最后一行的所有列, 以显示对应的结果。

```

        results = provider.CompileAssemblyFromSource(
            options, sb.ToString());
    }

    if (results.Errors.HasErrors)
    {
        hasError = true;
        var errorMessage = new StringBuilder();
        foreach (CompilerError error in results.Errors)
        {
            errorMessage.AppendFormat("{0} {1}", error.Line,
                error.ErrorText);
        }
        returnData = errorMessage.ToString();
    }
    else
    {
        TextWriter temp = Console.Out;
        var writer = new StringWriter();
        Console.SetOut(writer);
        Type driverType = results.CompiledAssembly.GetType("Driver");

        driverType.InvokeMember("Run", BindingFlags.InvokeMethod |
            BindingFlags.Static | BindingFlags.Public,
            null, null, null);
        Console.SetOut(temp);

        returnData = writer.ToString();
    }

    return returnData;
}
}
}

```

代码段 DynamicAssembly/CodeDriver.cs

`CompileAndRun()`方法需要一个字符串参数 `input`，在其中可以传递一行或多行 C# 代码。因为调用的每个方法都必须包含在方法和类中，所以变量 `prefix` 和 `postfix` 定义了动态地创建的 `Driver` 类的结构和包含参数中代码的 `Run()` 方法。使用 `StringBuilder`，把 `prefix`、`postfix` 和 `input` 变量中的代码合并起来，创建一个完整的、可编译的类。再使用这个得到的字符串，通过 `CSharpCodeProvider` 类编译代码。`CompileAssemblyFromSource()` 方法动态地创建一个程序集。因为这个程序集仅需要在内存中使用，所以设置了编译器参数选项 `GenerateInMemory`。

如果所传递的源代码包含错误，它们就会显示在 `CompilerResults` 的 `Errors` 集合中。错误和返回数据一起返回，变量 `hasError` 设置为 `true`。

如果源代码编译成功，就调用新的 `Driver` 类的 `Run()` 方法。这个方法的调用通过反射来实现。新编译的程序集可以使用 `CompilerResults.CompiledType` 来访问，在这个程序集中，新的 `Driver` 类由变量 `driverType` 引用。接着使用 `Type` 类的 `InvokeMember()` 方法调用 `Run()` 方法。因为这个方法定义为公共静态方法，所以必须相应地设置 `BindingFlags`。要查看程序写到控制台上的结果，需要把控制台重定向到 `StringWriter` 中，最终使用 `returnData` 变量返回程序的全部结果。



用 `InvokeMember()` 方法运行代码需要使用 .NET 反射功能, 该功能详见第 14 章。

WPF 按钮的 `Click` 事件连接到 `Compile_Click()` 方法上, 在该方法中, 实例化 `CodeDriver` 类, 并调用 `CompileAndRun()` 方法。从文本框 `textCode` 中提取输入, 把结果写到 `TextBlock` 控件 `textOutput` 中。



可从
wrox.com
下载源代码

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
    var driver = new CodeDriver ();
    bool isError;
    textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
    if (isError)
    {
        textOutput.Background = Brushes.Red;
    }
}
```

代码段 `DynamicAssembly/DynamicAssemblyWindow.xaml.cs`

现在可以启动应用程序, 在 `TextBox` 中输入 C# 代码, 如图 18-8 所示, 编译并运行代码。

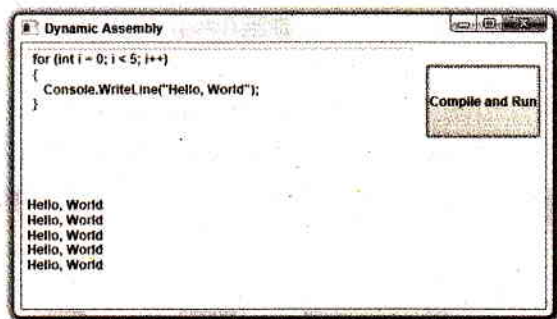


图 18-8

前面编写的程序有一个缺点: 每次单击 `Compile And Run` 按钮时, 都会创建并加载一个新程序集, 于是程序需要越来越多的内存。不能从应用程序中卸载程序集。要卸载程序集, 需要使用应用程序域。

18.3 应用程序域

在 .NET 之前的技术中, 进程作为独立的边界来使用, 每个进程都有其私有的虚拟内存; 运行在一个进程中的应用程序不能写入另一个应用程序的内存, 也不会因为这种方式破坏其他应用程序。该进程用作应用程序之间的一个独立而安全的边界。在 .NET 体系结构中, 应用程序有一个新的边界: 应用程序域。使用托管 IL 代码, 运行库可以确保在同一个进程中不能访问另一个应用程序的内存。多个应用程序可以运行在一个进程的多个应用程序域中, 如图 18-9 所示。

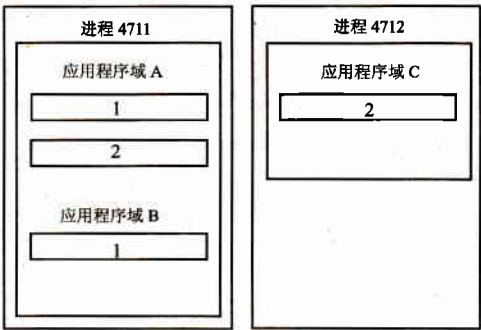


图 18-9

把程序集加载到应用程序域中。在图 18-9 中，进程 4711 有两个应用程序域。在应用程序域 A 中，实例化对象 1 和 2，对象 1 在程序集 1 中，对象 2 在程序集 2 中。在进程 4711 中，第二个应用程序域有一个实例 1。要最小化占用的内存，在应用程序域中，程序集的代码应只加载一次。实例和静态成员不能在应用程序域之间共享，也不能直接访问另一个应用程序域中的对象。此时需要一个代理(proxy)。所以在图 18-9 中，如果没有代理，应用程序域 B 中的对象 1 就不能直接访问应用程序域 A 中的对象 1 或 2。

AppDomain 类用于创建和终止应用程序域，加载和卸载程序集和类型，以及枚举应用程序域中的程序集和线程。下面用一个小例子来说明应用程序域。

首先，创建一个 C#控制台应用程序 AssemblyA，在 Main()方法中添加一个 Console.WriteLine()方法，这样，我们就知道这个方法什么时候调用。另外，添加 Demo 类，其构造函数的参数是两个 int 值，用这两个参数和 AppDomain 类创建实例。从将要创建的第二个应用程序中加载程序集 AssemblyA.exe:



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Assemblies
{
    public class Demo
    {
        public Demo(int val1, int val2)
        {
            Console.WriteLine("Constructor with the values {0}, {1}" +
                " in domain {2} called", val1, val2,
                AppDomain.CurrentDomain.FriendlyName);
        }
    }

    class Program
    {
        static void Main()
        {
            Console.WriteLine("Main in domain {0} called",
                AppDomain.CurrentDomain.FriendlyName);
        }
    }
}
```

代码段 AssemblyA/Program.cs

运行应用程序，结果如下所示：

```
Main in domain AssemblyA.exe called.
```

创建的第二个项目也是一个 C# 控制台应用程序：DomainTest。首先，使用 AppDomain 类的 FriendlyName 属性显示当前域的名称。调用 CreateDomain() 方法，新建应用程序域 NewAppDomain，然后把程序集 AssemblyA 加载到新域中，调用 ExecuteAssembly() 方法来调用 Main() 方法：



可从
wrox.com
下载源代码

```
using System;
using System.Reflection;

namespace Wrox.ProCSharp.Assemblies
{
    class Program
    {
        static void Main()
        {
            AppDomain currentDomain = AppDomain.CurrentDomain;
            Console.WriteLine(currentDomain.FriendlyName);
            AppDomain secondDomain = AppDomain.CreateDomain("New AppDomain");
            secondDomain.ExecuteAssembly("AssemblyA.exe");
        }
    }
}
```

代码段 DomainTest/Program.cs

在启动程序 DomainTest.exe 前，通过 DomainTest 项目引用 AssemblyA.exe 程序集。通过 Visual Studio 2010 引用程序集，就是把程序集复制到项目的目录中，这样项目才能找到这个程序集。如果找不到这个程序集，就会抛出 System.IO.FileNotFoundException 异常。

运行 DomainTest.exe 程序后，会得到如下所示的控制台输出。DomainTest.exe 是第一个应用程序域的友好名称。第二行是 New AppDomain 中新加载的程序集的输出结果。在进程查看器中看不到进程 AssemblyA.exe 的执行，因为没有新建进程，AssemblyA 加载到 DomainTest.exe 进程中。

```
DomainTest.exe
Main in domain New AppDomain called
```

在新加载的程序集中，还可以新建一个实例，以替代调用 Main() 方法。在下面的例子中，用 CreateInstance() 方法替代 ExecuteAssembly() 方法，它的第一个参数是程序集名 AssemblyA，第二个参数定义了应实例化的类型 Wrox.ProCSharp.Assemblies.AppDomains.Demo，第三个参数 true 表示不区分大小写。System.Reflection.BindingFlags.CreateInstance 是一个绑定的标志枚举值，用来指定应调用的构造函数：



可从
wrox.com
下载源代码

```
AppDomain secondDomain = AppDomain.CreateDomain("New AppDomain");
// secondDomain.ExecuteAssembly("AssemblyA.exe");
secondDomain.CreateInstance("AssemblyA", "Wrox.ProCSharp.Assemblies.Demo",
    true, BindingFlags.CreateInstance, null, new object[] { 7, 3},
    null, null);
```

代码段 DomainTest/Program.cs

在应用程序成功运行后，会得到如下所示的控制台输出。

```
DomainTest.exe
Constructor with the values 7, 3 in domain New AppDomain called
```

前面介绍了如何创建和调用应用程序域。在运行库宿主上，会自动创建应用程序域。ASP.NET 为运行在 Web 服务器上的每个 Web 应用程序创建一个应用程序域。Internet Explorer 创建运行托管控件的应用程序域。对于应用程序，如果要卸载一个程序集，创建应用程序域就非常有效。卸载程序集只能通过终止应用程序域来进行。



如果程序集是动态加载的，且需要在使用完后卸载程序集，应用程序域就非常有用了。在主应用程序域中，不能删除已加载的程序集，但可以终止应用程序域，在该应用程序域中加载的所有程序集都会从内存中清除。

了解了应用程序域后，就可以修改前面创建的 WPF 程序了。新类 `CodeDriverInAppDomain` 使用 `AppDomain.CreateDomain` 新建了一个应用程序域。在这个新应用程序域中，使用 `CreateInstanceAndUnwrap()` 方法实例化 `CodeDriver` 类。使用 `CodeDriver` 实例，调用 `CompileAndRun()` 方法，之后再次卸载新应用程序域。



可从
wrox.com
下载源代码

```
using System;
using System.Runtime.Remoting;

namespace Wrox.ProCSharp.Assemblies
{
    public class CodeDriverInAppDomain
    {
        public string CompileAndRun(string code, out bool hasError)
        {
            AppDomain codeDomain = AppDomain.CreateDomain("CodeDriver");

            CodeDriver codeDriver = (CodeDriver)
            codeDomain.CreateInstanceAndUnwrap("DynamicAssembly",
                "Wrox.ProCSharp.Assemblies.CodeDriver");

            string result = codeDriver.CompileAndRun(code, out hasError);

            AppDomain.Unload(codeDomain);
            return result;
        }
    }
}
```

代码段 `DynamicAssembly/CodeDriverInAppDomain.cs`



`CodeDriver` 类本身现在同时主应用程序域和新应用程序域中使用，因此不能删除这个类使用的代码。如果要删除这些代码，就可以定义一个由 `CodeDriver` 类实现的接口，再在主应用程序域中使用这个接口。但是在这个例子中，这并不是个问题，因为只需删除用 `Driver` 类动态创建的代码即可。

要在另一个应用程序域中访问 `CodeDriver` 类, `CodeDriver` 类就必须派生于基类 `MarshalByRefObject`。只有派生自这个基类的类才能通过另一个应用程序域来访问。在主应用程序域中,实例化一个代理,以通过应用程序域之间的信道调用这个类的方法。



可从
wrox.com
下载源代码

```
using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
    public class CodeDriver: MarshalByRefObject
    {
```

代码段 DynamicAssembly/CodeDriver.cs

`Compile_Click()`事件处理程序现在可以修改为使用 `CodeDriverInAppDomain` 类,而不是使用 `CodeDriver` 类:



可从
wrox.com
下载源代码

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
    var driver = new CodeDriverInAppDomain();
    bool isError;
    textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
    if (isError)
    {
        textOutput.Background = Brushes.Red;
    }
}
```

代码段 DynamicAssembly/DynamicAssemblyWindow.xaml.cs

现在可以单击应用程序的 `Compile And Run` 按钮任意多次,并且总是会卸载生成的程序集。



使用 `AppDomain` 类的 `GetAssemblies()`方法,就可以查看应用程序域中加载的程序集。

18.4 共享程序集

程序集可以由一个应用程序单独使用,在默认情况下不共享程序集。在使用共享程序集时,需要考虑一些特定的要求。

本节将探讨共享程序集所需的信息。必须使用强名唯一地标识共享程序集,强名通过给程序集的签名来创建。本节还将解释延迟签名的过程。共享程序集一般安装在全局程序集缓存(GAC)中,我们还会讨论如何使用 GAC。

18.4.1 强名

共享程序集名要求是它必须是全局唯一的，并且必须可以保护该名称。任何其他人不能使用同一个名称创建程序集。

COM 使用全局唯一标识符(GUID)解决了第一个问题。但第二个问题仍没有解决，因为每个人都可以盗用这个 GUID，用相同的标识符创建不同的对象。通过.NET 程序集的强名可以解决这两个问题。

强名由下述项组成：

- 程序集本身的名称
- 版本号。有了版本号，可以同时使用同一个程序集的不同版本。不同的版本可以同时存在，并可以同时加载到同一个进程上。
- 公钥保证强名是独一无二的。它也保证被引用的程序集不能从另一个源中替代。
- 区域性。区域性详见第 22 章。



共享程序集必须有一个强名，来唯一地标识该程序集。

强名是一个简单的文本名称，附带版本号、公钥和文化。不能使用每个程序集新建公钥，但可以在公司中有一个这样的公钥，这样该密钥就唯一地标识了公司的程序集。

但是，这个密钥不能用作信任密钥。程序集可以利用 Authenticode 签名来建立信任关系。Authenticode 签名的密钥可以与针对强名使用的密钥不同。



从开发的目的来看，可以使用不同的公钥，以后也可以与真正的密钥互换。该特性将在 18.4.8 节中介绍。

为了唯一地标识公司中的程序集，应使用名称空间层次结构来给类命名。下面是如何组织名称空间的一个简单例子：Wrox 出版社使用主名称空间 Wrox 来标识其类和名称空间。在名称空间 Wrox 下面的层次结构中，必须对名称空间进行组织，使所有的类都是唯一的。本书中的每一章都使用 Wrox.ProCSharp.<Chapter>形式的不同名称空间。本章使用 Wrox.ProCSharp.Assemblies 名称空间。这样，如果在某两章中都有 Hello 类，就不会有名称冲突，因为它们在不同的名称空间中。可以在不同的书中使用的实用工具类则放在 Wrox.Utilities 名称空间中。

公司名一般用作名称空间的第一部分，但它不一定是唯一的，因此必须使用某种机制来建立强名。此时可以使用公钥。因为在强名中使用公钥/私钥规则，所以不能访问私钥的人，就不能破坏性地创建一个程序集，让客户无意中调用该程序集。

18.4.2 使用强名获得完整性

在创建共享组件时，必须使用公钥/私钥对。编译器把公钥写入程序集清单，创建属于该程序集的所有文件的散列表，用私钥对这个散列表签名，该私钥不存储在程序集中。这样就可以确保没有人可以修改该程序集。签名可以使用公钥来验证。

在开发过程中，客户端程序集必须引用共享程序集。编译器把被引用程序集的公钥写入客户端程序集的清单中。要减少存储量，就不应把公钥写入客户端程序集的清单，而应写入公钥标记。公

钥标记是公钥散列表中的最后 8 位字节, 且是唯一的。

在运行期间加载共享程序集时(如果客户程序集是使用本机映像生成器安装的, 则应在安装期间加载), 共享程序集的散列表可以使用存储在客户端程序集中的公钥来验证。除了私钥的主人外, 其他人都不能修改共享的组件程序集。供应商 A 创建了一个组件 Math, 在客户端上引用该组件, 黑客的组件就无法替代它。只有私钥的主人才能用新版本替换原来的共享组件。只要共享程序集来自期望的发布者, 就保证了完整性。

图 18-10 显示了一个共享组件, 它的公钥由客户端程序集引用, 该程序集在其清单中包含共享程序集的公钥标记。

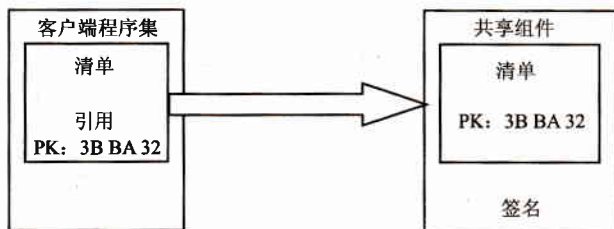


图 18-10

18.4.3 全局程序集缓存

顾名思义, 全局程序集缓存(Global Assembly Cache)就是可全局使用的程序集的缓存。大多数共享程序集都安装在这个缓存中; 另外, 也可以使用共享目录(也在服务器上)。

GAC 位于 <windows>\assembly 目录下。在这个目录中, 有多个 GACxxx 目录 和一个 NativeImages_<runtime version> 目录。GACxxx 目录包含共享程序集, GAC_MSIL 目录包含带纯 .NET 代码的程序集; GAC_32 包含专用于 32 位平台的程序集。在 64 位系统上, GAC-64 目录包含专用于 64 位平台的程序集。目录 GAC 用于向后兼容 .NET 1.0 和 1.1。在 NativeImages_<runtime version> 目录下, 有编译为本机代码的程序集。如果再深入该目录结构, 就会看到与程序集名类似的一些目录, 其下是一个版本目录和程序集本身。这样就可以安装同一个程序集的不同版本。

gacutil.exe 实用工具可以使用命令行安装、卸载和显示程序集。gacutil 的一些选项如下所示:

- gacutil /l 显示程序集缓存中的所有程序集。
- gacutil /i mydll 把共享程序集 mydll 安装到程序集缓存上。即使程序集已安装, 也可以使用选项/f 强制安装到 GAC 中。如果修改了程序集, 但没有改变版本号, 使用选项/f 就很有用。
- gacutil /u mydll 卸载程序集 mydll。



在产品系统中, 应使用安装程序, 把共享程序集安装在 GAC 中。部署参见第 17 章。

18.4.4 创建共享程序集

在本例中, 创建一个共享程序集, 再创建一个使用该共享程序集的客户。

创建共享程序集与创建私有程序集的区别不大。首先创建一个简单的 Visual C# 类库项目 SharedDemo。把名称空间改为 Wrox.ProCSharp.Assemblies, 把类名改为 SharedDemo。输入下面的代码。类的构造函数把文件的所有行都将读取到一个数组中。文件名作为参数传递给构造函数。

GetQuoteOfTheDay()方法只返回这个数组的一个随机字符串。



可从
wrox.com
下载源代码

```
using System;
using System.IO;

namespace Wrox.ProCSharp.Assemblies
{
    public class SharedDemo
    {
        private string[] quotes;
        private Random random;

        public SharedDemo(string filename)
        {
            quotes = File.ReadAllLines(filename);
            random = new Random();
        }

        public string GetQuoteOfTheDay()
        {
            int index = random.Next(1, quotes.Length);
            return quotes[index];
        }
    }
}
```

代码段 SharedDemo/SharedDemo.cs

18.4.5 创建强名

要共享这个程序集，需要一个强名。要创建这个名称，可以使用强名工具(sn)：

```
sn -k mykey.snk
```

强名实用工具生成和编写一个公钥/私钥对，并把该密钥对写到文件中，此处的文件是 mykey.snk。

在 Visual Studio 2010 中，可以选择 Signing 选项卡，用项目属性标记程序集，如图 18-11 所示。还可以使用这个工具创建密钥。但不需要为每个项目创建密钥文件。整个公司可以只使用几个密钥。最好根据安全要求创建不同的密钥，详见第 21 章。

用 Visual Studio 设置 signing 选项，会给编译器设置添加/keyfile 选项。Visual Studio 还允许创建用密码保护的密钥文件。这种文件的扩展名是.pfx，如图 18-11 所示。

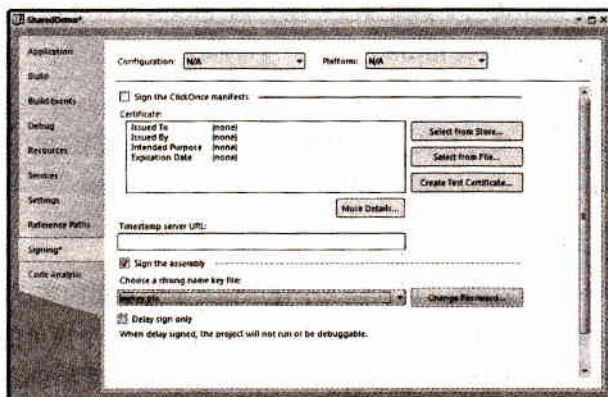


图 18-11

在重新生成该文件后,公钥就在该程序集清单中。可以使用 ildasm 验证这一点,如图 18-12 所示。

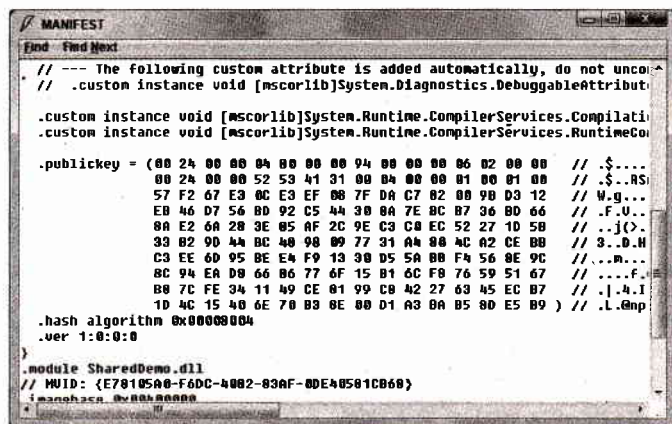


图 18-12

18.4.6 安装共享程序集

程序集中有了公钥后,就可以使用全局程序集缓存工具 gacutil 及其 /i 选项把它安装到全局程序集缓存中。即使程序集已安装,也可以使用选项 /f 强制把它写到 GAC 中。

```
gacutil /i SharedDemo.dll /f
```

然后,可以使用全局程序集缓存查看器或 gacutil /l SharedDemo 检查共享程序集的版本,看看它是否安装成功。

18.4.7 使用共享程序集

要使用共享程序集,应创建一个 C# 控制台应用程序 Client。把名称空间的名称改为 Wrox.ProCSharp.Assemblies。以引用私有程序集的方式引用共享程序集:使用菜单 Project | Add Reference 命令。



有了共享程序集,引用属性 Copy Local 就可以设置为 false,这样,共享程序集就不会复制到输出文件的目录中,而会从 GAC 中加载。

在项目条目中添加 Quotes.txt 文件,并把 Copy to Output Directory 属性设置为 Copy if newer。下面是 Client 应用程序的代码:



可从
wrox.com
下载源代码

```
using System;
namespace Wrox.ProCSharp.Assemblies
{
    class Program
    {
        static void Main()
        {
            var quotes = new SharedDemo("Quotes.txt");
            for (int i=0; i < 3; i++)
```

```

        Console.WriteLine(quotes.GetQuoteOfTheDay());
        Console.WriteLine();
    }
}

```

代码段 Client/Program.cs

在使用 ildasm 查看客户端程序集的清单时(如图 18-13 所示),可以看到对共享程序集 Shared Demo 的引用: .assembly extern SharedDemo。这些引用信息的一部分是版本号(详见后面的内容)和公钥的标记。

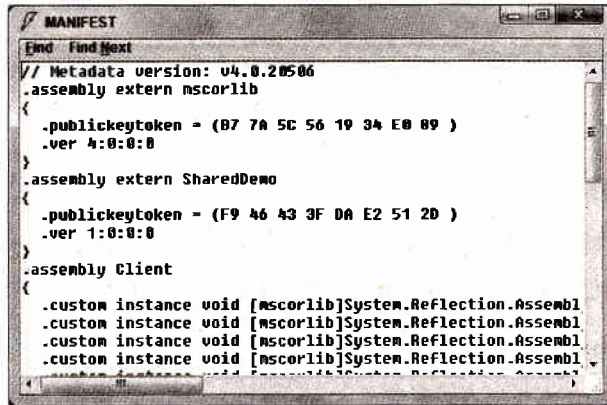


图 18-13

公钥的标记也可以使用强名实用工具 sn 在共享程序集中查看: sn -T 会显示程序集中的公钥标记, sn -Tp 显示标记和公钥。注意使用大写字母 T!

该程序引用了示例文件, 其结果如下所示。

"We don't like their sound. And guitar music is on the way out." — Decca Recording, Co., in rejecting the Beatles, 1962

"The ordinary 'horseless carriage' is at present a luxury for the wealthy; and although its price will probably fall in the future, it will never come into as common use as the bicycle." — The Literary Digest, 1889

"Landing and moving around the moon offers so many serious problems for human beings that it may take science another 200 years to lick them", Lord Kelvin (1824-1907)

18.4.8 程序集的延迟签名

公司的私钥应安全地存储。大多数公司都不允许所有的开发人员都能访问私钥。只有几个有安全权限的人才能访问它。这就是为什么程序集的签名可以以后(如在发布之前)添加的原因。在全局程序集特性 AssemblyDelaySign 设置为 true 时, 签名就不会存储在程序集中, 但保留了足够的空间, 以便以后添加它。不使用密钥, 就不能测试程序集, 在全局程序集缓存中安装它。但是, 可以使用临时密钥进行测试, 以后再用真正的公司密钥代替这个临时密钥。

延迟程序集的签名需要执行下面的步骤:

- (1) 首先必须用强名实用工具 **sn** 创建一个公钥/私钥对,生成的文件 **mykey.snk** 包含公钥和私钥。

```
sn -k mykey.snk
```

- (2) 接着提取公钥,使之可以用于开发人员。选项 **-p** 提取密钥文件的公钥。文件 **mypublickey.snk** 仅包含公钥。

```
sn -p mykey.snk mykeypub.snk
```

公司中的所有开发人员都可以使用这个密钥文件 **mykeypub.snk**,用 **/delaysign**+选项编译程序集。这样,签名就没有添加到程序集中,但可以以后添加它。在 Visual Studio 2010 中,延迟签名选项可以在 **Signing** 设置的复选框中设置。

- (3) 关闭签名的验证功能,因为程序集没有包含签名。

```
sn -Vr SharedDemo.dll
```

- (4) 在发布之前,程序集可以用 **sn** 实用工具重新签名。**-R** 选项用于对以前已签名或延迟签名的程序集进行重新签名。程序集的重新签名可以由部署应用程序的软件包且有权访问用于发布的私钥的人员完成。

```
sn -R MyAssembly.dll mykey.snk
```



签名的验证功能只能在开发过程中关闭。不经过验证是不能发布程序集的,因为这个程序集可能被恶意的程序集替代。



程序集的重新签名可以通过在 MSBuild 文件中定义任务来自动完成,相关内容参见第 16 章。

18.4.9 引用

GAC 中的程序集可以包含与它们相关联的引用。这些引用负责:如果应用程序仍需要引用程序集,缓存的程序集就不能删除。例如,如果 Microsoft 安装程序包(.msi 文件)安装了一个共享程序集,就只能通过卸载应用程序删除它,而不能从全局程序集缓存中删除它。从全局程序集缓存中删除程序集会得到一条错误消息:“程序集 <name>不能卸载,因为其他应用程序还需要它。”

使用 **gacutil** 实用工具和选项 **/r** 可以设置程序集的引用。**/r** 选项需要一个引用类型、一个引用 ID 和一个描述。引用的类型可以是下面 3 个选项中的一个: **UNINSTALL_KEY**、**FILEPATH** 或 **OPAQUE**。**UNINSTALL_KEY** 由 MSI 使用,定义一个注册表键之后卸载它需要使用该选项。用 **FILEPATH** 可以指定一个目录,应用程序的根目录是有用的目录。**OPAQUE** 引用类型允许设置任意类型的引用。

命令行:

```
gacutil /i shareddemo.dll /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

通过引用客户端应用程序的目录,在全局程序集缓存中安装程序集 **SharedDemo**。这个程序集的另一种安装可以使用另一条路径安装,或使用 **OPAQUE ID** 安装,如下面的命令行所示:

```
gacutil /i shareddemo.dll /r OPAQUE 4711 "Opaque installation"
```

现在全局程序集缓存中只有一个程序集，但它有两个引用。为了从全局程序集缓存中删除程序集，必须删除这两个引用：

```
gacutil /u shareddemo /r OPAQUE 4711 "Opaque installation"
gacutil /u shareddemo /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```



要删除共享程序集，选项/u需要不带文件扩展名 DLL 的程序集。而安装共享程序集的选项/i需要包含文件扩展名的完整文件名。



第 17 章介绍了程序集的部署，其中在 MSI 软件包中要处理引用总数。

18.4.10 本机映像生成器

使用本机映像生成器 `Ngen.exe`，可以在安装期间把 IL 代码编译为本机代码。这样程序启动就比较快，因为不再需要在运行时进行编译。比较预编译的程序集和在其中需要运行 JIT 编译器的程序集，其性能在编译 IL 代码后没有太大区别。使用本机映像生成器获得的唯一改进是应用程序的启动比较快，因为不需要运行 JIT。减少应用程序的启动时间是使用本机映像生成器的主要原因。如果从可执行文件中创建本机映像，也应从可执行文件加载的所有 DLL 中创建本机映像。否则，就仍需要运行 JIT 编译器。

`ngen` 实用工具在本机映像缓存中安装本机映像，本机映像缓存的物理目录是 `<windows>\assembly\NativeImages <RuntimeVersion>`。

使用 `ngen install myassembly` 可以把 MSIL 代码编译为本机代码，并把它安装到本机映像缓存中。如果要把程序集安装到本机映像缓存中，就应在安装程序中完成上述操作。

使用 `ngen` 和选项 `display` 还可以显示本机映像缓存中的所有程序集。如果把程序集名添加到选项 `display` 中，就可以得到这个程序集所有已安装版本的相关信息，以及依赖于本机程序集的程序集，如下所示。

```
C:\> ngen display System.AddIn
Microsoft (R) CLR Native Image Generator - Version 4.0.21006.1
Copyright (c) Microsoft Corporation. All rights reserved.

NGEN Roots:

System.AddIn, Version=3.5.0.0, Culture=Neutral, PublicKeyToken=b77a5c561934e089,
processorArchitecture=msil
System.AddIn, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

NGEN Roots that depend on "System.AddIn":

C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
AddInSideAdapters\Microsoft.VisualStudio.Tools.Applications.AddInAdapter.v9.0.dll
C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
AddInSideAdapters\Microsoft.VisualStudio.Tools.Office.AddInAdapter.v9.0.dll
...
```

如果系统的安全设置有变化,就不能保证本机映像达到运行应用程序的安全要求。这就是本机映像 in 系统配置变化时就无效的原因。使用命令 `ngen update`,会重新生成所有的本机映像,以包含新配置。

安装 .NET 4 时,会安装 Native Runtime Optimization Service。这个服务可以用于延迟本机映像的编译,重新生成已失效的本机映像。

安装程序可以使用 `ngen install myassembly /queue` 命令,通过本机映像服务延迟将 `myassembly` 编译为本机映像的过程。`ngen update /queue` 重新生成已失效的本机映像。使用 `ngen queue` 的选项 `pause`、`continue` 和 `status`,可以控制服务,并获取状态信息。



为什么不能在开发人员的系统中创建本机映像,而只能在产品系统中发布本机映像?因为本机映像生成器会处理与目标系统一起安装的 CPU,编译为 CPU 类型优化的代码。而在安装应用程序的过程中,CPU 是已知的。

18.5 配置 .NET 应用程序

COM 组件使用注册表来配置组件。.NET 应用程序的配置是使用配置文件完成的。使用注册表配置,就不可能使用 `xcopy` 部署了。配置文件甚至可以复制。配置文件使用 XML 语法来指定应用程序的启动和运行库配置。

本节将介绍:

- 可以使用 XML 基本配置文件进行哪些配置
- 用强名引用的程序集如何重新定向到另一个版本中
- 如何指定程序集的目录,以便在子目录中查找私有程序集,在公共目录或服务器上查找共享程序集

18.5.1 配置类别

可以把配置分为如下几类:

- **启动设置**——用于指定需要的运行库版本。同一个系统上可能安装了不同版本的运行库,使用 `<startup>` 元素来指定运行库版本。
- **运行库设置**——用于指定运行库如何进行垃圾收集,如何进行程序集绑定。也可以使用这些设置指定版本策略和代码库(code base)。本章的后面将详细介绍运行库设置。
- **WCF 设置**——用于利用 WCF 配置应用程序。这些配置参见第 43 章。
- **安全设置**——详见第 21 章。第 21 章将介绍加密配置和许可。

这些设置可以在 3 种配置文件中给出:

- **应用程序配置文件**——包含应用程序的特定设置,如程序集的绑定信息,远程对象的配置等。这个配置文件放在可执行文件所在的目录下,它与可执行文件同名,但最后添加了 `.config` 扩展名。ASP.NET 配置文件命名为 `web.config`。
- **计算机配置文件**——可以用于系统范围的配置。也可以在这里指定程序集绑定和远程配置。在绑定过程中,应在考虑应用程序配置文件之前考虑计算机配置文件。应用程序配置可以

重写计算机配置中的设置。应用程序配置文件应优先用于应用程序特定的设置，这样计算机配置文件会比较小，也容易管理。计算机配置文件位于`%runtime_install_path%\config\Machine.config`中。

- **发行者策略文件**——由组件的创建者用于指定共享程序集可以与旧版本兼容。如果新程序集版本仅修改了共享组件的一个错误，就不必把应用程序配置文件放在使用该组件的每个应用程序目录中，发行者可以添加一个发行者策略文件，从而把它标记为“可兼容的”。当组件不能用于所有的应用程序时，就可以在应用程序配置文件中重写发行者策略设置。与其他配置文件不同，发行者策略文件存储在全局程序集缓存中。

如何使用这些配置文件？客户如何根据程序集是共享还是私有的来查找程序集(也称为绑定)？私有程序集必须位于应用程序所在的目录或子目录下。进程 **probing** 可用于查找这样的程序集。如果程序集没有强名，**probing** 就不使用版本号。

共享程序集可以安装在全局程序集缓存中，或放在一个目录、共享网络或 Web 站点上。我们用 **codeBase** 的配置来指定这样一个目录(详见后面的内容)。在绑定共享程序集时，公钥、版本和区域性都很重要。所需程序集的引用记录在客户端程序集的清单中，包括名称、版本和公钥标记。所有的配置文件都要检查，以应用正确的版本策略。在全局程序集缓存和配置文件中指定的代码库也要检查，然后检查应用程序的目录，之后应用探测规则。

18.5.2 绑定程序集

前面已经介绍了如何把共享程序集安装到全局程序集缓存中。除了把共享程序集安装到全局程序集缓存中，还可以使用配置文件配置特定的共享目录。如果要在服务器上使用共享组件，就可以使用这个功能。如果要在应用程序之间共享一个程序集，但不希望它在全局程序集缓存中共享它，就可以把该程序集放在一个共享目录中。

查找程序集的正确目录有两种方式：使用 XML 配置文件中的 **codeBase** 元素，或者使用 **probing** 元素。**CodeBase** 元素配置只可用于共享程序集，而 **probing** 元素可用于私有和共享程序集。

1. <codeBase>

使用应用程序配置文件可以配置 **<codeBase>** 元素。下面的应用程序配置文件把对程序集 **SharedDemo** 的搜索重定向为从网络上加载它：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
          publicKeyToken="f946433fdade2512d" />
        <codeBase version="1.0.0.0"
          href="http://www.christiannagel.com/WroxUtils/SharedDemo.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

<codeBase> 元素有特性 **version** 和 **href**，使用 **version** 特性必须指定程序集的原始引用版本，使

用 href 特性可以定义应从中加载程序集的目录。在本例中, 使用 HTTP 协议所在的路径。使用 href="file:C:/WroxUtils/SharedDemo.dll" 可以指定本地系统上的一个目录或一个共享。

2. <probing>

如果没有配置 <codeBase> 元素, 程序集也没有存储在全局程序集缓存中, 运行库就会利用 probing 元素来查找程序集。 .NET 运行库会在应用程序目录或与所搜索程序集同名的子目录中查找文件扩展名为 .dll 或 .exe 的程序集。如果没有找到程序集, 会继续搜索。可以在应用程序配置文件的 <runtime> 部分中, 用 <probing> 元素配置搜索目录。使用 .NET Framework 配置工具选择应用程序的属性, 也可以很容易地完成这个 XML 配置。使用 .NET Framework 配置工具中的搜索路径可以配置该探测所在的目录。

得到的 XML 文件包含如下条目:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="bin;utils;" />
    </assemblyBinding>
  </runtime>
</configuration>
```

<probing> 元素只有一个必需的 privatePath 特性。这个应用程序配置文件告诉运行库, 应在应用程序的根目录下搜索程序集, 再在 bin 和 util 目录中搜索。这两个目录是应用程序根目录的子目录。不可能在应用程序根目录或其子目录的外部引用私有程序集。在应用程序根目录外部的程序集必须有一个共享名, 并可以使用 <codeBase> 元素来引用。

18.6 版本问题

对于私有程序集, 版本问题并不重要, 因为被引用的程序集会与客户端一起复制。客户端使用其私有目录下的程序集。

但是, 共享程序集就不是这样。下面先看看共享时发生的一些传统问题。使用共享组件, 则表示多个客户端应用程序可以使用同一个组件。在用新版本更新共享组件时, 新版本会中断已有客户端。我们不可能不发布新版本, 因为已有组件的新版本将需要并提供新功能。但我们可以仔细编程, 让程序保持向后兼容。但事情并不总是很顺利。

这个问题的解决方案是使用一种体系结构, 它允许安装共享组件的不同版本, 客户端使用它们在构建过程中引用的版本。这解决了许多问题, 但并没有解决全部问题。如果从客户端引用的组件中检测到了一个错误, 该怎么办? 我们可以更新这个组件, 并确保客户端使用新版本, 而不是在构建过程中引用的那个版本。

因此, 根据新版本中更改的类型, 有时要使用更新的版本, 有时则要使用旧一点的引用版本。所有 .NET 体系结构允许这两种情况。

在 .NET 中, 默认情况下使用原来引用的程序集。使用配置文件可以把引用重新定向到一个不同的版本。版本问题在绑定的体系结构中有非常关键的作用——客户端获得存储了其组件的正确程序集。

18.6.1 版本号

程序集的版本号由 4 部分组成, 例如 1.0.400.3300, 各部分分别是:

<Major>.<Minor>.<Build>.<Revision> .

这些号码根据应用程序配置来使用。



如果进行的改动与以前的版本不兼容, 但内部版本号或修订版本号的改动与之兼容最好改变主版本号或次版本号。这样就可以假定把程序集重定向到新版本中, 其中只有修改了的内部版本号和修订版本号是安全的。

在 Visual Studio 2010 中, 使用项目设置中的程序集信息可以指定程序集的版本号。项目设置将程序集特性[AssemblyVersion]写入 AssemblyInfo.cs 文件:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

除了全部定义版本号 4 部分之外, 还可以在第 3 或 4 个位置放置星号:

```
[assembly: AssemblyVersion("1.0.*")]
```

在这个设置中, 前两个数字指定主版本号和次版本号, 星号表示内部版本号和修订版本号是自动生成的。内部版本号是自从 2000 年 1 月 1 日以来的天数, 修订版本号表示自从当地时间的午夜开始的秒数除以 2。自动设置的版本号在开发期间很有帮助, 但在发布之前, 最好定义特定的版本号。

这个版本存储在程序集清单的 assembly 部分中。

在客户端应用程序中引用程序集, 会在客户端应用程序的程序集清单中存储引用的程序集版本。

18.6.2 通过编程方式获取版本

要查看在客户端应用程序中使用的程序集的版本, 可以在 SharedDemo 类中添加前面创建的只读属性 FullName, 以返回程序集的强名。要简化 Assembly 类的使用, 必须导入 System.Reflection 名称空间。



可从
wrox.com
下载源代码

```
public string FullName
{
    get
    {
        return Assembly.GetExecutingAssembly().FullName;
    }
}
```

代码段 SharedDemo/SharedDemo.cs

Assembly 类的 FullName 属性包含了类名、版本、位置和公钥标记, 在客户端应用程序中调用 FullName 时, 可以在其结果中看到这些内容。

在客户端应用程序中, 创建共享组件后, 在 Main() 方法中添加对 FullName 的一个调用:



可从
wrox.com
下载源代码

```
static void Main()
{
    var quotes = new SharedDemo("Quotes.txt");
```

```
Console.WriteLine(quotes.FullName);
```

代码段 Client/Program.cs

一定要使用 `gacutil` 在全局程序集缓存中再次注册共享程序集 `SharedDemo` 的新版本。如果没有找到引用的版本，就会抛出一个 `System.IO.FileLoadException` 异常，因为对没有绑定到正确的程序集。成功运行后，可以看到引用的程序集的完整名称，如下所示。

```
SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken= f946433fdae2512d
```

这个客户端程序可以用于测试这个共享组件的不同配置。

18.6.3 绑定到程序集版本

使用配置文件可以指定应绑定到共享程序集的另一个版本。假定创建共享程序集 `SharedDemo` 的一个新版本，其主版本号和次版本号是 1.1。我们不想重新建立客户端，只想在已有的客户端中使用程序集的新版本。这适用于下述场合：共享程序集有一个错误需要修改，或者因为新版本是兼容的，所以要删除旧版本。

运行 `gacutil.exe`，可以看到 `SharedDemo` 程序集安装了 1.0.0.0 和 1.0.3300.0 版本。

```
> gacutil -l SharedDemo

Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.21006.1
Copyright (c) Microsoft Corporation. All rights reserved.

The Global Assembly Cache contains the following assemblies:
    SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken=f946433fdae2512d,
    processorArchitecture=x86
    SharedDemo, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=f946433fdae251
    2d, processorArchitecture=x86

Number of items = 2
```

图 18-14 显示了客户端应用程序的清单，其中客户引用了程序集 `SharedDemo` 的 1.0.0.0 版本。

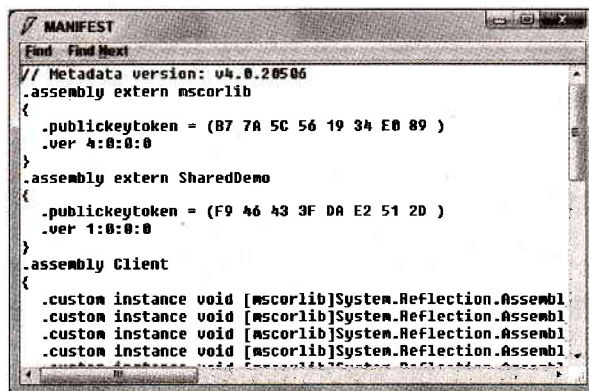


图 18-14

现在同样需要使用应用程序配置文件。与前面一样，重定向的程序集需要用 `<assemblyIdentity>` 元素指定。这个元素使用名称、区域性和公钥标记标识程序集。为了重定向到另一个版本上，要使用 `<bindingRedirect>` 元素。oldVersion 特性指定应把程序集的哪个版本重定向到新版本上。使用

oldVersion 特性可以指定一个范围,例如,应重定向 1.0.0.0~1.0.3300.0 之间所有程序集的版本。新版本用 newVersion 特性指定。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
          publicKeyToken="f946433fdae2512d" />
        <bindingRedirect oldVersion="1.0.0.0 - 1.0.3300.0"
          newVersion="1.0.3300.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

代码段 Client/App.config

18.6.4 发行者策略文件

使用全局程序集缓存器中的共享程序集,可以使用发行者策略避免版本冲突问题。假定有一个共享程序集由一些应用程序使用。如果在共享程序集中有一个关键的错误,会出现什么情况?可以看出,不需要重新建立所有使用该共享程序集的应用程序,因为可以使用配置文件重定向到这个共享程序集的新版本上。我们也许不了解所有使用该共享程序集的应用程序,但要为所有这些应用程序修改错误。此时可以创建发行者策略文件,把所有这些应用程序重新定向到该共享程序集的新版本上。



发行者策略文件只能应用于安装在全局程序集缓存中的共享程序集。

要建立发行者策略,必须:

- 创建发行者策略文件
- 创建发行者策略程序集
- 把发行者策略程序集添加到全局程序集缓存器中

1. 创建发行者策略文件

发行者策略文件是一个把已有版本或某个版本范围重定向到新版本的 XML 文件。这里使用的语法与应用程序配置文件相同,所以可以使用前面创建的文件,把旧版本 1.0.0.0~1.0.3300.00 重定向到新版本 1.0.3300.00 上。

把前面创建的文件重命名为 mypolicy.config,把它用作发行者策略文件。

2. 创建发行者策略程序集

要把发行者策略文件与共享程序集关联起来,必须创建一个发行者策略程序集,并把它放到全局程序集缓存器中。可以创建这种文件的工具是程序集链接器 al。/linkresource 选项把发行者策略文

件添加到生成的程序集中。生成的程序集的名称必须以 `policy` 开头，其后是应重定向的程序集的主次版本号，以及共享程序集的文件名。在本例中，发行者策略程序集必须命名为 `policy.1.0.SharedDemo.dll`，才能重定向主版本号为 1、次版本号为 0 的 `SharedDemo` 程序集。必须用 `/keyfile` 选项把一个密钥添加到这个发行者密钥中，新添加的这个密钥与用于签名共享程序集 `SharedDemo` 的密钥相同，这样才能保证该版本从同一个发行者重定向。

```
al /linkresource:mypolicy.config /out:policy.1.0.SharedDemo.dll
/keyfile:..\mykey.snk
```

3. 将发行者的策略程序集添加到全局程序集缓存中

现在，可以使用实用工具 `gacutil` 把发行者策略程序集添加到全局程序集缓存器中：

```
gacutil -i policy.1.0.SharedDemo.dll
```

如果已发布了同一个策略文件，就可以使用选项 `-f`。现在，可以删除位于客户端应用程序目录中的应用程序配置文件，并启动该客户端应用程序。尽管客户端程序集引用的是 1.0.0.0 版本，但因有了发行者策略，所以我们使用共享程序集的新版本 1.0.3300.0。

4. 重写发行者策略

有了发行者策略，共享程序集的发行者就可以保证程序集的新版本与旧版本兼容。从传统 DLL 的变化来看，这种保证并不总可靠。也许只有一个应用程序在使用新的共享程序集。为了修改使用新版本的应用程序中的错误，可以使用应用程序配置文件，重写发行者策略。

添加 XML `<publisherPolicy>` 元素和 `apply="no"` 特性，就可以禁用发行者策略。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
          publicKeyToken="f946433fdae2512d" />
        <publisherPolicy apply="no" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

代码段 Client/App.config

禁用发行者策略，可以在应用程序配置文件中配置不同版本的重定向。

18.6.5 运行库的版本

不仅可以安装和使用程序集的多个版本，还可以安装和使用 .NET 运行库 (CLR) 的多个版本。CLR 的 1.0、1.1、2.0 和 4.0 (及未来) 版本可以同时安装在一个操作系统上。Visual Studio 2010 面向通过 .NET 2.0、3.0、3.5 在 CLR 2.0 上运行的应用程序，以及通过 .NET 4 和 CLR 2.0 上运行的应用程序。

如果应用程序是用 CLR 2.0 构建的，就可以依然只安装了 CLR 4.0 版本的系统上运行它。但是，如果应用程序是用 CLR 4.0 构建的，就不能在只安装了 CLR 2.0 版本的系统上运行它。

在应用程序配置文件中，不仅可以重定向被引用的程序集的版本，还可以定义运行库所需的版本。在应用程序配置文件中可以指定应用程序需要的版本。`<supportedVersion>`元素标记应用程序支持的运行库版本。`<supportedVersion>`的顺序定义系统上可用的多个运行库版本的优先级。这里的配置首选.NET 4 运行库，也支持 2.0。为此，构建的应用程序必须面向.NET Framework 2.0、3.0 或 3.5。

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0.21006" />
    <supportedRuntime version="v2.0.50727" />
  </startup>
</configuration>
```

18.7 小结

程序集是为.NET 平台新安装的单元。Microsoft 从以前的体系结构中吸取教训，进行了全新的设计，以避免出现旧问题。本章讨论了程序集的特性：程序集是自描述的，不需要类型库和注册信息。版本的依赖性会准确地记录下来，这样，使用程序集时，旧 DLL 的 DLL Hell 问题就不复存在了。由于具备这些特性，因此开发、部署和管理就容易多了。

本章讨论了私有和共享程序集之间的差异，并介绍了如何创建共享程序集。有了私有程序集，就不必关心唯一性和版本冲突问题了，因为这些程序集仅由一个应用程序复制和使用。共享程序集需要使用一个密钥来保持其唯一性、确定其版本。我们介绍了全局程序集缓存，它可以用作共享程序集的智能存储器。

使用本机映像生成器可以更快地启动应用程序。有了本机映像生成器，就不需要运行 JIT 编译器，因为本机代码是在安装期间创建的。

本章还阐述了避免版本冲突问题，以使用与在开发过程中使用的版本不同的程序集。这通过发行者策略和应用程序配置文件实现。最后还讨论了 `probing` 元素如何使用私有程序集。

我们还讨论了如何动态地加载程序集，在运行期间创建程序集。如果要了解更多的信息，就可以参阅第 28 章介绍的.NET 4 中的插件模型。要使用插件模型和应用程序域，可以参见第 50 章介绍的 `System.Addin` 名称空间。

第 19 章

检 测

本章内容:

- 代码协定
- 跟踪
- 事件日志
- 性能监控

本章的内容有助于获得关于正在运行的应用程序的一些实时信息,找出应用程序在生产过程中某些问题的原因,或者监控需要的资源,以尽早适应较高的用户负载。这就是名称空间 `System.Diagnostics` 的作用。

当然,在应用程序中标记错误的一种方式抛出异常。然而,尽管应用程序不抛出异常,但仍不像期望的那样运行。应用程序可能在大多数系统上都运行良好,只在几个系统上出问题。在实时系统上,改变配置值可以改变日志行为,获得在应用程序运行的对象的详细实时信息。这可以用跟踪功能来实现。

如果应用程序出了问题,就需要通知系统管理员。使用事件查看器,系统管理员可以交互地监控应用程序的问题,通过添加订阅功能来了解发生的特定事件。事件日志机制允许写入应用程序的相关信息。

为了分析应用程序需要的资源,在指定的时间间隔内监控应用程序,规划另一个应用程序的分布或系统资源的扩展,系统管理员可使用性能监控器。使用性能计数器可以写入来自应用程序的实时数据。

现在,在 .NET 4 中, `System.Diagnostics.Contracts` 名称空间中的类提供了按协定设计的方式。使用这些类可以定义前提条件、后置条件和常量,它们不仅可以在运行期间检查,还可以使用静态的合同分析器检查。

本章介绍这些功能,并说明如何在应用程序中使用它们:

19.1 代码协定

按协定设计是 Eiffel 编程语言的一个理念。现在, .NET 4 在 `System.Diagnostics.Contracts` 名称空间中包含的类可用于静态检查代码和在运行期间检查代码,这些类可由所有的 .NET 语言使用。

利用这个功能可以定义方法中的前提条件、后置条件和常量。前提条件列出了参数必须满足的

要求，后置条件定义了返回数据必须满足的要求，常量定义了方法中变量必须满足的要求。

协定信息可以编译到调试代码和发布代码中。还可以定义一个单独的合同程序集，也可以进行许多静态检查，而无需运行应用程序。也可以在接口上定义协定，使接口的实现代码满足协定的要求。协定工具可以重写程序集，在运行时检查的代码中插入协定检查，在编译期间检查协定，给生成的 XML 文档中添加协定信息。

图 19-1 显示了 Visual Studio 2010 中代码协定的项目属性。这里可以定义应进行什么级别的运行时检查，指定在协定失败时是否打开断言对话框，配置静态检查。把 Perform Runtime Contract Checking 设置为 Full 来定义 CONTRACT_FULL 符号。因为许多协定方法都用 [Contract] (“CONTRACT_FULL”)特性注解，所以所有的运行时检查都只使用这个设置。

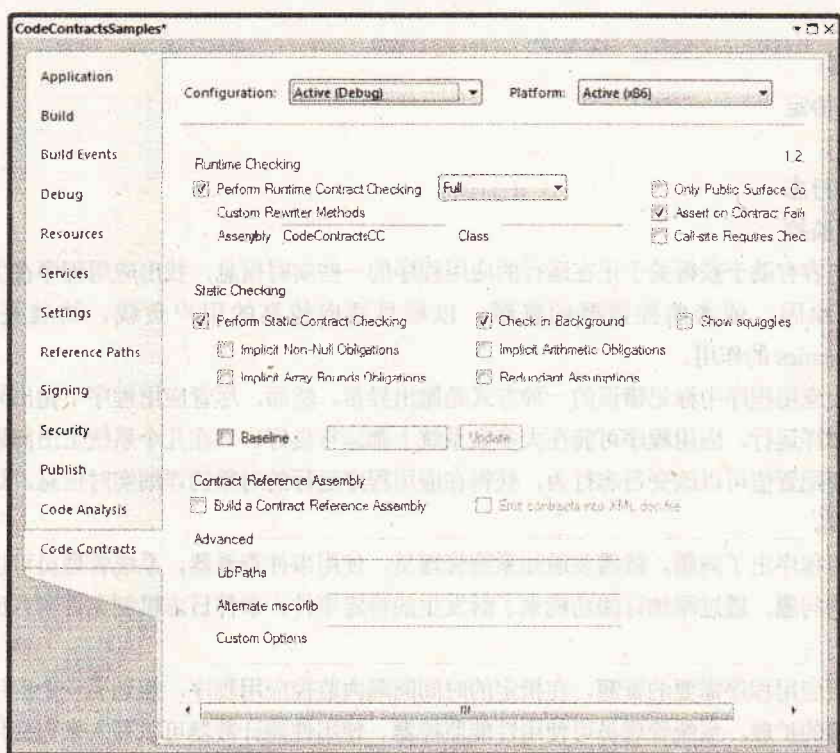


图 19-1

要使用代码协定，可以使用 .NET 4 在 System.Diagnostics.Contracts 名称空间中可用的类。但 Visual Studio 2010 没有包含这方面的工具。需要从 Microsoft Devlabs (<http://msdn.microsoft.com/en-us/devlabs/dd491992>)上下载 Visual Studio 的一个扩展。要使用这个工具进行静态分析，需要 Visual Studio Team System；对于运行时分析，有 Visual Studio 标准版就足够了。

代码协定用 Contract 类定义。在方法中定义的所有协定要求，无论是前提条件还是后置条件，都必须放在方法的开头。也可以给 ContractFailed 事件赋予一个全局的事件处理程序，运行期间失败的每个协定都会调用这个事件处理程序。调用 SetHandled()方法时包含 ContractFailedEventArgs 参数

e, 会禁止失败时的标准行为: 抛出异常。



可从
wrox.com
下载源代码

```
Contract.ContractFailed += (sender, e) =>
{
    Console.WriteLine(e.Message);
    e.SetHandled();
};
```

代码段 CodeContractsSamples/Program.cs

19.1.1 前提条件

前提条件检查传递给方法的参数。Requires() 和 Requires<TException>() 方法是可以用 Contract 类定义的前提条件。使用 Requires() 方法时, 必须传送一个布尔值, 第二个参数是一个可选的消息字符串, 当条件不满足时显示该消息。下面的示例要求, 参数 min 应小于等于参数 max。



可从
wrox.com
下载源代码

```
static void MinMax(int min, int max)
{
    Contract.Requires(min <= max);
    //...
}
```

代码段 CodeContractsSamples/Program.cs

如果参数 o 是空, 下面的协定就抛出一个 ArgumentNullException 异常。如果事件处理程序把 ContractFailed 事件设置为 handled, 就不抛出该异常。另外, 如果配置了 Assert on Contract Failure, 就执行 Trace.Assert() 方法停止程序, 而不是抛出所定义的异常。

```
static void Preconditions(object o)
{
    Contract.Requires<ArgumentNullException>(o != null,
        "Preconditions, o may not be null");
    //...
```

因为 Requires<TException>() 方法没有用 [Conditional ("CONTRACT_FULL")] 特性注解, 并且在 DEBUG 符号上它也没有条件, 所以这个运行时检查可以在任意情况下进行。如果条件不满足, Requires<TException>() 方法就抛出所定义的异常。

在许多遗留代码中, 参数常常用 if 语句检查, 如果条件不满足, 就抛出一个异常。而使用代码协定, 就不需要重写验证代码, 只需添加一行代码即可:

```
static void PreconditionsWithLegacyCode(object o)
{
    if (o == null) throw new ArgumentNullException("o");
    Contract.EndContractBlock();
}
```

EndContractBlock() 方法指定, 上述代码应作为一个合同处理。如果还使用了其他合同语句, 就不需要 EndContractBlock() 方法。

为了检查用作参数的集合, Contract 类提供了 Exists() 和 ForAll() 方法。ForAll() 方法检查集合中的每一项, 看看它们是否满足条件。在示例中, 检查集合中的每一项的值是否小于 12。使用 Exists() 方法可以检查集合中的任意一项是否满足条件。

```
static void ArrayTest(int[] data)
{
    Contract.Requires(Contract.ForAll(data, i => i < 12));
}
```

Exists()和 ForAll()方法都有一个重载版本,其中可以给该重载版本传递两个参数 fromInclusive 和 toExclusive,而不是传递 IEnumerable<T>。把一个数值范围(除去 toExclusive)传递给第 3 个参数定义的委托 Predicate<int>。Exists()和 ForAll()方法可以用于前提条件、后置条件和常量。

19.1.2 后置条件

后置条件定义了方法执行完后共享数据和返回值的保证。尽管后置条件定义了关于返回值的一些保证,但它们必须放在方法的开头;所有的协定要求都必须放在方法的开头。

Ensure()和 EnsuresOnThrow<TException>()方法是后置条件。下面的合同确保变量 sharedState 在方法执行完后小于 6。该值在方法执行期间可以改变。



可从
wrox.com
下载源代码

```
private static int sharedState = 5;
static void Postcondition()
{
    Contract.Ensures(sharedState < 6);
    sharedState = 9;
    Console.WriteLine("change sharedState invariant {0}", sharedState);
    sharedState = 3;
    Console.WriteLine("before returning change it to a valid value {0}",
        sharedState);
}
```

代码段 CodeContractsSamples/Program.cs

使用 EnsuresOnThrow<TException>()方法,可以保证如果抛出了指定的异常,共享状态就满足某条件。

为了保证返回某个值,可以对 Ensure()方法的协定使用特定的值 Result<T>。这里的结果是 int 类型,因为它是用泛型类型 T 给 Result()方法定义的。Ensure()方法的协定保证返回值小于 6。

```
static int ReturnValue()
{
    Contract.Ensures(Contract.Result<int>() < 6);
    return 3;
}
```

还可以比较新旧值。为此应使用 OldValue<T>()方法,它返回在方法入口给变量传递的初始值。下面的协定确保,返回的结果(Contract.Result<int>()方法返回的)大于参数 x(对应于 Contract.OldValue<int>()方法)中的旧值。

```
static int ReturnLargerThanInput(int x)
{
    Contract.Ensures(Contract.Result<int>() > Contract.OldValue<int>(x));
    return x + 3;
}
```

如果方法的返回值用 out 修饰符来修饰,而不仅仅使用 return 语句,就可以用 ValueAtResult()方法定义条件。下面的协定指定,变量 x 在返回时必须大于 5 且小于 20,变量 y 与 5 的取模结果在

返回时必须等于 0。

```
static void OutParameters(out int x, out int y)
{
    Contract.Ensures(Contract.ValueAtReturn<int>(out x) > 5 &&
        Contract.ValueAtReturn<int>(out x) < 20);
    Contract.Ensures(Contract.ValueAtReturn<int>(out y) % 5 == 0);
    x = 8;
    y = 10;
}
```

19.1.3 常量

常量为方法生命周期中的变量定义了协定。Contract.Requires()方法定义了输入要求, Contract.Ensure()方法定义了方法结束时的要求。Contract.Invariant()方法定义了在整个生命周期中都必须满足的条件。



可从
wrox.com
下载源代码

```
static void Invariant(ref int x)
{
    Contract.Invariant(x > 5);
    x = 3;
    Console.WriteLine("invariant value: {0}", x);
    x = 9;
}
```

代码段 CodeContractsSamples/Program.cs

19.1.4 接口的协定

对于接口,可以定义方法、属性和事件,派生自该接口的类必须实现这些方法、属性和事件。在接口的声明中,不能定义接口的实现方式,现在可以使用代码协定来实现。

看看下面的接口。IPerson 接口定义了 FirstName、LastName 和 Age 属性以及 ChangeName()方法。这个接口的特殊之处是 ContractClass 特性。这个特性应用于 IPerson 接口,并指定 PersonContract 类用作这个接口的代码协定。



可从
wrox.com
下载源代码

```
[ContractClass(typeof(PersonContract))]
public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }
    int Age { get; set; }
    void ChangeName(string firstName, string lastName);
}
```

代码段 CodeContractsSamples/IPerson.cs

PersonContract 类实现了 IPerson 接口,并为所有的成员定义了代码协定。PureAttribute 特性表示,方法或属性不能改变类实例的状态。这用属性的 get 访问器定义,还可以用不允许改变状态的所有方法来定义。FirstName 和 LastName 属性的 get 访问器也用 Contract.Result()方法指定,结果必须是一个字符串。Age 属性的 get 访问器定义了一个后置条件,并确保返回值在 0~120 之间。FirstName 和 LastName 属性的 set 访问器要求,传递的值非空。Age 属性的 set 访问器定义了一个后置条件,要求传递的值在 0~120 之间。



可从
wrox.com
下载源代码

```
[ContractClassFor(typeof(IPerson))]
public sealed class PersonContract : IPerson
{
    string IPerson.FirstName
    {
        [Pure] get { return Contract.Result<String>(); }
        set { Contract.Requires(value != null); }
    }
    string IPerson.LastName
    {
        [Pure] get { return Contract.Result<String>(); }
        set { Contract.Requires(value != null); }
    }
    int IPerson.Age
    {
        [Pure]
        get
        {
            Contract.Ensures(Contract.Result<int>() >= 0 &&
                Contract.Result<int>() < 121);
            return Contract.Result<int>();
        }
        set
        {
            Contract.Requires(value >= 0 && value < 121);
        }
    }
    void IPerson.ChangeName(string firstName, string lastName)
    {
        Contract.Requires(firstName != null);
        Contract.Requires(lastName != null);
    }
}
```

代码段 CodeContractsSamples/PersonContracts

现在实现 **IPerson** 接口的类必须满足所有的协定要求。**Person** 类是满足协定的接口的一个简单实现。



可从
wrox.com
下载源代码

```
public class Person : IPerson
{
    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public int Age { get; set; }

    public void ChangeName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

代码段 CodeContractsSamples/Person.cs

使用 `Person` 类时，也必须满足协定。例如，不允许给属性赋予空值：



可从
wrox.com
下载源代码

```
var p = new Person { FirstName = "Tom", LastName = null }; // contract error
```

代码段 CodeContractsSamples/Program.cs

也不允许给 `Age` 属性赋予无效值：

```
var p = new Person { FirstName = "Tom", LastName = "Turbo" };  
p.Age = 133; // contract error
```

19.2 跟踪

利用跟踪功能可以从正在运行的应用程序中查看消息。为了获得关于正在运行的应用程序的信息，可以在调试器中启动应用程序。在调试过程中，可以单步执行应用程序，在特定的代码行上设置断点，并在满足某些条件时设置断点。调试的问题是包含发布代码的程序与包含调试代码的程序以不同的方式运行。例如，程序在断点处停止运行时，应用程序的其他线程也会挂起。另外，在发布版本中，编译器生成的输出进行了优化，因此会产生不同的效果。此时也需要从发布版本中获得信息。跟踪消息要写入调试代码和发布代码中。

下面的场景描述了跟踪功能的作用。在部署应用程序后，它运行在一个系统中时没有问题，而在另一个系统上很快出现了问题。在出问题的系统上打开详细的跟踪功能，就会获得应用程序中所出现问题的详细信息。在运行时没有问题的系统上，将跟踪功能配置为把错误消息重定向到 Windows 事件日志系统中。系统管理员会查看重要的错误，跟踪功能的系统开销非常小，因为仅在需要时配置跟踪级别。

跟踪体系架构有 4 个主要部分：

- 源是跟踪信息的源头，使用源可以发送跟踪消息。
- 开关定义了要记录的信息级别。例如，可以只请求错误信息或详细的信息。
- 跟踪侦听器定义了写入跟踪消息的位置。
- 侦听器可以有相关的筛选器。筛选器定义了侦听器应写入哪些跟踪消息。这样，就可以给同一个源头使用不同的侦听器，写入不同级别的信息。

图 19-2 显示了主要的跟踪类和它们在 Visual Studio 类图中的连接方式。`TraceSource` 类使用一个开关来定义要记录的信息。`TraceSource` 类有一个相关联的 `TraceListenerCollection` 类，它指定写入跟踪消息的位置。该集合由 `TraceListener` 对象组成，每个侦听器都连接到一个 `TraceFilter` 类。

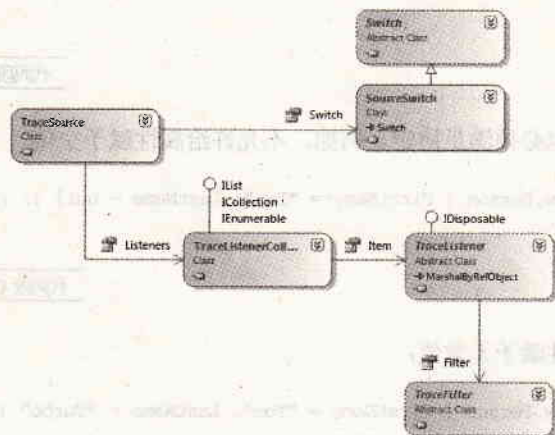


图 19-2

有几种.NET技术使用跟踪源,只需打开跟踪源,就能看到发生了什么情况。例如,此外, WPF 定义的源有 `System.Windows.Data`、`System.Windows.RoutedEvent`、`System.Windows.Markup`、`System.Windows.Media.Animation`。而在 WPF 中,打开跟踪功能时,不仅需要配置侦听器,还要在 `HKEY_CURRENT_USER\Software\Microsoft\Tracing\WPF` 注册键中把一个新的 DWORD 设置为 `ManagedTracing`,其值设置为 1。

`System.Net` 名称空间中的类使用跟踪源 `System.Net`, WCF 使用跟踪源 `System.ServiceModel` 和 `System.ServiceModel.MessageLogging`。关于 WCF 跟踪的内容参见第 43 章。

19.2.1 跟踪源

用 `TraceSource` 类可以写入跟踪消息。跟踪需要编译器设置的 `Trace` 标志。在 Visual Studio 项目中, `Trace` 标志在调试版本和发布版本中已进行了默认设置,但可以通过项目的 `Build` 属性修改它。

与写入跟踪消息的 `Trace` 相比, `TraceSource` 类较难使用,但它提供的选项较多。

要写入跟踪消息,需要创建一个新的 `TraceSource` 实例。在构造函数中,定义了跟踪源的名称。`TraceInformation()`方法在跟踪输出中写入一条信息型消息。`TraceEvent()`方法不只写入信息型消息,还需要一个 `TraceEventType` 类型的枚举值,来定义跟踪消息的类型。`TraceEventType.Error` 把消息指定为一条错误消息。可以用跟踪开关将它定义为只查看错误消息。`TraceEvent()`方法的第二个参数需要一个标识符。ID 可以用于应用程序自身。例如,可以使用 id 1 进入某个方法,用 id 2 退出某个方法。因为 `TraceEvent()`方法是重载的,所以只需要 `TraceEventType` 和 ID 这两个参数。使用重载方法的第 3 个参数,可以传递写入跟踪输出的消息。`TraceEvent()`方法还可以传递格式字符串和任意数量的参数,其方式与 `Console.WriteLine()`方法相同。`TraceInformation()`方法只是调用标识符为 0 的 `TraceEvent()`方法。`TraceInformation()`方法是 `TraceEvent()`方法的一个简化版本。在 `TraceData()`方法中,

可以传递任意对象,例如,异常实例,来替代消息。要确保数据由侦听器写入,且不存储在内存中,就需要执行 `Flush()` 方法。如果不再需要跟踪源,就可以调用 `Close()` 方法,它关闭与跟踪源相关的所有侦听器。`Close()` 方法也会执行 `Flush()` 方法。



可从
wrox.com
下载源代码

```
public class Program
{
    internal static TraceSource trace =
        new TraceSource("Wrox.ProCSharp.Instrumentation");

    static void TraceSourceDemo1()
    {
        trace.TraceInformation("Info message");

        trace.TraceEvent(TraceEventType.Error, 3, "Error message");
        trace.TraceData(TraceEventType.Information, 2, "data1", 4, 5);
        trace.Flush();
        trace.Close();
    }
}
```

代码段 TracingDemo/Program.cs



可以在应用程序中使用不同的跟踪源。为不同的库定义不同的跟踪源,这样就可以为应用程序的不同部分打开不同的跟踪级别。要使用跟踪源,必须知道它的名称。跟踪源的常用名称与程序集的名称相同。

`TraceEventType` 枚举作为一个参数传递给 `TraceEvent()` 方法,该枚举定义了下面的级别,来指定问题的严重程度: `Verbose`、`Information`、`Warning`、`Error` 和 `Critical`。`Critical` 定义了致命错误或使应用程序崩溃的错误; `Error` 表示可恢复的错误。`Verbose` 级别的跟踪消息可给出详细的调试信息。`TraceEventType` 还定义了操作级别: `Start`、`Stop`、`Suspend` 和 `Resume`。这些级别在逻辑操作中定义了及时事件。

代码在编写时没有显示任何跟踪消息,因为与跟踪源相关的开关是关闭的。

19.2.2 跟踪开关

要启用或禁用跟踪消息,可以配置一个跟踪开关。跟踪开关是派生自抽象基类 `Switch` 的类。派生类是 `BooleanSwitch`、`TraceSwitch` 和 `SourceSwitch`。`BooleanSwitch` 类可以打开和关闭,其他两个类提供了由 `SourceLevels` 枚举定义的范围级别。要配置跟踪开关,必须知道与 `SourceLevel` 枚举相关的值。`SourceLevels` 定义的值有 `Off`、`Error`、`Warning`、`Info` 和 `Verbose`。

设置 `TraceSource` 类的 `Switch` 属性,可以用编程方式关联跟踪开关。这里关联的开关是 `SourceSwitch` 类型,其名称是 `Wrox.ProCSharp.Instrumentation`,级别为 `Verbose`。



可从
wrox.com
下载源代码

```
internal static SourceSwitch traceSwitch =
    new SourceSwitch("Wrox.ProCSharp.Instrumentation")
    { Level = SourceLevels.Verbose };

internal static TraceSource trace =
    new TraceSource("Wrox.ProCSharp.Instrumentation")
    { Switch = traceSwitch };
```

代码段 TracingDemo/Program.cs

把级别设置为 **Verbose**，表示应写入所有的跟踪消息。如果把值设置为 **Error**，就应只显示错误消息。把值设置为 **Information**，表示显示错误、警告和信息型消息。在 **Output** 窗口中运行调试器，再次写入跟踪消息，就可以看到这些消息。

通常，要改变开关级别，而不希望重新编译应用程序，则最好只改变配置。跟踪源可以在应用程序配置文件中配置。跟踪的配置在 `<system.diagnostics>` 元素中完成。跟踪源在 `<source>` 元素中定义为 `<sources>` 的一个子元素。配置文件中跟踪源的名称必须精确匹配程序代码中跟踪源的名称。这里跟踪源的开关类型是 `System.Diagnostics.SourceSwitch`，名称是 `MySourceSwitch`。该开关在 `<switches>` 部分中定义，开关的级别设置为 **Verbose**。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Instrumentation" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch" />
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

代码段 TracingDemo/App.config

现在，要修改跟踪级别，只需修改配置文件，而无需重新编译代码。修改配置文件后，必须重新启动应用程序。

在调试会话中运行时，当前的跟踪消息只写入 Visual Studio 的 **Output** 窗口中。添加跟踪侦听器可以改变这种情况。

19.2.3 跟踪侦听器

默认情况下，把跟踪信息写入 Visual Studio 调试器的 **Output** 窗口中。只要改变应用程序的配置，就可以将跟踪输出重定向到不同的位置。

跟踪信息应写入什么位置由跟踪侦听器确定。跟踪侦听器派生自抽象基类 `TraceListener`。 .NET 包含的几个跟踪侦听器会把跟踪事件写入不同的目标位置。对于基于文件的跟踪侦听器，使用基类 `TextWriterTraceListener`，写入 XML 文件的派生类 `XmlWriterTraceListener`，以及写入有分隔符的文件的派生类 `DelimitedListTraceListener`。写入事件日志可使用 `EventLogTraceListener` 类或 `EventProviderTraceListener` 类来完成。`EventProviderTraceListener` 类使用自 Windows Vista 以来新增的事件文件格式。还可以合并 Web 跟踪和 `System.Diagnostics` 跟踪功能，使用 `WebPageTraceListener` 把 `System.Diagnostics` 跟踪信息也写入 Web 跟踪文件 `trace.axd` 中。

.NET Framework 发布了许多可写入跟踪信息的侦听器。如果侦听器不满足用户的需要，就可以从基类 `TraceListener` 中派生一个类，从而创建自定义侦听器。使用自定义侦听器，可以将跟踪信息写入 Web 服务，将消息写入手机等。其实手机在闲暇时间接收到数百条消息并没有那么有趣。使用 **Verbose** 跟踪级别，这会变得相当昂贵。

创建一个侦听器对象，并将它赋予 `TraceSource` 类的 `Listeners` 属性，就可以用编程方式配置跟

踪侦听器。但是，只改变配置，就定义另一个侦听器会比较有趣。

可以把侦听器配置为<source>元素的子元素。在侦听器中，可以定义侦听器类的类型，并使用 initializeData 指定侦听器的输出应写入什么位置。这里的配置将 XmlWriterTraceListener 类定义为写入 demotrace.xml 文件中，并将 DelimitedListTraceListener 类定义为写入 demotrace.txt 文件中。



```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
            type="System.Diagnostics.XmlWriterTraceListener"
            traceOutputOptions="None"
            initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" delimiter=":"
            type="System.Diagnostics.DelimitedListTraceListener"
            traceOutputOptions="DateTime, ProcessId"
            initializeData="c:/logs/mytrace.txt" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

代码段 TracingDemo/App.config

在侦听器中，还可以指定把哪些额外信息写入跟踪日志中。这些信息和 XML 特性 traceOutputOptions 一起由 TraceOptions 枚举定义。该枚举定义了 Callstack、DateTime、LogicalOperationStack、ProcessId、ThreadId 和 None。需要的信息可以用逗号分隔符添加到 XML 特性 traceOutputOptions 中，如带分隔符的跟踪侦听器所示。

下面是 DelimitedListTraceListener 类中带分隔符的文件输出，包括进程 ID 和日期/时间：

```
"Wrox.ProCSharp.Instrumentation":Information:0:"Info message"::
5288:""::"2009-10-11T10:35:55.8479950Z"::
"Wrox.ProCSharp.Instrumentation":Error:3:"Error message"::5288:""::
"2009-10-11T10:35:55.8509257Z"::
"Wrox.ProCSharp.Instrumentation":Information:2::
"data1","4","5":5288:""::"2009-10-11T10:35:55.8519026Z"::
```

XmlWriterTraceListener 类中的 XML 输出总是包含计算机名、进程 ID、线程 ID、消息、创建时间、源和活动 ID。其他字段，如调用栈、逻辑操作栈、时间戳等，都依赖于跟踪输出选项。



可以使用 XmlDocument 类和 XPathNavigator 类分析 XML 文件中的内容。这些类详见第 33 章。

如果侦听器应由多个跟踪源使用,就可以将侦听器配置添加到<sharedListeners>元素中,它独立于跟踪源。配置为共享侦听器的侦听器名称必须从跟踪源的侦听器中引用:



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
            type="System.Diagnostics.XmlWriterTraceListener"
            traceOutputOptions="None"
            initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="delimitedListener" delimiter=":"
        type="System.Diagnostics.DelimitedListTraceListener"
        traceOutputOptions="DateTime, ProcessId"
        initializeData="c:/logs/mytrace.txt" />
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

代码段 TracingDemo/app.config

19.2.4 筛选器

每个侦听器都有一个 Filter 属性,它定义了侦听器是否应写入跟踪消息。例如,多个侦听器可以与同一个跟踪源一起使用。其中一个侦听器将详细消息写入日志文件中,另一个侦听器将错误消息写入事件日志中。在侦听器写入跟踪信息之前,调用相关筛选器对象的 ShouldTrace()方法,确定是否应写入跟踪信息。

筛选器是派生自抽象基类 TraceFilter 的类。.NET 4 提供了两个已实现的筛选器: SourceFilter 和 EventTypeFilter。使用 SourceFilter 筛选器,可以指定只从特定的源中写入跟踪信息。EventTypeFilter 筛选器是对开关功能的扩展。使用开关,可以根据跟踪的严重程度,确定事件源是否应将跟踪消息写入侦听器中。如果写入跟踪消息,侦听器就可以使用筛选器确定是否应写入该消息。

改变的配置现在确定,只有严重程度为警告或更高,带分隔符的侦听器才应写入跟踪消息,因为定义了 EventTypeFilter 筛选器。XML 侦听器指定了 SourceFilter 筛选器,只接收源 Wrox.ProCSharp.Tracing 中的跟踪消息。如果定义了大量跟踪源,以便将跟踪消息写入同一个侦听器中,就可以修改该侦听器的配置,只处理指定源中的跟踪消息。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
```

```

<source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
  switchType="System.Diagnostics.SourceSwitch">
  <listeners>
    <add name="xmlListener" />
    <add name="delimitedListener" />
  </listeners>
</source>
</sources>
<sharedListeners>
  <add name="delimitedListener" delimiter=":"
    type="System.Diagnostics.DelimitedListTraceListener"
    traceOutputOptions="DateTime, ProcessId"
    initializeData="c:/logs/mytrace.txt">
    <filter type="System.Diagnostics.EventTypeFilter"
      initializeData="Warning" />
  </add>
  <add name="xmlListener"
    type="System.Diagnostics.XmlWriterTraceListener"
    traceOutputOptions="None"
    initializeData="c:/logs/mytrace.xml">
    <filter type="System.Diagnostics.SourceFilter"
      initializeData="Wrox.ProCSharp.Instrumentation" />
  </add>
</sharedListeners>
<switches>
  <add name="MySourceSwitch" value="Verbose"/>
</switches>
</system.diagnostics>
</configuration>

```

代码段 TracingDemo/App.config

跟踪体系结构可以扩展。可以编写一个派生自 `TraceListener` 基类的自定义侦听器，同样，也可以创建一个派生自 `TraceFilter` 的自定义筛选器。因此，可以创建一个筛选器，根据时间、以后发生的异常或天气，指定写入跟踪消息。

19.2.5 相关性

在跟踪日志中，可以用几种方式查看不同方法的关系。要查看跟踪事件的调用栈，只需进行一个配置，就可以用 XML 侦听器跟踪调用栈。还可以定义一个能在日志消息中显示的逻辑调用栈。也可以定义活动，以映射跟踪消息。

为了显示调用栈和带跟踪消息的逻辑调用栈，可以把 `XmlWriterTraceListener` 类配置为对应的 `traceOutputOptions`。MSDN 文档给出了可以配置为通过这个侦听器进行跟踪的所有其他选项的详细内容。



可从
wrox.com
下载源代码

```

<sharedListeners>
  <add name="xmlListener" type="System.Diagnostics.XmlWriterTraceListener"
    traceOutputOptions=" LogicalOperationStack, Callstack "
    initializeData="c:/logs/mytrace.xml">
  </add>
</sharedListeners>

```

代码段 TracingDemo/App.config

为了查看相关性和逻辑调用栈,可在 Main()方法中设置 ActivityID 属性,把一个新的活动 ID 赋予 CorrelationManager。TraceEventType.Start 和 TraceEventType.Stop 类型的事件在 Main()方法的开始和结束时触发。另外,使用 StartLogicalOperation()和 StopLogicalOperation()方法分别启动和停止一个逻辑操作“Main”。



可从
wrox.com
下载源代码

```
static void Main()
{
    // start a new activity
    if (Trace.CorrelationManager.ActivityId == Guid.Empty)
    {
        Guid newGuid = Guid.NewGuid();
        Trace.CorrelationManager.ActivityId = newGuid;
    }
    trace.TraceEvent(TraceEventType.Start, 0, "Main started");

    // start a logical operation
    Trace.CorrelationManager.StartLogicalOperation("Main");

    TraceSourceDemol();
    StartActivityA();
    Trace.CorrelationManager.StopLogicalOperation();
    Thread.Sleep(3000);
    trace.TraceEvent(TraceEventType.Stop, 0, "Main stopped");
}
```

代码段 TracingDemo/App.config

在 Main()方法中调用的 StartActivityA()方法会把 CorrelationManager 的 ActivityId 设置为一个新的 GUID,以新建一个活动。在该活动停止之前,把 CorrelationManager 的 ActivityId 重置为以前的值。这个方法调用 Foo()方法,并用 Task.Factory.StartNew()方法新建一个任务。创建了这个任务后,就可以查看不同的线程如何显示在跟踪查看器中。



任务可参见第 20 章。

```
private static void StartActivityA()
{
    Guid oldGuid = Trace.CorrelationManager.ActivityId;
    Guid newActivityId = Guid.NewGuid();
    Trace.CorrelationManager.ActivityId = newActivityId;

    Trace.CorrelationManager.StartLogicalOperation("StartActivityA");
    trace.TraceEvent(TraceEventType.Verbose, 0,
        "starting Foo in StartNewActivity");
    Foo();

    trace.TraceEvent(TraceEventType.Verbose, 0,
        "starting a new task");
    Task.Factory.StartNew(WorkForATask);

    Trace.CorrelationManager.StopLogicalOperation();
    Trace.CorrelationManager.ActivityId = oldGuid;
}
```

从 `StartActivityA()` 方法内部启动的 `Foo()` 方法启动了一个新的逻辑操作。逻辑操作 `Foo` 在逻辑操作 `StartActivityA` 内部启动。

```
private static void Foo()
{
    Trace.CorrelationManager.StartLogicalOperation("Foo operation");

    trace.TraceEvent(TraceEventType.Verbose, 0, "running Foo");

    Trace.CorrelationManager.StopLogicalOperation();
}
```

在 `StartActivityA()` 方法内部创建的任务运行 `WorkForATask()` 方法。这里仅把包含启动、停止信息和详细信息的跟踪事件写入跟踪日志中。

```
private static void WorkForATask()
{
    trace.TraceEvent(TraceEventType.Start, 0, "WorkForATask started");

    trace.TraceEvent(TraceEventType.Verbose, 0, "running WorkForATask");

    trace.TraceEvent(TraceEventType.Stop, 0, "WorkForATask completed");
}
```

为了分析跟踪信息，可以启动服务跟踪查看器工具 `svctraceviewer.exe`。这个工具主要用于分析 WCF 跟踪，但它也可以用于分析用 `XmlWriterTraceListener` 类写入的任何跟踪信息。图 19-3 显示了 Activity 屏幕中的活动，每个活动都在右边的屏幕上显示了事件。选择一个事件时，可以把显示器设置为在 XML 或格式化的视图中显示完整的消息。在格式化的视图中，基本信息、应用程序数据、逻辑操作栈和调用栈都进行了很好的格式化。

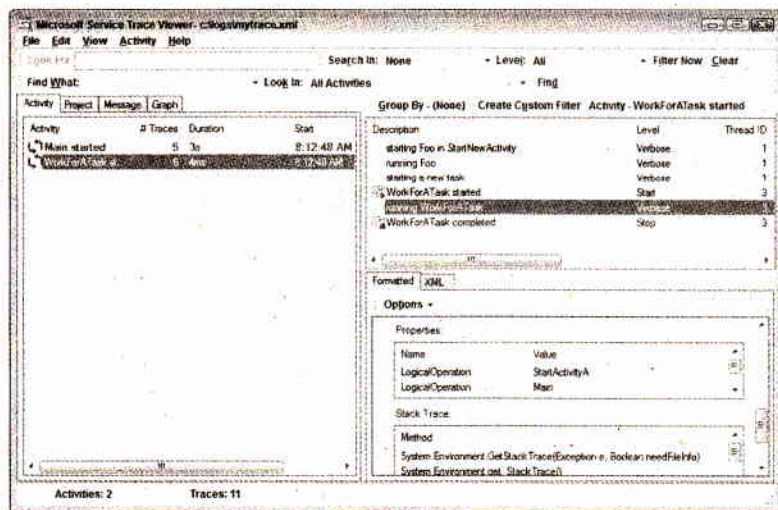


图 19-3

图 19-4 显示了图形视图，其中不同的进程或线程可以选择性地显示在各自的泳道中。用 `TaskFactory` 类新建线程时，会显示第二个区域，以选择线程视图。

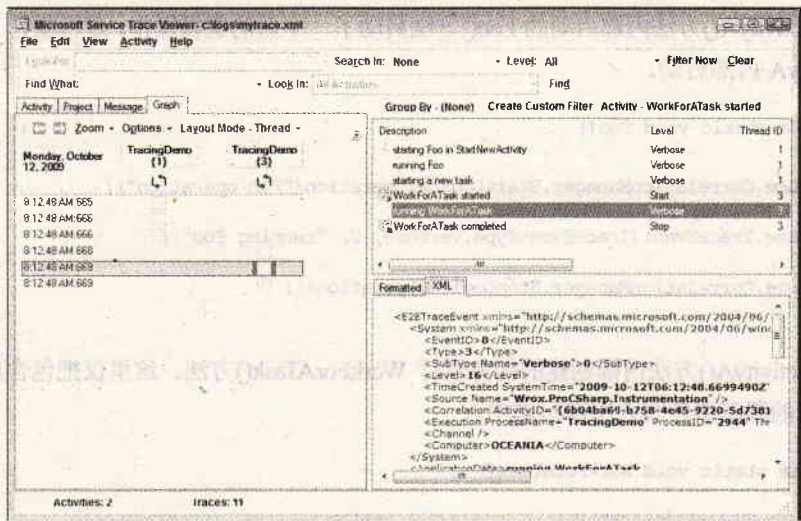


图 19-4

19.3 事件日志

系统管理员使用事件查看器获得关于系统和应用程序正常运行的重要消息和信息型消息。应将应用程序的错误消息写入事件日志，从而可以用事件查看器读取这些信息。

如果配置了 `EventLogTraceListener` 类，跟踪消息就可以写入事件日志中。`EventLogTraceListener` 类有一个与之相关联的 `EventLog` 对象，可以方便写入事件日志项。也可以使用 `EventLog` 类读写事件日志。

本节介绍如下内容：

- 事件日志体系结构
- `System.Diagnostics` 名称空间中用于事件日志的类
- 在服务和其他应用程序类型中添加事件日志
- 用 `EventLog` 类的 `EnableRaisingEvents` 属性创建事件日志侦听器
- 使用资源文件定义消息

图 19-5 显示了使用分布式 COM 访问失败的一个日志项。



图 19-5

对于自定义事件日志，可以使用 `System.Diagnostics` 名称空间中的类。

19.3.1 事件日志体系结构

事件日志信息存储在几个日志文件中。最重要的日志文件是应用程序、安全性和系统日志文件。查看事件日志服务的注册表配置，会注意到 `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog` 下的几项带有指向特定文件的配置。系统日志文件用于系统驱动程序和设备驱动程序。应用程序和服务则将事件写入应用程序日志中。安全性日志是应用程序的只读日志。操作系统的审计功能使用安全性日志。每个应用程序还可以创建自定义类别和日志文件，在其中写入事件日志项。例如，`Media Center` 就可以这么做。

使用管理工具“事件查看器”就可以读取这些事件。要启动事件查看器，可以直接在 `Visual Studio` 的 `Server Explorer` 窗口中右击 `Event Logs` 项，从弹出的上下文菜单中选择 `Launch Event Viewer` 命令。事件查看器如图 19-6 所示。

在事件日志中，可以看到如下信息：

- **类型**——该类型可以是 `Information`、`Warning` 或 `Error`。`Information` 是不常成功的操作；`Warning` 表示不太重要的问题；`Error` 表示重要问题。其他类型有 `FailureAudit` 和 `SuccessAudit`，但这些类型仅用于安全性日志。
- **日期**——`Date and Time` 显示事件发生的日期和时间。
- **源**——`Source` 是记录事件的软件的名称。应用程序日志的源在如下注册表键中配置：

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\Application\
[ApplicationName]
```

在这个键下，`EventMessageFile` 的值配置为指向一个源 DLL，该 DLL 保存了错误消息。

- **事件 ID**——`Event` 标识符指定某条事件消息。
- **类别**——`Category` 可以定义为允许在使用 `Event Viewer` 时筛选事件日志。类别可以通过事件源来定义。

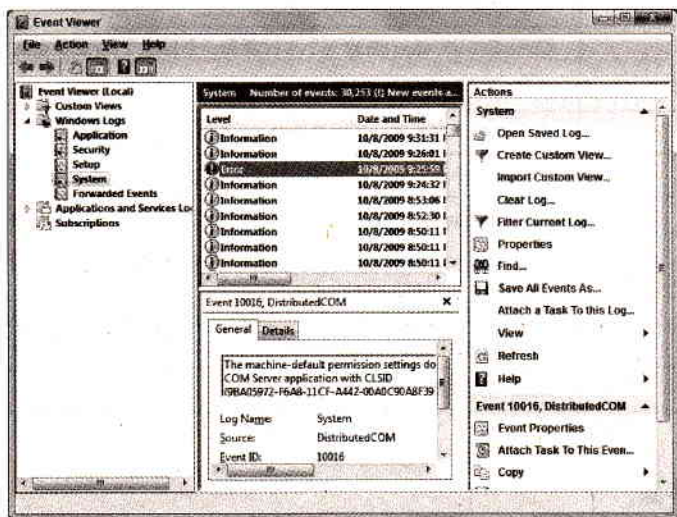


图 19-6

19.3.2 事件日志类

写入事件日志时，可以使用两个不同的 Windows API。在 Windows Vista 发布后可用的一个 API 由 System.Diagnostics.Eventing 名称空间中的类封装，另一个封装类在 System.Diagnostics 名称空间中。System.Diagnostics 名称空间中的一些类用于事件日志，如表 19-1 所示。

表 19-1

类	说 明
EventLog	使用 EventLog 类可以读写事件日志中的项，并非为事件源建立应用程序
EventLogEntry	EventLogEntry 类表示事件日志中的唯一一项。使用 EventLogEntryCollection 类可以遍历 EventLogEntry 项
EventLogInstaller	EventLogInstaller 类是 EventLog 组件的安装程序。EventLogInstaller 类调用 EventLog.CreateEventSource()方法创建事件源
EventLogTraceListener	使用 EventLogTraceListener 类有助于把跟踪信息写入事件日志中。这个类实现了抽象类 TraceListener

事件日志的核心是 EventLog 类。这个类的成员如表 19-2 所示。

表 19-2

EventLog 类的成员	说 明
Entries	使用 Entries 属性可以读取事件日志。Entries 属性返回一个 EventLogEntryCollection，它包含的 EventLogEntry 对象存储了事件的相关信息。不需要调用 Read()方法。只要访问这个属性，就会填充该集合
Log	使用 Log 属性可指定用于读写事件日志的记录
LogDisplayName	LogDisplayName 是一个只读属性，返回日志的显示名称
MachineName	使用 MachineName 可以指定在哪个系统上读写日志项
Source	Source 属性指定要写入日志项的源
CreateEventSource()	CreateEventSource()新建一个事件源和一个日志文件(如果在该方法中指定了新的日志文件)
DeleteEventSource()	要删除一个事件源，可以调用 DeleteEventSource()方法
SourceExists()	在创建事件源之前，可以使用这个元素验证该源是否已存在
WriteEntry() WriteEvent()	用 WriteEntry()或 WriteEvent()方法可以写入事件日志项。WriteEntry()方法比较简单，只需传递一个字符串即可。WriteEvent()方法比较灵活，因为用户可以使用独立于应用程序的消息文件，还可以支持本地化
Clear()	Clear()方法删除事件日志中的所有项
Delete()	Delete()方法删除一个完整的事件日志

19.3.3 创建事件源

在写入事件之前，必须创建一个事件源。为此，可以使用 EventLog 类或 EventLogInstaller 类的 CreateEventSource()方法。在创建事件源时需要管理员权限，所以最好用一个安装程序定义新事件源。



第 17 章介绍了如何创建安装程序。

下面的例子验证了事件日志源 `EventLogDemoApp` 已存在。如果它不存在，就实例化一个 `EventSourceCreationData` 类型的对象，该对象定义源名 `EventLogDemoApp` 和日志名 `ProCSharpLog`。这里，该源的所有事件都写入 `ProCSharpLog` 事件日志。默认为应用程序日志。



可从
wrox.com
下载源代码

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";

if (!EventLog.SourceExists(sourceName))
{
    var eventSourceData = new EventSourceCreationData(sourceName,
        logName);

    EventLog.CreateEventSource(eventSourceData);
}
```

代码段 `EventLogDemo/Program.cs`

事件源的名称是写入事件的应用程序的标识符。系统管理员在读取日志时，该信息有助于识别事件日志项，将它们映射到应用程序类别上。事件日志源的名称包括：如用于性能监控器的 `LoadPerf`，用于 Microsoft SQL Server 的 `MSSQLSERVER`，用于 Windows 安装程序的 `MsiInstaller`，以及 `Winlogon`、`Tcpip`、`TimeService` 等。

为事件日志设置名称 `Application`，会把事件日志项写入应用程序日志中。也可以创建自己的日志，方法是指定不同的应用程序日志名。日志文件位于 `<windows>\System32\WinEvt\Logs` 目录中。

使用 `EventSourceCreationData`，还可以为事件日志指定更多的特性，如表 19-3 所示。

表 19-3

EventSourceCreationData	说 明
Source	Source 属性可获取或设置事件源的名称
LogName	LogName 属性定义了事件日志项写到什么日志中。默认为应用程序日志
MachineName	使用 MachineName 属性可以定义读写日志项的系统
CategoryResourceFile	使用 CategoryResourceFile 属性可以定义类别的资源文件。类别有助于筛选单一源中的事件日志项
CategoryCount	CategoryCount 属性定义了类别资源文件中的类别个数
MessageResourceFile	除了指定程序中应写入事件日志中的消息之外该程序用来写入事件；消息还可以在一个资源文件中定义，该文件由 MessageResourceFile 属性指定。该资源文件中的信息是可以本地化的
ParameterResourceFile	资源文件中的消息可以带参数。参数可以用一个资源文件中定义的字符串替代，该资源文件由 ParameterResourceFile 属性指定

19.3.4 写入事件日志

要写入事件日志项，可以使用 `EventLog` 类的 `WriteEntry()` 或 `WriteEvent()` 方法。`EventLog` 类有一

一个静态方法 `WriteEntry()` 和一个实例方法 `WriteEntry()`。静态方法 `WriteEntry()` 的参数是事件源。该源也可以用 `EventLog` 类的构造函数设置。在下面的构造函数中，定义了日志名、本地计算机和事件源名。接着将消息作为 `WriteEntry()` 方法的第一个参数，写入 3 个事件日志项。`WriteEntry()` 方法是重载版本。指定的第二个参数是 `EventLogEntryType` 类型的枚举。使用 `EventLogEntryType` 类型，可以指定事件日志项的严重程度。其值可以是 `Information`、`Warning` 和 `Error`，以及用于审计的 `FailureAudit` 和 `SuccessAudit`。根据该类型，在 `Event Viewer` 窗口中会显示不同的图标。第 3 个参数指定与应用程序相关的事件 ID，它可以由应用程序使用。另外，还可以传递与应用程序相关的二进制数据和类别。



```
using (var log = new EventLog(logName, ".", sourceName))
{
    log.WriteEntry("Message 1");
    log.WriteEntry("Message 2", EventLogEntryType.Warning);
    log.WriteEntry("Message 3", EventLogEntryType.Information, 33);
}
```

代码段 EventLogDemo/Program.cs

19.3.5 资源文件

除了在 C# 代码中为事件日志定义消息，把它传递给 `WriteEntry()` 方法之外，还可以创建消息资源文件，在该资源文件中定义消息，将消息的标识符传递给 `WriteEvent()` 方法。资源文件还支持本地化。



消息资源文件是本地资源文件，它与 .NET 资源文件没有共同之处。NET 资源文件详见第 22 章。

资源文件是一个文本文件，扩展名是 `.mc`。这个文件用于定义消息的语法非常严格。示例文件 `EventLogMessages.mc` 包含 4 个类别，之后是事件消息。每个消息都有一个 ID，它可以由写入日志项的应用程序使用。可以从应用程序中传递的参数在消息文本中用 % 语法定义。



消息文件的语法可参见 MSDN 文档中的“消息文本文件”。



```
; // EventLogDemoMessages.mc
; // *****
; // - Event categories -
; // Categories must be numbered consecutively starting at 1.
; // *****

MessageId=0x1
Severity=Success
SymbolicName=INSTALL_CATEGORY
Language=English
Installation

MessageId=0x2
Severity=Success
SymbolicName=DATA_CATEGORY
```

```

Language=English
Database Query

```

```

MessageId=0x3
Severity=Success
SymbolicName=UPDATE_CATEGORY
Language=English
Data Update

```

```

MessageId=0x4
Severity=Success
SymbolicName=NETWORK_CATEGORY
Language=English
Network Communication

```

```

; // - Event messages -
; // *****

```

```

MessageId = 1000
Severity = Success
Facility = Application
SymbolicName = MSG_CONNECT_1000
Language=English
Connection successful.

```

```

MessageId = 1001
Severity = Error
Facility = Application
SymbolicName = MSG_CONNECT_FAILED_1001
Language=English
Could not connect to server %1.

```

```

MessageId = 1002
Severity = Error
Facility = Application
SymbolicName = MSG_DB_UPDATE_1002
Language=English
Database update failed.

```

```

MessageId = 1003
Severity = Success
Facility = Application
SymbolicName = APP_UPDATE
Language=English
Application %5002 updated.

```

```

; // - Event log display name -
; // *****

```

```

MessageId = 5001
Severity = Success

```

```

Facility = Application
SymbolicName = EVENT_LOG_DISPLAY_NAME_MSGID
Language=English
Professional C# Sample Event Log
.

; // - Event message parameters -
; // Language independent insertion strings
; // *****

MessageId = 5002
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_SERVICE_NAME_MSGID
Language=English
EventLogDemo.EXE

```

代码段 EventLogDemo/EventLogDemoMessages.mc

使用消息编译器 **mc.exe**，创建一个二进制的消息文件。**mc -s EventLogDemoMessages.mc** 命令会把包含消息的源文件编译为一个扩展名为 **.bin** 的消息文件和 **Messages.rc** 文件，后者包含对二进制消息文件的引用。

```
mc -s EventLogMessages.mc
```

接着，需要使用资源编译器 **rc.exe**。命令 **rc EventLogDemoMessages.rc** 会创建资源文件 **EventLogDemoMessages.Res**：

```
rc EventLogMessages.rc
```

使用链接器，可以把二进制消息文件 **EventLogDemoMessages.RES** 绑定到一个本地 DLL 上。

```
link /DLL /SUBSYSTEM:WINDOWS /NOENTRY /MACHINE:x86 EventLogDemoMessages.Res
```

现在，就可以用下面的代码注册一个事件源，来定义资源文件了。首先检查事件源 **EventLogDemoApp** 是否存在。如果它不存在，就必须创建事件日志。接着验证资源文件是否可用。MSDN 文档中的一些示例说明了如何把消息文件写入 **<windows>\system32** 目录中，但这里不应这么做。把消息 DLL 复制到与程序相关的目录中，该目录可以使用 **SpécialFolder** 枚举的值 **ProgramFiles** 指定。如果需要在多个应用程序中共享消息文件，就可以把它放在 **Enviornment.SpecialFolder.CommonProgramFiles** 中。

如果该文件存在，就实例化 **EventSourceCreationData** 类型的一个新对象。在构造函数中，定义了源的名称和日志的名称。使用 **CatagoryResourceFile**、**MessageResourceFile** 和 **ParameterResourceFile** 属性定义资源文件的引用。在创建了事件源后，就可以使用事件源在注册表中找到资源文件的信息。**CreateEventSource()** 方法注册新的事件源和日志文件。最后，**EventLog** 类的 **RegisterDisplayName()** 方法指定日志显示在事件查看器中的名称。从消息文件中提取 ID 5001。



如果要删除以前创建的事件源，就可以使用 **EventLog.DeleteEventSource(sourceName)**。
要删除日志，可以调用 **EventLog.Delete(logName)**。



可从
wrox.com
下载源代码

```
string logName = "ProcSharpLog";
string sourceName = "EventLogDemoApp";
string resourceFile = Environment.GetFolderPath(
    Environment.SpecialFolder.ProgramFiles) +
    @"\procsharp\EventLogDemoMessages.dll";

if (!EventLog.SourceExists(sourceName))
{
    if (!File.Exists(resourceFile))
    {
        Console.WriteLine("Message resource file does not exist");
        return;
    }

    var eventSource = new EventSourceCreationData(sourceName, logName);

    eventSource.CategoryResourceFile = resourceFile;
    eventSource.CategoryCount = 4;
    eventSource.MessageResourceFile = resourceFile;
    eventSource.ParameterResourceFile = resourceFile;
    EventLog.CreateEventSource(eventSource);
}
else
{
    logName = EventLog.LogNameFromSourceName(sourceName, ".");
}

var evLog = new EventLog(logName, ".", sourceName);
evLog.RegisterDisplayName(resourceFile, 5001);
```

代码段 EventLogDemo/Program.cs

现在, 可以使用 `WriteEvent()` 方法而不是 `WriteEntry()` 写入事件日志项。 `WriteEvent()` 方法需要把一个 `EventInstance` 类型的对象作为参数。使用 `EventInstance` 可以指定消息 ID、类别和 `EventLogEntryType` 类型的严重程度。除 `EventInstance` 参数之外, `WriteEvent()` 方法的参数还允许接收消息的参数, 这些消息包含字节数组格式的参数和二进制数据。

```
using (var log = new EventLog(logName, ".", sourceName))
{
    var info1 = new EventInstance(1000, 4,
        EventLogEntryType.Information);
    log.WriteEvent(info1);
    var info2 = new EventInstance(1001, 4,
        EventLogEntryType.Error);
    log.WriteEvent(info2, "avalon");

    var info3 = new EventInstance(1002, 3,
        EventLogEntryType.Error);
    byte[] additionalInfo = { 1, 2, 3 };
    log.WriteEvent(info3, additionalInfo);
}
```



消息标识符可以定义带常量值的类, 该值在应用程序中为标识符提供更有意义的名称。

使用事件查看器可以读取事件日志项。

19.4 性能监控

性能监控可以用于获取关于应用程序的正常行为的一般信息。性能监控是一个强大的工具，有助于理解系统的工作负载，观察变化和趋势，尤其是运行在服务器上的应用程序。

Microsoft Windows 有许多性能对象，如 System、Memory、Objects、Process、Processor、Thread、Cache 等。这些对象有许多要监控的计数。例如，使用 Process 对象可以监控所有进程实例或特定进程实例的用户时间、句柄数、页面错误、线程数等。一些应用程序，如 SQL Server，还添加了与应用程序相关的对象。

19.4.1 性能监控类

System.Diagnostics 名称空间为性能监控提供了如下类：

- PerformanceCounter 类可以用于监控计数和写入计数。还可以使用这个类创建新的性能类别。
- PerformanceCounterCategory 类可以查看所有已有的类别，以及创建新类别。可以以编程方式获得一个类别中的所有计数器。
- PerformanceCounterInstaller 类用于安装性能计数器，它的用法类似于前面讨论的 EventLogInstaller 类。

19.4.2 性能计数器生成器

示例应用程序 PerformanceCounterDemo 是一个简单的 Windows 应用程序，它只有两个按钮，用于说明如何编写性能计数。使用一个按钮的处理程序注册性能计数器的类别，使用另一个按钮的处理程序写入性能计数器的值。采用同样的方式，可以在 Windows 服务(参见第 23 章)、网络应用程序(参见第 24 章)和任何其他要接收实时计数的应用程序中添加性能计数器。

使用 Visual Studio，可以创建新的性能计数器类别，具体方法是在 Server Explorer 中选择性能计数器，并在弹出的上下文菜单中选择菜单项 Create New Category 命令。这会启动 Performance Counter Builder(性能计数器生成器)，如图 19-7 所示。



要使用 Visual Studio 创建性能计数器类别，Visual Studio 必须以更高的模式启动。

将性能计数器类别的名称设置为 Wrox Performance Counters。表 19-4 列出了 quote 服务的所有性能计数器。

表 19-4

名 称	说 明	类 型
# of button clicks	按钮单击的总次数#	NumberOfItems32
# of button clicks/sec	一秒内按钮单击次数#	RateOfCountsPerSecond32
# of mouse move events	鼠标移动事件的总数#	NumberOfItems32
# of mouse move events/sec	一秒内鼠标移动事件的总数#	RateOfCountsPerSecond32

Performance Counter Builder 将配置写入性能数据库中。这也可以用 System.Diagnostics 名称空间中的 PerformanceCounterCategory 类的 Create() 方法动态完成。使用 Visual Studio 很容易在以后添加其他系统的安装程序。

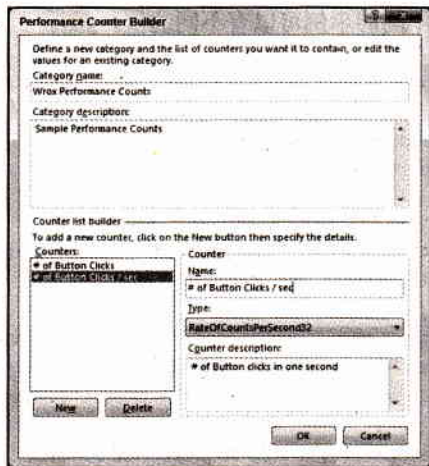


图 19-7

下面的代码段说明了如何以编程方式添加性能类别。使用 Visual Studio 中的工具，只能创建全局的性能类别，该类别不能为运行应用程序的不同进程指定不同的值。以编程方式创建性能类别，可以监控不同应用程序的性能计数，这里就采用这种方式。

首先，给类别名定义一个 const，再定义一个包含性能计数类别名的 SortedList<TKey, TValue>。



可从
wrox.com
下载源代码

```
private const string performanceCounterCategoryName = "Wrox Performance Counters";
private SortedList<string, Tuple<string, string>> perfCountNames;
```

代码段 PerformanceCounterDemo/MainWindow.xaml.cs

perfCountNames 变量的列表在 InitializePerformanceCountNames() 方法中填充。有序列表的值定义为 Tuple<string, string>，以确定性能计数器的名称和描述。



可从
wrox.com
下载源代码

```
private void InitializePerformanceCountNames()
{
    perfCountNames = new SortedList<string, Tuple<string, string>>();
    perfCountNames.Add("clickCount",
        Tuple.Create("# of button Clicks", "Total # of button clicks"));
    perfCountNames.Add("clickSec",
        Tuple.Create("# of button clicks/sec",
            "# of mouse button clicks in one second"));
    perfCountNames.Add("mouseCount",
        Tuple.Create("# of mouse move events",
            "Total # of mouse move events"));
    perfCountNames.Add("mouseSec",
        Tuple.Create("# of mouse move events/sec",
            "# of mouse move events in one second"));
}
```

代码段 PerformanceCounterDemo/MainWindow.xaml.cs

接着在 `OnRegisterCounts()` 方法中创建性能计数器类别。验证该类别不存在后，就创建一个 `CounterCreationData` 数组，用性能计数的类型和名称填充该数组。接着，使用 `PerformanceCounterCategory.Create()` 方法创建新类别。`PerformanceCounterCategoryType.MultiInstance` 指定这些计数不是全局的，但不同的示例可以有不同的值。



可从
wrox.com
下载源代码

```
private void OnRegisterCounts(object sender, RoutedEventArgs e)
{
    if (!PerformanceCounterCategory.Exists(
        perfomanceCounterCategoryName))
    {
        var counterCreationData = new CounterCreationData[4];
        counterCreationData[0] = new CounterCreationData
        {
            CounterName = perfCountNames["clickCount"].Item1,
            CounterType = PerformanceCounterType.NumberOfItems32,
            CounterHelp = perfCountNames["clickCount"].Item2
        };
        counterCreationData[1] = new CounterCreationData
        {
            CounterName = perfCountNames["clickSec"].Item1,
            CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
            CounterHelp = perfCountNames["clickSec"].Item2,
        };
        counterCreationData[2] = new CounterCreationData
        {
            CounterName = perfCountNames["mouseCount"].Item1,
            CounterType = PerformanceCounterType.NumberOfItems32,
            CounterHelp = perfCountNames["mouseCount"].Item2,
        };
        counterCreationData[3] = new CounterCreationData
        {
            CounterName = perfCountNames["mouseSec"].Item1,
            CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
            CounterHelp = perfCountNames["mouseSec"].Item2,
        };
        var counters =
            new CounterCreationDataCollection(counterCreationData);

        var category = PerformanceCounterCategory.Create(
            perfomanceCounterCategoryName,
            "Sample Counters for Professional C#",
            PerformanceCounterCategoryType.MultiInstance,
            counters);

        MessageBox.Show(String.Format(
            "category {0} successfully created",
            category.CategoryName));
    }
}
```

代码段 PerformanceCounterDemo/MainWindow.xaml.cs

19.4.3 添加 PerformanceCounter 组件

通过 Windows Forms 或 Windows Service 应用程序, 可以从工具箱中添加 PerformanceCounter 组件, 或者从 Server Explorer 中把该组件拖放到设计界面上。

通过 WPF 应用程序, 就不能实现上述操作。但定义性能计数器所需的手工劳动并不多, 因为它使用了 InitializePerformanceCounts() 方法。这里, 所有性能计数的 CategoryName 都在 const 字符串 performanceCounterCategoryName 中设置, CounterName 在有序列表中设置。由于应用程序要写入性能计数, 因此 ReadOnly 属性必须设置为 false。如果所编写的应用程序出于显示目的仅读取性能计数, 就可以使用 ReadOnly 属性的默认值 true。PerformanceCounter 对象的 InstanceName 属性设置为应用程序名。如果计数器配置为全局计数, 就不能设置 InstanceName 属性。



可从
wrox.com
下载源代码

```
private PerformanceCounter performanceCounterButtonClicks;
private PerformanceCounter performanceCounterButtonClicksPerSec;
private PerformanceCounter performanceCounterMouseMoveEvents;
private PerformanceCounter performanceCounterMouseMoveEventsPerSec;

private void InitializePerformanceCounts()
{
    performanceCounterButtonClicks = new PerformanceCounter
    {
        CategoryName = performanceCounterCategoryName,
        CounterName = perfCountNames["clickCount"].Item1,
        ReadOnly = false,
        MachineName = ".",
        InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
        InstanceName = this.instanceName
    };
    performanceCounterButtonClicksPerSec = new PerformanceCounter
    {
        CategoryName = performanceCounterCategoryName,
        CounterName = perfCountNames["clickSec"].Item1,
        ReadOnly = false,
        MachineName = ".",
        InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
        InstanceName = this.instanceName
    };
    performanceCounterMouseMoveEvents = new PerformanceCounter
    {
        CategoryName = performanceCounterCategoryName,
        CounterName = perfCountNames["mouseCount"].Item1,
        ReadOnly = false,
        MachineName = ".",
        InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
        InstanceName = this.instanceName
    };
    performanceCounterMouseMoveEventsPerSec = new PerformanceCounter
    {
        CategoryName = performanceCounterCategoryName,
        CounterName = perfCountNames["mouseSec"].Item1,
        ReadOnly = false,
        MachineName = ".",
        InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
        InstanceName = this.instanceName
    };
}
```

为了计算性能值，需要添加 `clickCountPerSec` 和 `mouseMoveCountPerSec` 字段：

```
public partial class MainWindow : Window
{
    // Performance monitoring counter values
    private int clickCountPerSec = 0;
    private int mouseMoveCountPerSec = 0;
```

给按钮的 `Click` 事件添加事件处理程序，给按钮的 `MouseMove` 事件添加事件处理程序，在处理程序中添加如下代码：

```
private void OnButtonClick(object sender, RoutedEventArgs e)
{
    this.performanceCounterButtonClicks.Increment();
    this.clickCountPerSec++;
}

private void OnMouseMove(object sender, MouseEventArgs e)
{
    this.performanceCounterMouseMoveEvents.Increment();
    this.mouseMoveCountPerSec++;
}
```

`PerformanceCounter` 对象的 `Increment()` 方法给计数器递增 1。如果需要给计数器添加其他信息，例如，添加一个字节的收发计数，就可以使用 `IncrementBy()` 方法。对于显示以秒为单位的值的性能计数，只需递增两个变量 `clickCountPerSec` 和 `mouseMovePerSec`。

为了显示每秒更新后的值，可给 `MainWindow` 的成员添加一个 `DispatcherTimer` 类。

```
private DispatcherTimer timer;
```

这个计时器在构造函数中配置和启动。`DispatcherTimer` 类是 `System.Windows.Threading` 名称空间中的一个计时器。对于非 WPF 应用程序，可以使用第 20 章要讨论的其他计时器。计时器调用的代码在一个匿名方法中指定：

```
public MainWindow()
{
    InitializeComponent();
    InitializePerformanceCountNames();
    InitializePerformanceCounts();
    if (PerformanceCounterCategory.Exists(performanceCounterCategoryName))
    {
        buttonCount.IsEnabled = true;
        timer = new DispatcherTimer(TimeSpan.FromSeconds(1),
            DispatcherPriority.Background,
            delegate
            {
                this.performanceCounterButtonClicksPerSec.RawValue =
                    this.clickCountPerSec;
                this.clickCountPerSec = 0;
                this.performanceCounterMouseMoveEventsPerSec.RawValue =
                    this.mouseMoveCountPerSec;
```

```

        this.mouseMoveCountPerSec = 0;
    },
    Dispatcher.CurrentDispatcher);
timer.Start();
}
}

```

19.4.4 perfmon.exe

现在就可以监控应用程序了。从“管理工具”中可以启动“性能监控器”，在“性能监控器”中，单击工具栏上的“+”按钮，可以添加性能计数。Wrox PerformanceCounts 服务显示为一个性能对象。所有已配置的计数器都显示在计数器列表中，如图 19-8 所示。

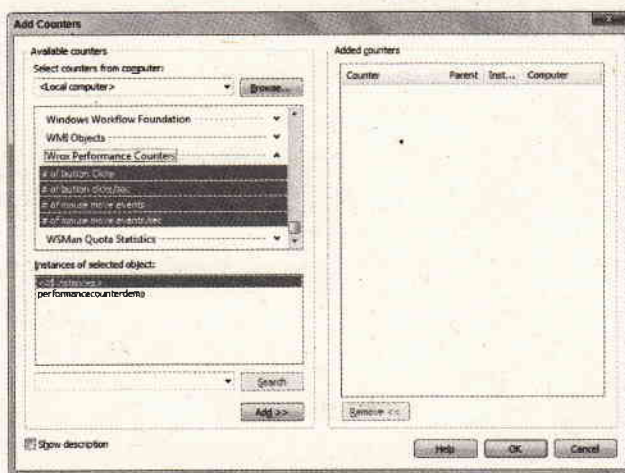


图 19-8

在性能监控器中添加了计数器后，就可以查看服务随时间变化的实际值，如图 19-9 所示。使用这个性能工具，还可以创建日志文件，在以后分析性能。

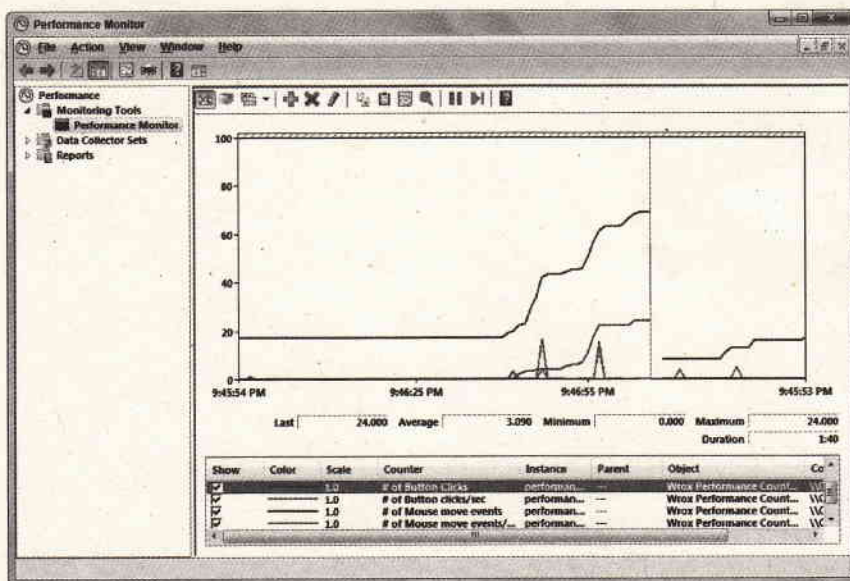


图 19-9

19.5 小结

本章介绍了跟踪和日志功能，它们有助于找出应用程序中的中间问题。应尽早规划，把这些功能内置于应用程序中。这可以避免以后的许多故障排除问题。

使用跟踪功能，可以把调试消息写入应用程序，也可以用于最终发布的产品。如果出了问题，就可以修改配置值，从而打开跟踪功能，并找出问题。

事件日志为系统管理员提供信息，帮助找出应用程序的某些严重问题。性能监控有助于分析应用程序的负载，提前规划将来可能需要的资源。

下一章学习如何编写多线程的应用程序。

第 20 章

线程、任务和同步

本章内容:

- 线程概述
- 使用委托的轻型线程
- 线程类和线程池
- 任务
- Parallel 类
- 取消架构
- 线程问题
- 同步技术
- 计时器
- 基于事件的异步模式

使用线程有几个原因。假设从应用程序中进行网络调用需要一定的时间。用户不希望分割用户界面，并且让用户一直等待直到从服务器返回一个响应为止。用户可以同时执行其他一些操作，或者甚至取消发送给服务器的请求。这些都可以使用线程来实现。

对于所有需要等待的操作，例如，因为文件、数据库或网络访问都需要一定的时间，此时就可以启动一个新线程，同时完成其他任务。即使是处理密集型的任务，线程也是有帮助的。一个进程的多个线程可以同时运行在不同的 CPU 上，或多核 CPU 的不同内核上。

还必须注意运行多个线程时的一些问题。它们可以同时运行，但如果线程访问相同的数据，就很容易出问题。必须实现同步机制。

本章介绍用多个线程编写应用程序所需的基础知识。

20.1 概述

线程是程序中独立的指令流。使用 C# 编写任何程序时，都有一个入口点：Main() 方法。程序从 Main() 方法的第一条语句开始执行，直到这个方法返回为止。

这种程序结构非常适合于其中有一个可识别的任务序列的程序，但程序常常需要同时完成多个任务。线程对客户端和服务端应用程序都非常重要。在 Visual Studio 编辑器中输入 C# 代码时，系统会分析代码，用下划线标出遗漏的分号或其他语法错误，这用一个后台线程完成。Microsoft Word

的拼写检查器也会做相同的事。一个线程等待用户输入，而另一个线程进行后台搜索。第3个线程将写入的数据存储在临时文件中，第4个线程从 Internet 上下载其他数据。

运行在服务器上的应用程序中，等待客户请求的线程，称为侦听器线程。只要接收到请求，就把它传递给另一个工作线程，之后继续与客户通信。侦听器线程会立即返回，接收下一个客户发送的下一个请求。

进程包含资源，如 Window 句柄、文件系统句柄或其他内核对象。每个进程都分配了虚拟内存。一个进程至少包含一个线程。操作系统会调度线程。线程有一个优先级、实际上正在处理的程序的位置计数器、一个存储其局部变量的栈。每个线程都有自己的栈，但程序代码的内存和堆由一个进程的所有线程共享。这使一个进程的所有线程之间的通信非常快——该进程的所有线程都寻址相同的虚拟内存。但是，这也使处理比较困难，因为多个线程可以修改同一个内存位置。

进程管理的资源包括虚拟内存和 Window 句柄，其中至少包含一个线程。线程是运行程序所必需的。

20.2 异步委托

创建线程的一种简单方式是定义一个委托，并异步调用它。第8章提到，委托是方法的类型安全的引用。Delegate 类还支持异步地调用方法。在后台，Delegate 类会创建一个执行任务的线程。



委托使用线程池来完成异步任务。线程池详见本章后面的内容。

为了说明委托的异步特性，从一个需要一定的时间才能执行完毕的方法开始。TakesAWhile()方法至少需要经过第2个变量传递的毫秒数才能执行完，因为它调用 Thread.Sleep()方法：



可从
wrox.com
下载源代码

```
static int TakesAWhile(int data, int ms)
{
    Console.WriteLine("TakesAWhile started");
    Thread.Sleep(ms);
    Console.WriteLine("TakesAWhile completed");
    return ++data;
}
```

代码段 AsyncDelegate/Program.cs

要从委托中调用这个方法，必须定义一个有相同参数和返回类型的委托，如下面的 TakesAWhileDelegate()方法所示：

```
public delegate int TakesAWhileDelegate(int data, int ms);
```

现在可以使用不同的技术异步地调用委托，并返回结果。

20.2.1 投票

一种技术是投票，并检查委托是否完成了它的任务。所创建的 Delegate 类提供了 BeginInvoke()

方法,在该方法中,可以传递用委托类型定义的输入参数。`BeginInvoke()`方法总是有 `AsyncCallback` 和 `object` 类型的两个额外参数(稍后讨论)。现在重要的是 `BeginInvoke()`方法的返回类型: `IAsyncResult`。通过 `IAsyncResult`,可以获得该委托的相关信息,并验证该委托是否完成了任务,这是 `IsCompleted` 属性的功劳。只要委托没有完成其任务,程序的主线程就继续执行 `while` 循环。

```
static void Main()
{
    // synchronous method call
    // TakesAWhile(1, 3000);

    // asynchronous by using a delegate
    TakesAWhileDelegate dl = TakesAWhile;

    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (!ar.IsCompleted)
    {
        // doing something else in the main thread
        Console.WriteLine(".");
        Thread.Sleep(50);
    }
    int result = dl.EndInvoke(ar);
    Console.WriteLine("result: {0}", result);
}
```

代码段 Async Delegate/Program.cs

运行应用程序时,可以看到主线程和委托线程同时运行,在委托线程执行完毕后,主线程就停止循环。

```
.TakesAWhile started
..TakesAWhile completed
result: 2
```

除了检查委托是否完成之外,还可以在完成了由主线程执行的工作后,调用委托类型的 `EndInvoke()`方法。`EndInvoke()`方法会一直等待,直到委托完成其任务为止。



如果在委托结束之前不等待委托完成其任务就结束主线程,委托线程就会停止。

20.2.2 等待句柄

等待异步委托的结果的另一种方式是使用与 `IAsyncResult` 相关联的等待句柄。使用 `AsyncWaitHandle` 属性可以访问等待句柄。这个属性返回一个 `WaitHandle` 类型的对象,它可以等待委托线程完成其任务。`WaitOne()`方法将一个超时时间作为可选的第一个参数,在其中可以定义要等待的最长时间。这里设置为 50 毫秒。如果发生超时, `WaitOne()`方法就返回 `false`, `while` 循环会继续执行。如果等待操作成功,就用一个中断退出 `while` 循环,用委托的 `EndInvoke()`方法接收结果。从 UI 的立场来看,结果与前面的示例类似,只是用另一种方式执行等待操作。

```
static void Main()
{
```

```

TakesAWhileDelegate dl = TakesAWhile;

IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
while (true)
{
    Console.WriteLine(".");
    if (ar.AsyncWaitHandle.WaitOne(50, false))
    {
        Console.WriteLine("Can get the result now");
        break;
    }
}

int result = dl.EndInvoke(ar);
Console.WriteLine("result: {0}", result);
}

```



等待句柄的内容详见 20.9 节。

20.2.3 异步回调

等待委托的结果的第 3 种方式是使用异步回调。在 `BeginInvoke()` 方法的第 3 个参数中，可以传递一个满足 `AsyncCallback` 委托的需求的方法。`AsyncCallback` 委托定义了一个 `IAsyncResult` 类型的参数，其返回类型是 `void`。这里，把 `TakesAWhileCompleted()` 方法的地址赋予第 3 个参数，它满足 `AsyncCallback` 委托的需求。对于最后一个参数，可以传递任意对象，以便从回调方法中访问它。传递委托实例很有用，这样回调方法就可以使用它获得异步方法的结果。

现在，只要 `TakesAWhileDelegate` 委托完成其任务，就调用 `TakesAWhileCompleted()` 方法。不需要在主线程中等待结果。但是在委托线程的任务未完成之前，不能停止主线程，除非主线程结束时停止的委托线程没有问题。



可从
wrox.com
下载源代码

```

static void Main()
{
    TakesAWhileDelegate dl = TakesAWhile;

    dl.BeginInvoke(1, 3000, TakesAWhileCompleted, dl);
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(".");
        Thread.Sleep(50);
    }
}

```

代码段 Async Delegate/Program.cs

`TakesAWhileCompleted()` 方法用 `AsyncCallback` 委托指定的参数和返回类型来定义。用 `BeginInvoke()` 方法传递的最后一个参数可以使用 `ar.AsyncState` 读取。在 `TakesAWhileDelegate` 委托中，可以调用 `EndInvoke()` 方法来获得结果。

```

static void TakesAWhileCompleted(IAsyncResult ar)
{

```

```

if (ar == null) throw new ArgumentNullException("ar");

TakesAWhileDelegate d1 = ar.AsyncState as TakesAWhileDelegate;
Trace.Assert(d1 != null, "Invalid object type");

int result = d1.EndInvoke(ar);
Console.WriteLine("result: {0}", result);
}

```



使用回调方法，必须注意这个方法从委托线程中调用，而不是从主线程中调用。

除了定义一个单独的方法，并给它传递 `BeginInvoke()` 方法之外，`Lambda` 表达式也非常适合这种情况。参数 `ar` 是 `IAAsyncResult` 类型。在实现代码中，不需要把一个值赋予 `BeginInvoke()` 方法的最后一个参数，因为 `Lambda` 表达式可以直接访问该作用域外部的变量 `d1`。但是，`Lambda` 表达式的实现代码仍是从委托线程中调用，以这种方式定义方法时，这不是很明显。

```

static void Main()
{
    TakesAWhileDelegate d1 = TakesAWhile;
    d1.BeginInvoke(1, 3000,
        ar =>
        {
            int result = d1.EndInvoke(ar);
            Console.WriteLine("result: {0}", result);
        },
        null);
    for (int i = 0; i < 100; i++)
    {
        Console.Write(".");
        Thread.Sleep(50);
    }
}

```

编程模型和所有这些包含异步委托的选项——投票、等待句柄和异步回调——不仅能用于委托。相同的编程模型(即异步模式)在 .NET Framework 的各个地方都能见到。例如，可以用 `HttpRequest` 类的 `BeginGetResponse()` 方法异步发送 HTTP Web 请求，使用 `SqlCommand` 类的 `BeginExecuteReader()` 方法给数据库发送异步请求。这些参数类似于委托的 `BeginInvoke()` 方法的参数，也可以使用相同的方式获得结果。



`HttpRequest` 参见第 24 章，`SqlCommand` 参见第 30 章。

除了使用委托创建线程之外，还可以使用 `Thread` 类创建线程，如下一节所述。

20.3 Thread 类

使用 Thread 类可以创建和控制线程。下面的代码是创建和启动一个新线程的简单例子。Thread 类的构造函数重载为接受 ThreadStart 和 ParameterizedThreadStart 类型的委托参数。ThreadStart 委托定义了一个返回类型为 void 的无参数方法。在创建了 Thread 对象后,就可以用 Start()方法启动线程:



可从
wrox.com
下载源代码

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            var t1 = new Thread(ThreadMain);
            t1.Start();
            Console.WriteLine("This is the main thread.");
        }

        static void ThreadMain()
        {
            Console.WriteLine("Running in a thread.");
        }
    }
}
```

代码段 ThreadSamples/Program.cs

运行这个程序时,得到两个线程的输出:

```
This is the main thread.
Running in a thread.
```

不能保证哪个结果先输出。线程由操作系统调度,每次哪个线程在前面可以不同。

前面探讨了 Lambda 表达式如何与异步委托一起使用。Lambda 表达式还可以与 Thread 类一起使用,将线程方法的实现代码传送给 Thread 构造函数的实参:

```
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            var t1 = new Thread(() => Console.WriteLine("running in a thread, id: {0}",
                Thread.CurrentThread.ManagedThreadId));
            t1.Start();
            Console.WriteLine("This is the main thread, id: {0}",
                Thread.CurrentThread.ManagedThreadId);
        }
    }
}
```

在应用程序的输出中，现在还可以看到线程名和 ID：

```
This is the main thread, id: 1
Running in a thread, id: 3.
```

20.3.1 给线程传递数据

给线程传递一些数据可以采用两种方式。一种方式是使用带 `ParameterizedThreadStart` 委托参数的 `Thread` 构造函数，另一种方式是创建一个自定义类，把线程的方法定义为实例方法，这样就可以初始化实例的数据，之后启动线程。

要给线程传递数据，需要某个存储数据的类或结构。这里定义了包含字符串的 `Data` 结构，但可以传递任意对象。



可从
wrox.com
下载源代码

```
public struct Data
{
    public string Message;
}
```

代码段 ThreadSamples/Program.cs

如果使用了 `ParameterizedThreadStart` 委托，线程的入口点必须有一个 `object` 类型的参数，且返回类型为 `void`。对象可以强制转换为任意数据类型，这里是把消息写入控制台。

```
static void ThreadMainWithParameters(object o)
{
    Data d = (Data)o;
    Console.WriteLine("Running in a thread, received {0}", d.Message);
}
```

通过 `Thread` 类的构造函数，可以将新的入口点赋予 `ThreadMainWithParameters`，传递变量 `d`，以此调用 `Start()` 方法。

```
static void Main()
{
    var d = new Data { Message = "Info" };
    var t2 = new Thread(ThreadMainWithParameters);
    t2.Start(d);
}
```

给新线程传递数据的另一种方式是定义一个类(参见 `MyThread` 类)，在其中定义需要的字段，将线程的主方法定义为类的一个实例方法：

```
public class MyThread
{
    private string data;

    public MyThread(string data)
    {
        this.data = data;
    }

    public void ThreadMain()
    {
        Console.WriteLine("Running in a thread, data: {0}", data);
    }
}
```

```

    }
}

```

这样, 就可以创建 `MyThread` 的一个对象, 给 `Thread` 类的构造函数传递对象和 `ThreadMain()` 方法。线程可以访问数据。

```

var obj = new MyThread("info");
var t3 = new Thread(obj.ThreadMain);
t3.Start();

```

20.3.2 后台线程

只要有一个前台线程在运行, 应用程序的进程就在运行。如果多个前台线程在运行, 而 `Main()` 方法结束了, 应用程序的进程就仍然是激活的, 直到所有前台线程完成其任务为止。

在默认情况下, 用 `Thread` 类创建的线程是前台线程。线程池中的线程总是后台线程。

在用 `Thread` 类创建线程时, 可以设置 `IsBackground` 属性, 以确定该线程是前台线程还是后台线程。`Main()` 方法将线程 `t1` 的 `IsBackground` 属性设置为 `false` (默认值)。在启动新线程后, 主线程就把结束消息写入控制台中。新线程会写入启动消息和结束消息, 在这个过程中它要睡眠 3 秒。在新线程会完成其工作前, 这 3 秒钟有利于主线程结束。



可从
wrox.com
下载源代码

```

class Program
{
    static void Main()
    {
        var t1 = new Thread(ThreadMain)
        { Name = "MyNewThread", IsBackground = false };
        t1.Start();
        Console.WriteLine("Main thread ending now.");
    }

    static void ThreadMain()
    {
        Console.WriteLine("Thread {0} started", Thread.CurrentThread.Name);
        Thread.Sleep(3000);
        Console.WriteLine("Thread {0} completed", Thread.CurrentThread.Name);
    }
}

```

代码段 ThreadSamples/Program.cs

尽管主线程会提前完成其工作, 但在启动应用程序时, 会看到写入控制台的完成消息。原因是新线程也是一个前台线程。

```

Main thread ending now.
Thread MyNewThread1 started
Thread MyNewThread1 completed

```

如果将用来启动新线程的 `IsBackground` 属性改为 `true`, 显示在控制台上的结果就会不同。在控制台上, 可以看到相同的结果——新线程的启动消息, 但没有结束消息。如果线程没有正常结束, 就也有可能看不到启动消息。

```

Main thread ending now.

```

Thread MyNewThread1 started

后台线程非常适合于完成后台任务。例如，如果关闭 Word 应用程序，拼写检查器继续运行其进程就没有意义了。在关闭应用程序时，拼写检查器线程就可以关闭。但是，组织 Outlook 消息库的线程应一直是激活的，直到关闭 Outlook，它才结束。

20.3.3 线程的优先级

前面提到，线程由操作系统调度。给线程指定优先级，就可以影响调度顺序。

在改变优先级之前，必须理解线程调度器。操作系统根据优先级来调度线程。调度优先级最高的线程以在 CPU 上运行。线程如果在等待资源，它就会停止运行，并释放 CPU。线程必须等待几个原因，例如，响应睡眠指令、等待磁盘 I/O 的完成，等待网络包的到达等。如果线程不是主动释放 CPU，线程调度器就会抢占该线程。如果线程有一个时间量，它就可以继续使用 CPU。如果优先级相同的多个线程等待使用 CPU，线程调度器就会使用一个循环调度规则，将 CPU 逐个交给线程使用。如果线程被其他线程抢占，它就会排在队列的最后。

只有优先级相同的多个线程在运行，才用得上时间量和循环规则。优先级是动态的。如果线程是 CPU 密集型的(一直需要 CPU，且不等待资源)，其优先级就低于用该线程定义的基本优先级。如果线程在等待资源，随着优先级的提高它的优先级就会增加。由于优先级的提高，线程才有可能在下次等待结束时获得 CPU。

在 Thread 类中，可以设置 Priority 属性，以影响线程的基本优先级。Priority 属性需要 ThreadPriority 枚举定义的一个值。定义的级别有 Highest、AboveNormal、BelowNormal 和 Lowest。



在给线程指定较高的优先级时要小心，因为这可能降低其他线程的运行概率。根据需要，可以短暂地改变优先级。

20.3.4 控制线程

调用 Thread 对象的 Start() 方法，可以创建线程。但是，在调用 Start() 方法后，新线程仍不是处于 Running 状态，而是处于 Unstarted 状态。只要操作系统的线程调度器选择了要运行的线程，线程就会改为 Running 状态。读取 Thread.ThreadState 属性，就可以获得线程的当前状态。

使用 Thread.Sleep() 方法，会使线程处于 WaitSleepJoin 状态，在经历 Sleep() 方法定义的时间段后，线程就会等待再次被唤醒。

要停止另一个线程，可以调用 Thread.Abort() 方法。调用这个方法时，会在接到终止命令的线程中抛出一个 ThreadAbortException 类型的异常。用一个处理程序捕获这个异常，线程可以在结束前完成一些清理工作。线程还可以在接收到调用 Thread.ResetAbort() 方法的结果 ThreadAbortException 异常后继续运行。如果线程没有重置终止，接收到终止请求的线程的状态就从 AbortRequested 改为 Aborted。

如果需要等待线程的结束，就可以调用 Thread.Join() 方法。Thread.Join() 方法会停止当前线程，并把它设置为 WaitSleepJoin 状态，直到加入的线程完成为止。

20.4 线程池

创建线程需要时间。如果有不同的小任务要完成，就可以事先创建许多线程，在应完成这些任务时发出请求。这个线程数最好在需要更多的线程时增加，在需要释放资源时减少。

不需要自己创建这样一个列表。该列表由 `ThreadPool` 类托管。这个类会在需要时增减池中线程的线程数，直到最大的线程数。池中的最大线程数是可配置的。在双核 CPU 中，默认设置为 1023 个工作线程和 1000 个 I/O 线程。也可以指定在创建线程池时应立即启动的最小线程数，以及线程池中可用的最大线程数。如果有更多的作业要处理，线程池中线程的个数也到了极限，最新的作业就要排队，且必须等待线程完成其任务。

下面的示例应用程序首先要读取工作线程和 I/O 线程的最大线程数，把这些信息写入控制台中。接着在 for 循环中，调用 `ThreadPool.QueueUserWorkItem()` 方法，传递一个 `WaitCallback` 类型的委托，把 `JobForAThread()` 方法赋予线程池中的线程。线程池收到这个请求后，就会从池中选择一个线程，来调用该方法。如果线程池还没有运行，就会创建一个线程池，并启动第一个线程。如果线程池已经在运行，且有一个空闲线程来完成该任务，就把该作业传递给这个线程。



可从
wrox.com
下载源代码

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        static void Main()
        {
            int nWorkerThreads;
            int nCompletionPortThreads;
            ThreadPool.GetMaxThreads(out nWorkerThreads, out nCompletionPortThreads);
            Console.WriteLine("Max worker threads: {0}, " +
                             "I/O completion threads: {1}",
                             nWorkerThreads, nCompletionPortThreads);

            for (int i = 0; i < 5; i++)
            {
                ThreadPool.QueueUserWorkItem(JobForAThread);
            }
            Thread.Sleep(3000);
        }

        static void JobForAThread(object state)
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("loop {0}, running inside pooled thread {1}",
                                   i, Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(50);
            }
        }
    }
}
```

代码段 `ThreadPoolSamples/Program.cs`

运行应用程序时, 可以看到 50 个工作线程的当前设置。5 个作业只由两个线程池中的线程处理, 读者运行该程序的结果可能与此不同, 也可以改变作业的睡眠时间和要处理的作业数, 得到完全不同的结果。

```
Max worker threads: 1023, I/O completion threads: 1000
loop 0, running inside pooled thread 4
loop 0, running inside pooled thread 3
loop 1, running inside pooled thread 4
loop 1, running inside pooled thread 3
loop 2, running inside pooled thread 4
loop 2, running inside pooled thread 3
loop 0, running inside pooled thread 4
loop 0, running inside pooled thread 3
loop 1, running inside pooled thread 4
loop 1, running inside pooled thread 3
loop 2, running inside pooled thread 4
loop 2, running inside pooled thread 3
loop 0, running inside pooled thread 4
loop 1, running inside pooled thread 4
loop 2, running inside pooled thread 4
```

线程池使用起来很简单, 但它有一些限制:

- 线程池中的所有线程都是后台线程。如果进程的所有前台线程都结束了, 所有的后台线程就会停止。不能把入池的线程改为前台线程。
- 不能给入池的线程设置优先级或名称。
- 对于 COM 对象, 入池的所有线程都是多线程单元(multithreaded apartment, MTA)线程。许多 COM 对象都需要单线程单元(single-threaded apartment, STA)线程。
- 入池的线程只能用于时间较短的任务。如果线程要一直运行(如 Word 的拼写检查器线程), 就应使用 Thread 类创建一个线程。

20.5 任务

.NET 4 包含新名称空间 System.Threading.Tasks, 它包含的类抽象出了线程功能。在后台使用 ThreadPool。任务表示应完成的某个单元的工作。这个单元的工作可以在单独的线程中运行, 也可以以同步方式启动一个任务, 这需要等待主调线程。使用任务不仅可以获得一个抽象层, 还可以对底层线程进行很多控制。

在安排需要完成的工作时, 任务提供了非常大的灵活性。例如, 可以定义连续的工作——在一个任务完成后该执行什么工作。这可以区分任务成功与否。另外, 还可以在层次结构中安排任务。例如, 父任务可以创建新的子任务。这可以创建一种依赖关系, 这样, 取消父任务, 也会取消其子任务。

20.5.1 启动任务

要启动任务, 可以使用 TaskFactory 类或 Task 类的构造函数和 Start()方法。Task 类的构造函数

在创建任务上提供的灵活性较大。

在启动任务时,会创建 `Task` 类的一个实例,利用 `Action` 或 `Action<object>` 委托(不带参数或带一个 `object` 参数),可以指定应运行的代码。这类似于 `Thread` 类。下面定义了一个无参数的方法。在实现代码中,把任务的 ID 写入控制台中:



可从
wrox.com
下载源代码

```
static void TaskMethod()
{
    Console.WriteLine("running in a task");
    Console.WriteLine("Task id: {0}", Task.CurrentId);
}
```

代码段 TaskSamples/Program.cs

在上面的代码中,可以看到启动新任务的不同方式。第一种方式使用实例化的 `TaskFactory` 类,在其中把 `TaskMethod()` 方法传递给 `StartNew()` 方法,就会立即启动任务。第二种方式使用 `Task` 类的构造函数。实例化 `Task` 对象时,任务不会立即运行,而是指定 `Created` 状态。接着调用 `Task` 类的 `Start()` 方法,来启动任务。使用 `Task` 类时,除了调用 `Start()` 方法,还可以调用 `RunSynchronously()` 方法。这样,任务也会启动,但在调用者的当前线程中它正在运行,调用者需要一直等待到该任务结束。默认情况下,任务是异步运行的。

```
// using task factory
TaskFactory tf = new TaskFactory();
Task t1 = tf.StartNew(TaskMethod);

// using the task factory via a task
Task t2 = Task.Factory.StartNew(TaskMethod);

// using Task constructor
Task t3 = new Task(TaskMethod);
t3.Start();
```

使用 `Task` 类的构造函数和 `TaskFactory` 类的 `StartNew()` 方法时,都可以传递 `TaskCreationOptions` 枚举中的值。设置 `LongRunning` 选项,可以通知任务调度器,该任务需要较长时间执行,这样调度器更可能使用新线程。如果该任务应关联到父任务上,而父任务取消了,则该任务也应取消,此时应设置 `AttachToParent` 选项。`PerferFairness` 的值表示,调度器应提取出已在等待的第一个任务。如果一个任务在另一个任务内部创建,这就不是默认情况。如果任务使用子任务创建其他工作,子任务就优先于其他任务。它们不会排在线程池队列中的最后。如果这些任务应以公平的方式与所有其他任务一起处理,就设置该选项为 `PerferFairness`。

```
Task t4 = new Task(TaskMethod, TaskCreationOptions.PreferFairness);
t4.Start();
```

20.5.2 连续的任务

通过任务,可以指定在任务完成后,应开始运行另一个特定任务,例如,一个使用前一个任务的结果的新任务,如果前一个任务失败了,这个任务就应执行一些清理工作。

任务处理程序或者不带参数或者带一个对象参数,而连续处理程序有一个 `Task` 类型的参数,这里可以访问起始任务的相关信息:



可从
wrox.com
下载源代码

```
static void DoOnFirst()
{
    Console.WriteLine("doing some task {0}", Task.CurrentId);
    Thread.Sleep(3000);
}

static void DoOnSecond(Task t)
{
    Console.WriteLine("task {0} finished", t.Id);
    Console.WriteLine("this task id {0}", Task.CurrentId);
    Console.WriteLine("do some cleanup");
    Thread.Sleep(3000);
}
```

代码段 TaskSamples/Program.cs

连续任务通过在任务上调用 `ContinueWith()` 方法来定义。也可以使用 `TaskFactory` 类来定义。`t1.OnContinueWith(DoOnSecond)` 方法表示, 调用 `DoOnSecond()` 方法的新任务应在任务 `t1` 结束时立即启动。在一个任务结束时, 可以启动多个任务, 连续任务也可以有另一个连续任务, 如下面的例子所示:

```
Task t1 = new Task(DoOnFirst);
Task t2 = t1.ContinueWith(DoOnSecond);
Task t3 = t1.ContinueWith(DoOnSecond);
Task t4 = t2.ContinueWith(DoOnSecond);
```

无论前一个任务是如何结束的, 前面的连续任务总是在前一个任务结束时启动。使用 `TaskCreationOptions` 枚举中的值, 可以指定, 连续任务只有在起始任务成功(或失败)结束时启动。一些可能的值是 `OnlyOnFaulted`、`NotOnFaulted`、`OnlyOnCanceled`、`NotOnCanceled` 和 `OnlyOnRanToCompletion`。

```
Task t5 = t1.ContinueWith(DoOnError,
    TaskContinuationOptions.OnlyOnFaulted);
```

20.5.3 任务层次结构

利用任务连续性, 可以在一个任务结束后启动另一个任务。任务也可以构成一个层次结构。一个任务启动一个新任务时, 就启动了一个父子层次结构。

下面的代码段在父任务内部新建一个任务。创建子任务的代码与创建父任务的代码相同, 唯一的区别是这个任务从另一个任务内部创建。



可从
wrox.com
下载源代码

```
static void ParentAndChild()
{
    var parent = new Task(ParentTask);
    parent.Start();
    Thread.Sleep(2000);
    Console.WriteLine(parent.Status);
    Thread.Sleep(4000);
    Console.WriteLine(parent.Status);
}

static void ParentTask()
{
    Console.WriteLine("task id {0}", Task.CurrentId);
    var child = new Task(ChildTask);
```

```

        child.Start();
        Thread.Sleep(1000);
        Console.WriteLine("parent started child");
    }
    static void ChildTask()
    {
        Console.WriteLine("child");
        Thread.Sleep(5000);
        Console.WriteLine("child finished");
    }
}

```

代码段 TaskSamples/Program.cs

如果父任务在子任务之前结束，父任务的状态就显示为 `WaitingForChildrenToComplete`。只要子任务也结束时，父任务的状态就变成 `RanToCompletion`。当然，如果父任务用 `TaskCreationOptions` 枚举中的 `DetachedFromParent` 创建子任务时，这就无效。

取消父任务，也会取消子任务。取消架构在本章后面讨论。

20.5.4 任务的结果

任务结束时，它可以把一些有用的状态信息写到共享对象中。这个共享对象必须是线程安全的。另一个选项是使用返回某个结果的任务。使用 `Task` 类的泛型版本，就可以定义返回某个结果的任务的返回类型。

为了返回某个结果任务调用的方法可以声明为带任意返回类型。示例方法 `TaskWithResult()` 利用一个元组返回两个 `int` 值。该方法的输入可以是 `void` 或 `object` 类型，如下所示：



可从
wrox.com
下载源代码

```

static Tuple<int, int> TaskWithResult(object division)
{
    Tuple<int, int> div = (Tuple<int, int>)division;
    int result = div.Item1 / div.Item2;
    int reminder = div.Item1 % div.Item2;
    Console.WriteLine("task creates a result...");

    return Tuple.Create<int, int>(result, reminder);
}

```

代码段 TaskSamples/Program.cs



元组参见第6章。

定义一个调用 `TaskWithResult()` 方法的任务时，要使用泛型类 `Task<TResult>`。泛型参数定义了返回类型。通过构造函数，把这个方法传递给 `Func` 委托，第二个参数定义了输入值。因为这个任务在 `object` 参数中需要两个输入值，所以还创建了一个元组。接着启动该任务。`Task` 实例 `t1` 的 `Result` 属性被禁用，并一直等到该任务完成。任务完成后，`Result` 属性包含任务的结果。

```

var t1 = new Task<Tuple<int,int>> (TaskWithResult,
                                   Tuple.Create<int, int>(8, 3));

t1.Start();
Console.WriteLine(t1.Result);
t1.Wait();

```

```
Console.WriteLine("result from task: {0} {1}", t1.Result.Item1,
    t1.Result.Item2);
```

20.6 Parallel 类

在 .NET 4 中, 另一个新增的抽象线程是 `Parallel` 类。这个类定义了并行的 `for` 和 `foreach` 的静态方法。在为 `for` 和 `foreach` 定义的语言中, 循环从一个线程中运行。`Parallel` 类使用多个任务, 因此使用多个线程来完成这个作业。

`Parallel.For()` 和 `Parallel.ForEach()` 方法多次调用同一个方法, 而 `Parallel.Invoke()` 方法允许同时调用不同的方法。

20.6.1 用 `Parallel.For()` 方法循环

`Parallel.For()` 方法类似于 C# 的 `for` 循环语句, 也是多次执行一个任务。使用 `Parallel.For()` 方法, 可以并行运行迭代。迭代的顺序没有定义。

在 `For()` 方法中, 前两个参数定义了循环的开头和结束。示例从 0 迭代到 9。第 3 个参数是一个 `Action<int>` 委托。整数参数是循环的迭代次数, 该参数被传递给 `Action<int>` 委托引用的方法。`Parallel.For()` 方法的返回类型是 `ParallelLoopResult` 结构, 它提供了循环是否结束的信息。



可从
wrox.com
下载源代码

```
ParallelLoopResult result =
    Parallel.For(0, 10, i =>
    {
        Console.WriteLine("{0}, task: {1}, thread: {2}", i,
            Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(10);
    });
Console.WriteLine(result.IsCompleted);
```

代码段 `ParallelSamples/Program.cs`

在 `Parallel.For()` 的方法体中, 把索引、任务标识符和线程标识符写入控制台中。从输出可以看出, 顺序是不能保证的。程序这次的运行顺序是 0-5-1-6-2……, 有 3 个任务和 3 个线程。

```
0, task: 1, thread: 1
5, task: 2, thread: 3
1, task: 3, thread: 4
6, task: 2, thread: 3
2, task: 1, thread: 1
4, task: 3, thread: 4
7, task: 2, thread: 3
3, task: 1, thread: 1
8, task: 3, thread: 4
9, task: 3, thread: 4
True
```

也可以提前中断 `Parallel.For()` 方法。`For()` 方法的一个重载版本接受第 3 个 `Action<int, ParallelLoopState>` 类型的参数。使用这些参数定义一个方法, 就可以调用 `ParallelLoopState` 的 `Break()` 或 `Stop()` 方法, 以影响循环的结果。

注意, 迭代的顺序没有定义。

```

ParallelLoopResult result =
    Parallel.For(10, 40, (int i, ParallelLoopState pls) =>
    {
        Console.WriteLine("i: {0} task {1}", i, Task.CurrentId);
        Thread.Sleep(10);
        if (i > 15)
            pls.Break();
    });
Console.WriteLine(result.IsCompleted);
Console.WriteLine("lowest break iteration: {0}",
    result.LowestBreakIteration);

```

应用程序这次的运行说明，迭代在值大于 15 时中断，但其他任务可以同时运行，有其他值的任务也可以运行。利用 `LowestBreakIteration` 属性，可以忽略其他任务的结果。

```

10 task 1
25 task 2
11 task 1
13 task 3
12 task 1
14 task 3
16 task 1
15 task 3
False
lowest break iteration: 16

```

`Parallel.For()` 方法可能使用几个线程来执行循环。如果需要对每个线程进行初始化，就可以使用 `Parallel.For<TLocal>()` 方法。除了 `from` 和 `to` 对应的值之外，`For()` 方法的泛型版本还接受 3 个委托参数。第一个参数的类型是 `Func<TLocal>`，因为这里的例子对于 `TLocal` 使用字符串，所以该方法需要定义为 `Func<string>`，即返回 `string` 的方法。这个方法仅对于用于执行迭代的每个线程调用一次。

第二个委托参数为循环体定义了委托。在示例中，该参数的类型是 `Func<int, ParallelLoopState, string, string>`。其中第一个参数是循环迭代，第二个参数 `ParallelLoopState` 允许停止循环，如前所述。循环体方法通过第 3 个参数接收从 `init` 方法返回的值，循环体方法还需要返回一个值，其类型是用泛型 `for` 参数定义的。

`For()` 方法的最后一个参数指定一个委托 `Action<TLocal>`；在该示例中，接收一个字符串。这个方法仅对于每个线程调用一次，这是一个线程退出方法。

```

Parallel.For<string>(0, 20,
    () =>
    {
        // invoked once for each thread
        Console.WriteLine("init thread {0}, task {1}",
            Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
        return String.Format("t{0}",
            Thread.CurrentThread.ManagedThreadId);
    },
    (i, pls, str1) =>
    {
        // invoked for each member
        Console.WriteLine("body i {0} str1 {1} thread {2} task {3}", i, str1,
            Thread.CurrentThread.ManagedThreadId,

```

```

        Task.CurrentId);
    Thread.Sleep(10);
    return String.Format("i {0}", i);
},
(str1) =>
{
    // final action on each thread
    Console.WriteLine("finally {0}", str1);
});

```

运行一次这个程序的结果如下:

```

init thread 1, task 1
body i 0 str1 t1 thread 1 task 1
body i 1 str1 i 0 thread 1 task 1
init thread 3, task 2
body i 10 str1 t3 thread 3 task 2
init thread 4, task 3
body i 3 str1 t4 thread 4 task 3
body i 2 str1 i 1 thread 1 task 1
body i 11 str1 i 10 thread 3 task 2
body i 4 str1 i 3 thread 4 task 3
body i 6 str1 i 2 thread 1 task 1
body i 12 str1 i 11 thread 3 task 2
body i 5 str1 i 4 thread 4 task 3
body i 7 str1 i 6 thread 1 task 1
body i 13 str1 i 12 thread 3 task 2
body i 17 str1 i 5 thread 4 task 3
body i 8 str1 i 7 thread 1 task 1
body i 14 str1 i 13 thread 3 task 2
body i 9 str1 i 8 thread 1 task 1
body i 18 str1 i 17 thread 4 task 3
body i 15 str1 i 14 thread 3 task 2
body i 19 str1 i 18 thread 4 task 3
finally i 9
body i 16 str1 i 15 thread 3 task 2
finally i 19
finally i 16

```

20.6.2 使用 Parallel.ForEach()方法循环

Parallel.ForEach()方法遍历实现了 IEnumerable 的集合,其方式类似于 foreach 语句,但以异步方式遍历。这里也没有确定遍历顺序。



可从
wrox.com
下载源代码

```

string[] data = {"zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine",
    "ten", "eleven", "twelve"};

ParallelLoopResult result =
    Parallel.ForEach<string>(data, s =>
    {
        Console.WriteLine(s);
    });

```

代码段 ParallelSamples/Program.cs

如果需要中断循环, 就可以使用 `ForEach()` 方法的重载版本和 `ParallelLoopState` 参数。其方式与前面的 `For()` 方法相同。`ForEach()` 方法的一个重载版本也可以用于访问索引器, 从而获得迭代次数, 如下所示:

```
Parallel.ForEach<string>(data,
    (s, pls, l) =>
    {
        Console.WriteLine("{0} {1}", s, l);
    });
```

20.6.3 通过 `Parallel.Invoke()` 方法调用多个方法

如果多个任务应并行运行, 就可以使用 `Parallel.Invoke()` 方法。`Parallel.Invoke()` 方法允许传递一个 `Action` 委托数组, 在其中可以指定应运行的方法。示例代码传递了要并行调用的 `Foo()` 和 `Bar()` 方法:



可从
wrox.com
下载源代码

```
static void ParallelInvoke()
{
    Parallel.Invoke(Foo, Bar);
}

static void Foo()
{
    Console.WriteLine("foo");
}

static void Bar()
{
    Console.WriteLine("bar");
}
```

代码段 `ParallelSamples/Program.cs`

20.7 取消架构

.NET 4 包含一个新的取消架构, 允许以标准方式取消长时间运行的任务。调用的每个块都应支持这种机制。当然目前, 并不是所有调用的块都实现了这个新技术, 但越来越多调用的块都支持它。已经提供了这种机制的技术有任务、并发集合类、并行 LINQ 和几种同步机制。

取消架构基于协作行为, 它不是强制的。长时间运行的任务会检查它是否被取消, 并返回控制权。

支持取消的方法接受一个 `CancellationToken` 参数。这个类定义了 `IsCancellationRequested` 属性, 其中长时间运行的操作可以检查它是否应终止。长时间运行的操作检查取消的其他方式有: 取消标记时, 使用标记的 `WaitHandle` 属性, 或者使用 `Register()` 方法。`Register()` 方法接受 `Action` 和 `ICancelableOperation` 类型的参数。`Action` 委托引用的方法在取消标记时调用。这类似于 `ICancelableOperation`, 其中实现这个接口的对象的 `Cancel()` 方法在执行取消操作时调用。

20.7.1 `Parallel.For()` 方法的取消

以一个使用 `Parallel.For()` 方法的简单例子开始。`Parallel` 类提供了 `For()` 方法的重载版本, 在重载版本中, 可以传递 `ParallelOptions` 类型的参数。使用 `ParallelOptions` 类型, 可以传递一个 `CancellationToken` 参数。`CancellationToken` 参数通过创建 `CancellationTokenSource` 来生成。由于 `CancellationTokenSource`

实现了 `ICancelableOperation` 接口, 因此可以用 `CancellationToken` 注册, 并允许使用 `Cancel()` 方法取消操作。要取消并行循环, 应新建一个任务, 以在 500 毫秒后调用 `CancellationTokenSource` 的 `Cancel()` 方法。

在 `For()` 循环的实现代码内部, `Parallel` 类验证 `CancellationToken` 的结果, 并取消操作。一旦取消操作, `For()` 方法就抛出一个 `OperationCanceledException` 类型的异常, 这是本例捕获的异常。使用 `CancellationToken` 可以注册取消操作时的信息。为此, 需要调用 `Register()` 方法, 并传递一个在取消操作时调用的委托。



可从
WROX.COM
下载源代码

```
var cts = new CancellationTokenSource();
cts.Token.Register(() =>
    Console.WriteLine(" * * * token canceled"));

// start a task that sends a cancel after 500 ms
new Task(() =>
{
    Thread.Sleep(500);
    cts.Cancel(false);
}).Start();

try
{
    ParallelLoopResult result =
        Parallel.For(0, 100,
            new ParallelOptions()
            {
                CancellationToken = cts.Token,
            },
            x =>
            {
                Console.WriteLine("loop {0} started", x);
                int sum = 0;
                for (int i = 0; i < 100; i++)
                {
                    Thread.Sleep(2);
                    sum += i;
                }
                Console.WriteLine("loop {0} finished", x);
            });
}
catch (OperationCanceledException ex)
{
    Console.WriteLine(ex.Message);
}
```

代码段 CancellationSamples/Program.cs

运行应用程序, 会得到如下类似结果, 第 0、1、25、26、50、54、75 和 76 次迭代都启动了。这在一个有 4 个内核 CPU 的系统上运行。通过取消操作, 所有其他的迭代操作都在启动之前就取消了。启动的迭代操作允许完成, 因为取消操作总是以协作方式进行, 以避免在取消迭代操作的中间泄露资源。

```
loop 0 started
loop 50 started
```

```

loop 25 started
loop 75 started
loop 50 finished
loop 25 finished
loop 0 finished
loop 1 started
loop 26 started
loop 51 started
loop 75 finished
loop 76 started
* * token canceled
loop 1 finished
loop 51 finished
loop 26 finished
loop 76 finished
The operation was canceled.

```

20.7.2 任务的取消

同样的取消模式也可用于任务。首先，新建一个 `CancellationTokenSource`。如果仅需要一个取消标记，就可以访问 `Task.Factory.CancellationToken`，以使用默认的取消标记。接着，与前面的代码类似，新建一个任务，通过在 500 毫秒后调用 `Cancel()` 方法，把一个取消请求发送给这个 `CancellationTokenSource`。在循环中执行主要工作的任务通过 `TaskFactory` 对象接受取消标记。在构造函数中，把取消标记赋予 `TaskFactory`。这个取消标记由任务用于检查 `CancellationToken` 的 `IsCancellationRequested` 属性，以确定是否请求了取消。



可从
wrox.com
下载源代码

```

var cts = new CancellationTokenSource();
cts.Token.Register(() =>
    Console.WriteLine(" * * * task canceled"));

// start a task that sends a cancel to the
// cts after 500 ms
Task.Factory.StartNew(() =>
{
    Thread.Sleep(500);
    cts.Cancel();
});

var factory = new TaskFactory(cancellationSource.Token);
Task t1 = factory.StartNew(new Action < object > (f =>
{
    Console.WriteLine("in task");
    for (int i = 0; i < 20; i++)
    {
        Thread.Sleep(100);
        CancellationTokentoken ct = (f as TaskFactory).CancellationToken;
        if (ct.IsCancellationRequested)
        {
            Console.WriteLine("canceling was requested, " +
                "canceling from within the task");
            ct.ThrowIfCancellationRequested();
            break;
        }
    }
}

```

```

        Console.WriteLine("in loop");
    }
    Console.WriteLine("task finished without cancellation");
    }), factory, cts.Token);
try
{
    tl.Wait();
}
catch (Exception ex)
{
    Console.WriteLine("exception: {0}, {1}", ex.GetType().Name,
        ex.Message);
    if (ex.InnerException != null)
        Console.WriteLine("inner exception: {0}, {1}",
            ex.InnerException.GetType().Name,
            ex.InnerException.Message);
}
Console.WriteLine("status of the task: {0}", tl.Status);

```

代码段 CancellationSamples/Program.cs

运行应用程序，可以看到任务启动了，运行了几个循环，并获得了取消请求。之后取消任务，并抛出 `TaskCanceledException` 异常，它是从方法调用 `ThrowIfCancellationRequested()` 中启动的。调用者等待任务时，会捕获 `AggregateException` 异常，它包含内部异常 `TaskCanceledException`。例如，如果在一个也被取消的任务中运行 `Parallel.For()` 方法，这就可以用于取消的层次结构。任务的最终状态是 `Canceled`。

```

in task
in loop
in loop
in loop
in loop
* * * task canceled
canceling was requested, canceling from within the task
exception AggregateException, One or more errors occurred.
inner exception TaskCanceledException, A task was canceled.
status of the task: Canceled

```

20.8 线程问题

用多个线程编程并不容易。在启动访问相同数据的多个线程时，会间歇性地遇到难以发现的问题。如果使用任务、并行 LINQ 或 `Parallel` 类，就也会遇到这些问题。为了避免这些问题，必须特别注意同步问题和多个线程可能发生的其他问题。下面探讨与线程相关的问题：争用条件和死锁。

20.8.1 争用条件

如果两个或多个线程访问相同的对象，或者访问不同步的共享状态，就会出现争用条件。

为了说明争用条件，定义一个 `StateObject` 类，它包含一个 `int` 字段和一个 `ChangeState()` 方法。在 `ChangeState()` 方法的实现代码中，验证状态变量是否包含 5。如果它包含，就递增其值。下一条

语句是 `Trace.Assert`，它立刻验证 `state` 现在是否包含 6。

在给包含 5 的变量递增了 1 后，可能希望该变量的值就是 6。但事实不一定是这样。例如，如果一个线程刚刚执行完 `If (state == 5)` 语句，它就被其他线程抢占，调度器运行另一个线程。第二个线程现在进入 `if` 体，因为 `state` 的值仍是 5，所以将它递增到 6。第一个线程现在再次被调度，在下一条语句中，`state` 递增到 7。这时就发生了争用条件，并显示断言消息。



可从
wrox.com
下载源代码

```
public class StateObject
{
    private int state = 5;

    public void ChangeState(int loop)
    {
        if (state == 5)
        {
            state++;
            Trace.Assert(state == 6, "Race condition occurred after " +
                loop + " loops");
        }
        state = 5;
    }
}
```

代码段 ThreadingIssues/SampleTask.cs

下面通过给任务定义一个方法来验证这一点。`SampleTask` 类的 `RaceCondition()` 方法将一个 `StateObject` 类作为其参数。在一个无限 `while` 循环中，调用 `ChangeState()` 方法。变量 `i` 仅用于显示断言消息中的循环次数。



可从
wrox.com
下载源代码

```
public class SampleTask
{
    public void RaceCondition(object o)
    {
        Trace.Assert(o is StateObject, "o must be of type StateObject");
        StateObject state = o as StateObject;

        int i = 0;
        while (true)
        {
            state.ChangeState(i++);
        }
    }
}
```

代码段 ThreadingIssues/SampleTask.cs

在程序的 `Main()` 方法中，新建了一个 `StateObject` 对象，它由所有任务共享。在 `Thread` 类的构造函数中，给 `RaceCondition` 的地址传递一个 `SampleThread` 类型的对象，以创建 `Task` 对象。接着传递 `state` 对象，使用 `Start()` 方法启动这个任务。接着主线程睡眠 10 秒，假定在这段时间内，出现了争用条件。



可从
wrox.com
下载源代码

```
static void Main()
{
    var state = new StateObject();
    for (int i = 0; i < 20; i++)
    {
```

```

        new Task(new SampleTask().RaceCondition, state).Start();
    }
    Thread.Sleep(10000);
}

```

代码段 ThreadingIssues/SampleTask.cs

启动程序，就会出现争用条件。在争用条件第一次出现后，还需要多长时间才能第二次出现争用条件，取决于系统以及将程序构建为发布版本还是调试版本。如果构建为发布版本，该问题的出现次数就会比较多，因为代码被优化了。如果系统中有多 CPU 或使用双核/四核 CPU，其中多个线程可以同时运行，则该问题也会比单核 CPU 的出现次数多。在单核 CPU 中，因为线程调度是抢占式的，也会出现该问题，只是没有那么频繁。

图 20-1 显示在 75 069 个循环后，出现争用条件的程序断言。多次启动应用程序，总是会得到不同的结果。

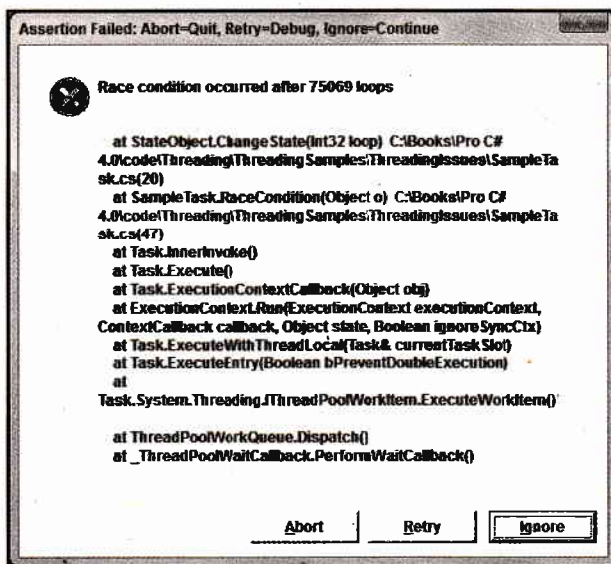


图 20-1

要避免该问题，可以锁定共享的对象。这可以在线程中完成：用下面的 lock 语句锁定在线程中共享的 state 变量。只有一个线程能在锁定块中处理共享的 state 对象。由于这个对象在所有的线程之间共享，因此如果一个线程锁定了 state，另一个线程就必须等待该锁定的解除。一旦接受锁定，线程就拥有该锁定，直到该锁定块的末尾才解除锁定。如果改变 state 变量引用的对象的每个线程都使用一个锁定，就不会出现争用条件。



可从
wrox.com
下载源代码

```

public class SampleTask
{
    public void RaceCondition(object o)
    {
        Trace.Assert(o is StateObject, "o must be of type StateObject");
        StateObject state = o as StateObject;

        int i = 0;
        while (true)
        {

```

```

        lock (state) // no race condition with this lock
        {
            state.ChangeState(i++);
        }
    }
}

```

代码段 ThreadingIssues/SampleTask.cs

在使用共享对象时，除了进行锁定之外，还可以将共享对象设置为线程安全的对象。其中 `ChangeState()` 方法包含一条 `lock` 语句。由于不能锁定 `state` 变量本身(只有引用类型才能用于锁定)，因此定义一个 `object` 类型的变量 `sync`，将它用于 `lock` 语句。如果每次 `state` 的值更改时，都使用同一个同步对象来锁定，就不会出现争用条件。



可从
wrox.com
下载源代码

```

public class StateObject
{
    private int state = 5;
    private object sync = new object();

    public void ChangeState(int loop)
    {
        lock (sync)
        {
            if (state == 5)
            {
                state++;
                Trace.Assert(state == 6, "Race condition occurred after " +
                    loop + " loops");
            }
            state = 5;
        }
    }
}

```

代码段 ThreadingIssues/SampleTask.cs

20.8.2 死锁

过多的锁定也会有麻烦。在死锁中，至少有两个线程被挂起，并等待对方解除锁定。由于两个线程都在等待对方，就出现了死锁，线程将无限等待下去。

为了说明死锁，下面实例化 `StateObject` 类型的两个对象，并把它们传递给 `SampleThread` 类的构造函数。创建两个任务，其中一个任务运行 `Deadlock1()` 方法，另一个任务运行 `Deadlock2()` 方法：



可从
wrox.com
下载源代码

```

var statel = new StateObject();
var state2 = new StateObject();
new Task(new SampleTask(statel, state2).Deadlock1).Start();
new Task(new SampleTask(statel, state2).Deadlock2).Start();

```

代码段 ThreadingIssues/Program.cs

`Deadlock1()` 和 `Deadlock2()` 方法现在改变两个对象 `s1` 和 `s2` 的状态。这是造成锁定的原因。

`Deadlock1()`方法先锁定 `s1`，接着锁定 `s2`。`Deadlock2()`方法先锁定 `s2`，再锁定 `s1`。现在，有可能 `Deadlock1()`方法中 `s1` 的锁定会被解除。接着，出现一次线程切换，`Deadlock2()`方法开始运行，并锁定 `s2`。第二个线程现在等待 `s1` 锁定的解除。因为它需要等待，所以线程调度器再次调度第一个线程，但第一个线程在等待 `s2` 锁定的解除。这两个线程现在都在等待，只要锁定块没有结束，就不会解除锁定。这是一个典型的死锁。



可从
wrox.com
下载源代码

```
public class SampleThread
{
    public SampleThread(StateObject s1, StateObject s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    private StateObject s1;
    private StateObject s2;

    public void Deadlock1()
    {
        int i = 0;
        while (true)
        {
            lock (s1)
            {
                lock (s2)
                {
                    s1.ChangeState(i);
                    s2.ChangeState(i++);
                    Console.WriteLine("still running, {0}", i);
                }
            }
        }
    }

    public void Deadlock2()
    {
        int i = 0;
        while (true)
        {
            lock (s2)
            {
                lock (s1)
                {
                    s1.ChangeState(i);
                    s2.ChangeState(i++);
                    Console.WriteLine("still running, {0}", i);
                }
            }
        }
    }
}
```

代码段 `ThreadingIssues/SampleTask.cs`

结果是，程序运行了许多次循环，不久就没有响应了。“仍在运行”的消息仅写入控制台中几次。同样，死锁问题的发生频率也取决于系统配置，每次运行的结果都不同。

在 Visual Studio 2010 中，可以在调试模式下运行程序，单击 Break All 按钮，打开 Parallel Tasks 窗口，如图 20-2 所示。其中显示，线程处于 Waiting-Deadlocked 状态。

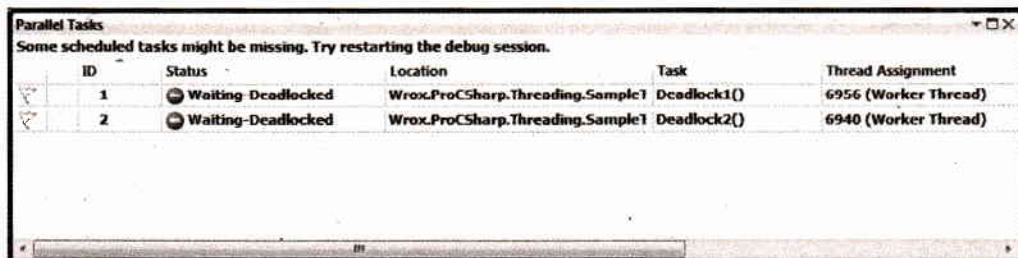


图 20-2

死锁问题并不总是如图 20-2 那样明显。一个线程锁定了 s1，接着锁定 s2；另一个线程锁定了 s2，接着锁定 s1。只需改变锁定顺序，这两个线程就会以相同的顺序进行锁定。但是，锁定可能隐藏在方法的深处。为了避免这个问题，可以在应用程序的体系架构中，从一开始就设计好锁定顺序，也可以为锁定定义超时时间。如何定义超时时间详见下一节的内容。

20.9 同步

要避免同步问题，最好不要在线程之间共享数据。当然，这并不总是可行的。如果需要共享数据，就必须使用同步技术，确保一次只有一个线程访问和改变共享状态。注意，同步问题与争用条件和死锁有关。如果不注意这些问题，就很难在应用程序中找到问题的原因，因为线程问题是不定期发生的。

本节讨论可以用于多个线程的同步技术：

- lock 语句
- Interlocked 类
- Monitor 类
- SpinLock 结构
- WaitHandle 类
- Mutex 类
- Semaphore 类
- Event 类
- ReaderWriterLockSlim 类

lock 语句、Interlocked 类和 Monitor 类可用于进程内部的同步。Mutex 类、Semaphore 类、Event 类和 ReaderWriterLockSlim 类提供了多个进程之间的线程同步。

20.9.1 lock 语句和线程安全

C# 为多个线程的同步提供了自己的关键字：lock 语句。lock 语句是设置锁定和解除锁定的一种

简单方式。

在添加 lock 语句之前, 先进入另一个争用条件。SharedState 类说明了如何使用线程之间的共享状态, 并保存一个整数值。



可从
wrox.com
下载源代码

```
public class SharedState
{
    public int State { get; set; }
}
```

代码段 SynchronizationSamples/SharedState.cs

Job 类包含 DoTheJob() 方法, 该方法是新任务的入口点。通过其实现代码, 将 SharedState 变量的 State 递增 50 000 次。sharedState 变量在这个类的构造函数中初始化:



可从
wrox.com
下载源代码

```
public class Job
{
    SharedState sharedState;
    public Job(SharedState sharedState)
    {
        this.sharedState = sharedState;
    }
    public void DoTheJob()
    {
        for (int i = 0; i < 50000; i++)
        {
            sharedState.State += 1;
        }
    }
}
```

代码段 SynchronizationSamples/Job.cs

在 Main() 方法中, 创建一个 SharedState 对象, 并把它传递给 20 个 Task 对象的构造函数。在启动所有的任务后, Main() 方法进入另一个循环, 使 20 个任务全部处于等待状态, 直到所有的任务都执行完毕为止。任务执行完毕后, 把共享状态的合计值写入控制台中。因为执行了 50 000 次循环, 有 20 个任务, 所以写入控制台的值应是 1 000 000。但是, 事实常常并非如此。



可从
wrox.com
下载源代码

```
class Program
{
    static void Main()
    {
        int numTasks = 20;
        var state = new SharedState();
        var tasks = new Task[numTasks];

        for (int i = 0; i < numTasks; i++)
        {
            tasks[i] = new Task(new Job(state).DoTheJob);
            tasks[i].Start();
        }

        for (int i = 0; i < numTasks; i++)
        {
            tasks[i].Wait();
        }
    }
}
```

```

    }
    Console.WriteLine("summarized {0}", state.State);
}
}
}

```

代码段 SynchronizationSamples/Program.cs

多次运行应用程序的结果如下所示:

```

summarized 785895
summarized 776131
summarized 774400
summarized 871286

```

每次运行的结果都不同,但没有一个结果是正确的。调试版本和发布版本的差别很大。根据使用的 CPU 类型,结果也不一样。如果将循环次数改为比较小的值,就会多次得到正确的值,但不是每次。这个应用程序非常小,很容易看出问题,但该问题的原因在大型应用程序中就很难确定。

必须在这个程序中添加同步功能,这可以用 `lock` 关键字实现。

用 `lock` 语句定义的对象表示,要等待指定对象的锁定。只能传递引用类型。锁定值类型只是锁定了副本,这没有什么意义。如果对值类型使用了 `lock` 语句, C# 编译器就会提供一个错误。进行了锁定后——只有锁定一个线程,就可以运行 `lock` 语句块。在 `lock` 语句块的最后,对象的锁定被解除,另一个等待锁定的线程就可以获得该锁定块了。

```

lock (obj)
{
    // synchronized region
}

```

要锁定静态成员,可以把锁放在 `object` 类型上:

```

lock (typeof(StaticClass))
{
}

```

使用 `lock` 关键字可以将类的实例成员设置为线程安全的。这样,一次只有一个线程能访问相同实例的 `DoThis()` 和 `DoThat()` 方法。

```

public class Demo
{
    public void DoThis()
    {
        lock (this)
        {
            // only one thread at a time can access the DoThis and DoThat methods
        }
    }
    public void DoThat()
    {
        lock (this)
        {
        }
    }
}

```

但是，因为实例的对象也可以用于外部的同步访问，我们不能在类自身中控制这种访问，所以应采用 **SyncRoot** 模式。通过 **SyncRoot** 模式，创建一个私有对象 **syncRoot**，将这个对象用于 **lock** 语句。

```
public class Demo
{
    private object syncRoot = new object();

    public void DoThis()
    {
        lock (syncRoot)
        {
            // only one thread at a time can access the DoThis and DoThat methods
        }
    }

    public void DoThat()
    {
        lock (syncRoot)
        {
        }
    }
}
```

使用锁定需要时间，且并不总必需。可以创建类的两个版本，一个同步版本，一个异步版本。这里通过修改 **Demo** 类来说明。**Demo** 类本身并不是同步的，这可以在 **DoThis()**和 **DoThat()**方法的实现中看出。该类还定义了 **IsSynchronized** 属性，客户可以从该属性中获得类的同步选项信息。为了获得该类的同步版本，可以使用静态方法 **Synchronized()**传递一个非同步对象，这个方法会返回 **SynchronizedDemo** 类型的对象。**SynchronizedDemo** 实现为派生自基类 **Demo** 的一个内部类，并重写基类的虚拟成员。重写的成员使用了 **SyncRoot** 模式。

```
public class Demo
{
    private class SynchronizedDemo: Demo
    {
        private object syncRoot = new object();
        private Demo d;

        public SynchronizedDemo(Demo d)
        {
            this.d = d;
        }

        public override bool IsSynchronized
        {
            get { return true; }
        }

        public override void DoThis()
        {
            lock (syncRoot)
            {
                d.DoThis();
            }
        }
    }
}
```

```

        public override void DoThat()
        {
            lock (syncRoot)
            {
                d.DoThat();
            }
        }
    }

    public virtual bool IsSynchronized
    {
        get { return false; }
    }

    public static Demo Synchronized(Demo d)
    {
        if (!d.IsSynchronized)
        {
            return new SynchronizedDemo(d);
        }
        return d;
    }

    public virtual void DoThis()
    {
    }

    public virtual void DoThat()
    {
    }
}

```

必须注意, 在使用 **SynchronizedDemo** 类时, 只有方法是同步的。对这个类的两个成员的调用并没有同步。

首先修改异步的 **SharedState** 类, 以使用 **SyncRoot** 模式。如果试图用 **SyncRoot** 模式锁定对属性的访问, 使 **SharedState** 类变成线程安全的, 就仍会出现前面描述的争用条件。



可从
wrox.com
下载源代码

```

public class SharedState
{
    private int state = 0;
    private object syncRoot = new object();

    public int State // there's still a race condition,
                    // don't do this!
    {
        get { lock (syncRoot) {return state; }}
        set { lock (syncRoot) {state = value; }}
    }
}

```

代码段 [SynchronizationSamples/SharedState.cs](#)

调用方法 **DoTheTask()** 方法的线程访问 **SharedState** 类的 **get** 存取器, 以获得 **state** 的当前值, 接

着 get 存取器给 state 设置新值。在调用对象的 get 和 set 存取器期间，对象没有锁定，另一个线程可以获得临时值。



可从
wrox.com
下载源代码

```
public void DoTheJob()
{
    for (int i = 0; i < 50000; i++)
    {
        sharedState.State += 1;
    }
}
```

代码段 SynchronizationSamples/Job.cs

所以，最好不改变 SharedState 类，让它依旧没有线程安全性。



可从
wrox.com
下载源代码

```
public class SharedState
{
    public int State { get; set; }
}
```

代码段 SynchronizationSamples/SharedState.cs

然后在 DoTheTask() 方法中，将 lock 语句添加到合适的地方：



可从
wrox.com
下载源代码

```
public void DoTheJob()
{
    for (int i = 0; i < 50000; i++)
    {
        lock (sharedState)
        {
            sharedState.State += 1;
        }
    }
}
```

代码段 SynchronizationSamples/Job.cs

这样，应用程序的结果就总是正确的：

```
summarized 1000000
```



在一个地方使用 lock 语句并不意味着，访问对象的其他线程都正在等待。必须对每个访问共享状态的线程显式地使用同步功能。

当然，还必须修改 SharedState 类的设计，并作为一个原子操作提供递增方式。这是一个设计问题——什么是类的原子功能？



可从
wrox.com
下载源代码

```
public class SharedState
{
    private int state = 0;
    private object syncRoot = new object();

    public int State
```

```

    {
        get { return state; }
    }

    public int IncrementState()
    {
        lock (syncRoot)
        {
            return ++state;
        }
    }
}

```

代码段 SynchronizationSamples/SharedState.cs

锁定状态的递增还有一种更快的方式，如下节所示。

20.9.2 Interlocked 类

Interlocked 类用于使变量的简单语句原子化。`i++`不是线程安全的，它的操作包括从内存中获取一个值，给该值递增 1，再将它存储回内存。这些操作都可能会被线程调度器打断。**Interlocked** 类提供了以线程安全的方式递增、递减、交换和读取值的方法。

与其他同步技术相比，使用 **Interlocked** 类会快得多。但是，它只能用于简单的同步问题。

例如，这里不使用 `lock` 语句锁定对 `someState` 变量的访问，把它设置为一个新值，以防它是空的，而可以使用 **Interlocked** 类，它比较快：



可从
wrox.com
下载源代码

```

lock (this)
{
    if (someState == null)
    {
        someState = newState;
    }
}

```

代码段 SynchronizationSamples/SharedState.cs

这个功能相同但比较快的版本使用了 `Interlocked.CompareExchange()` 方法：

```

Interlocked.CompareExchange<SomeState>(ref someState,
    newState, null);

```

并且不在 `lock` 语句中执行递增操作：

```

public int State
{
    get
    {
        lock (this)
        {
            return ++state;
        }
    }
}

```

而使用较快的 `Interlocked.Increment()` 方法:

```
public int State
{
    get
    {
        return Interlocked.Increment(ref state);
    }
}
```

20.9.3 Monitor 类

C# 的 `lock` 语句由编译器解析为使用 `Monitor` 类。下面的 `lock` 语句:

```
lock (obj)
{
    // synchronized region for obj
}
```

被解析为调用 `Enter()` 方法, 该方法会一直等待, 直到线程被对象锁定为止。一次只有一个线程能被对象锁定。只要解除了锁定, 线程就可以进入同步阶段。`Monitor` 类的 `Exit()` 方法解除了锁定。编译器把 `Exit()` 方法放在 `try` 块的 `finally` 处理程序中, 所以如果抛出了异常, 就会解除该锁定。



`try/finally` 详见第 15 章。



可从
wrox.com
下载源代码

```
Monitor.Enter(obj);
try
{
    // synchronized region for obj
}
finally
{
    Monitor.Exit(obj);
}
```

代码段 `SynchronizationSamples/Program.cs`

与 C# 的 `lock` 语句相比, `Monitor` 类的主要优点是: 可以添加一个等待被锁定的超时值。这样就不会无限期地等待被锁定, 而可以使用 `TryEnter()` 方法, 其中给它传递一个超时值, 指定等待被锁定的最长时间。如果 `obj` 被锁定, `TryEnter()` 方法就把布尔型的引用参数设置为 `true`, 并同步地访问由对象 `obj` 锁定的状态。如果另一个线程锁定 `obj` 的时间超过了 500 毫秒, `TryEnter()` 方法就把变量 `lockTaken` 设置为 `false`, 线程不再等待, 而是用于执行其他操作。也许在以后, 该线程会尝试再次被锁定。

```
bool lockTaken = false;
Monitor.TryEnter(obj, 500, ref lockTaken);
if (lockTaken)
{
    try
    {

```

```

        // acquired the lock
        // synchronized region for obj
    }
    finally
    {
        Monitor.Exit(obj);
    }
}
else
{
    // didn't get the lock, do something else
}

```

20.9.4 SpinLock 结构

SpinLock 结构是.NET 4 新增的。如果基于对象的锁定对象(Monitor)的系统开销由于垃圾回收而过高,就可以使用 SpinLock 结构。如果有大量的锁定(例如,列表中的每个节点都有一个锁定),且锁定的时间总是非常短,SpinLock 结构就很有用。应避免使用多个 SpinLock 结构,也不要调用任何可能阻塞的内容。

除了体系结构上的区别之外,SpinLock 结构的用法非常类似于 Monitor 类。获得锁定使用 Enter()或 TryEnter()方法,释放锁定使用 Exit()方法。SpinLock 结构还提供了属性 IsHeld 和 IsHeldByCurrentThread,指定它当前是否是锁定的。



传送 SpinLock 实例时要小心。因为 SpinLock 定义为结构,把一个变量赋予另一个变量会创建一个副本。总是通过引用传送 SpinLock 实例。

20.9.5 WaitHandle 基类

WaitHandle 是一个抽象基类,用于等待一个信号的设置。可以等待不同的信号,因为 WaitHandle 是一个基类,可以从中派生一些类。

在本章前面使用异步委托时,已经使用了 WaitHandle 基类。异步委托的 BeginInvoke()方法返回一个实现了 IAsyncResult 接口的对象。使用 IAsyncResult 接口,可以用 AsyncWaitHandle 属性访问 WaitHandle 基类。在调用 WaitOne()方法时,线程会等待接收一个与等待句柄相关的信号。



可从
wrox.com
下载源代码

```

static void Main()
{
    TakesAWhileDelegate dl = TakesAWhile;
    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (true)
    {
        Console.WriteLine(".");
        if (ar.AsyncWaitHandle.WaitOne(50, false))
        {
            Console.WriteLine("Can get the result now");
            break;
        }
    }
}

```

```

int result = dl.EndInvoke(ar);
Console.WriteLine("result: {0}", result);
}

```

代码段 AsyncDelegate/Program.cs

使用 `WaitHandle` 基类可以等待一个信号的出现(`WaitOne()`方法)、等待必须发出信号的多个对象(`WaitAll()`方法)、或者等待多个对象中的一个(`WaitAny()`方法)。`WaitAll()`和 `WaitAny()`是 `WaitHandle` 类的静态方法,接收一个 `WaitHandle` 参数数组。

`WaitHandle` 基类有一个 `SafeWaitHandle` 属性,其中可以将一个本机句柄赋予一个操作系统资源,并等待该句柄。例如,可以指定一个 `SafeWaitHandle` 等待文件 I/O 操作的完成,或者指定自定义 `SafeTransactionHandle`,参见第 23 章。

因为 `Mutex`、`EventWaitHandle` 和 `Semaphore` 类派生自 `WaitHandle` 基类,所以可以在等待时使用它们。

20.9.6 Mutex 类

`Mutex`(mutual exclusion, 互斥)是 .NET Framework 中提供跨多个进程同步访问的一个类。它非常类似于 `Monitor` 类,因为它们都只有一个线程能拥有锁定。只有一个线程能获得互斥锁定,访问受互斥保护的同步代码区域。

在 `Mutex` 类的构造函数中,可以指定互斥是否最初应由主调线程拥有,定义互斥的名称,获得互斥是否已存在的信息。在下面的示例代码中,第 3 个参数定义为输出参数,接收一个表示互斥是否为新建的布尔值。如果返回的值是 `false`,就表示互斥已经定义。互斥可以在另一个进程中定义,因为操作系统能够识别有名称的互斥,它由不同的进程共享。如果没有给互斥指定名称,互斥就是未命名的,不在不同的进程之间共享。

```

bool createdNew;
Mutex mutex = new Mutex(false, "ProCSharpMutex", out createdNew);

```

要打开已有的互斥,还可以使用 `Mutex.OpenExisting()`方法,它不需要用构造函数创建互斥时需要的相同 .NET 权限。

由于 `Mutex` 类派生自基类 `WaitHandle`,因此可以利用 `WaitOne()`方法获得互斥锁定,在该过程中成为该互斥的拥有者。调用 `ReleaseMutex()`方法,即可释放互斥。

```

if (mutex.WaitOne())
{
    try
    {
        // synchronized region
    }
    finally
    {
        mutex.ReleaseMutex();
    }
}
else
{
    // some problem happened while waiting
}

```

由于系统能识别有名称的互斥, 因此可以使用它禁止应用程序启动两次。在下面的 Windows 窗体应用程序中, 调用了 `Mutex` 对象的构造函数。接着, 验证名称为 `SingletonWinAppMutex` 的互斥是否存在。如果存在, 应用程序就退出。

```
static class Program
{
    [STAThread]
    static void Main()
    {
        bool createdNew;
        Mutex mutex = new Mutex(false, "SingletonWinAppMutex",
                                out createdNew);

        if (!createdNew)
        {
            MessageBox.Show("You can only start one instance " +
                             "of the application");
            Application.Exit();
            return;
        }

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

20.9.7 Semaphore 类

信号量非常类似于互斥, 其区别是, 信号量可以同时由多个线程使用。信号量是一种计数的互斥锁定。使用信号量, 可以定义允许同时访问受旗语锁定保护的资源的线程个数。如果有许多可用资源, 且只允许一定数量的线程访问该资源, 就可以使用信号量。例如, 要访问系统上的物理 I/O 端口, 且有 3 个端口可用, 就允许 3 个线程同时访问 I/O 端口, 但第 4 个线程需要等待前 3 个线程中的一个释放资源。

.NET 4 为信号量功能提供了两个类 `Semaphore` 和 `SemaphoreSlim`。`Semaphore` 类可以命名, 使用系统范围内的资源, 允许在不同进程之间同步。`SemaphoreSlim` 类是对于较短等待时间进行了优化的轻型版本。

在下面的示例应用程序中, 在 `Main()` 方法中创建了 6 个线程和一个计数为 4 的信号量。在 `Semaphore` 类的构造函数中, 定义了锁定个数的计数, 它可以用信号量(第二个参数)来获得, 还定义了最初释放的锁定数(第一个参数)。如果第一个参数的值小于第二个参数, 它们的差就是已经分配线程的计数值。与互斥一样, 也可以给信号量指定名称, 使之在不同的进程之间共享。这里定义信号量时没有指定名称, 所以它只能在这个进程中使用。在创建了 `SemaphoreSlim` 对象之后, 启动 6 个线程, 它们都获得了相同的信号量。



可从
wrox.com
下载源代码

```
using System;
using System.Threading;
using System.Diagnostics;

namespace Wrox.ProCSharp.Threading
{
    class Program
```

```

static void Main()
{
    int threadCount = 6;
    int semaphoreCount = 4;
    var semaphore = new SemaphoreSlim(semaphoreCount, semaphoreCount);
    var threads = new Thread[threadCount];

    for (int i = 0; i < threadCount; i++)
    {
        threads[i] = new Thread(ThreadMain);
        threads[i].Start(semaphore);
    }

    for (int i = 0; i < threadCount; i++)
    {
        threads[i].Join();
    }

    Console.WriteLine("All threads finished");
}

```

代码段 Semaphore/Program.cs

在线程的主方法 `ThreadMain()` 中，线程利用 `WaitOne()` 方法锁定信号量。信号量的计数是 4，所以有 4 个线程可以获得锁定。第 5 个线程必须等待，这里还定义了最长的等待时间为 500 毫秒。如果在该等待时间过后未能获得锁定，线程就把一条消息写入控制台，在循环中继续等待。只要获得了锁定，线程就把一条消息写入控制台，睡眠一段时间，然后解除锁定。在解除锁定时，在任何情况下一定要解除资源的锁定，这一点很重要。这就是在 `finally` 处理程序中调用 `Semaphore` 类的 `Release()` 方法的原因。

```

static void ThreadMain(object o)
{
    SemaphoreSlim semaphore = o as SemaphoreSlim;
    Trace.Assert(semaphore != null, "o must be a Semaphore type");
    bool isCompleted = false;
    while (!isCompleted)
    {
        if (semaphore.Wait(600))
        {
            try
            {
                Console.WriteLine("Thread {0} locks the semaphore",
                    Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(2000);
            }
            finally
            {
                semaphore.Release();
                Console.WriteLine("Thread {0} releases the semaphore",
                    Thread.CurrentThread.ManagedThreadId);
                isCompleted = true;
            }
        }
        else

```

```

        Console.WriteLine("Timeout for thread {0}; wait again",
            Thread.CurrentThread.ManagedThreadId);
    }
}

```

运行应用程序，可以看到有 4 个线程很快被锁定。ID 为 7 和 8 的线程需要等待。该等待会重复进行，直到其中一个被锁定的线程之一解除了信号量。

```

Thread 3 locks the semaphore
Thread 4 locks the semaphore
Thread 5 locks the semaphore
Thread 6 locks the semaphore
Timeout for thread 8; wait again
Timeout for thread 7; wait again
Timeout for thread 8; wait again
Timeout for thread 7; wait again
Timeout for thread 7; wait again
Timeout for thread 8; wait again
Thread 3 releases the semaphore
Thread 8 locks the semaphore
Thread 4 releases the semaphore
Thread 7 locks the semaphore
Thread 5 releases the semaphore
Thread 6 releases the semaphore
Thread 8 releases the semaphore
Thread 7 releases the semaphore
All threads finished

```

20.9.8 Events 类

事件是另一个系统范围内的资源同步方法。为了从托管代码中使用系统事件，.NET Framework 在 `System.Threading` 名称空间中提供了 `ManualResetEvent`、`AutoResetEvent`、`ManualResetEventSlim` 和 `CountdownEvent` 类。`ManualResetEventSlim` 和 `CountdownEvent` 类是 .NET 4 新增的。



第 8 章介绍了 C# 中的 `event` 关键字，它与 `System.Threading` 名称空间中的 `event` 类没有关系。`event` 关键字基于委托，而上述两个 `event` 类是 .NET 封装器，用于系统范围内的手机事件资源的同步。

可以使用事件通知其他任务：这里有一些数据，并完成了一些操作等。事件可以发信号，也可以不发信号。使用前面介绍的 `WaitHandle` 类，任务可以等待处于发信号状态的事件。

调用 `Set()` 方法，即可向 `ManualResetEventSlim` 发信号。调用 `Reset()` 方法，可以使之返回不发信号的状态。如果多个线程等待向一个事件发信号，并调用了 `Set()` 方法，就释放所有等待的线程。另外，如果一个线程刚刚调用了 `WaitOne()` 方法，但事件已经发出信号，等待的线程就可以继续等待。

也通过 `Set()` 方法向 `AutoResetEvent` 发信号。也可以使用 `Reset()` 方法使之返回不发信号的状态。但是，如果一个线程在等待自动重置的事件发信号，当第一个线程的等待状态结束时，该事件会自

动变为不发信号的状态。这样，如果多个线程在等待向事件发信号，就只有一个线程结束其等待状态，它不是等待时间最长的线程，而是优先级最高的线程。

为了说明 `ManualResetEventSlim` 类的事件，下面的 `Calculator` 类定义了 `Calculation()` 方法，这是任务的入口点。在这个方法中，该任务接收用于计算的输入数据，将结果写入变量 `result`，该变量可以从 `Result` 属性来访问。只要完成了计算(在随机的一段时间过后)，就调用 `ManualResetEventSlim` 类的 `Set()` 方法，向事件发信号。



可从
wrox.com
下载源代码

```
public class Calculator
{
    private ManualResetEventSlim mEvent;

    public int Result { get; private set; }

    public Calculator(ManualResetEventSlim ev)
    {
        this.mEvent = ev;
    }

    public void Calculation(object obj)
    {
        Tuple<int, int> data = (Tuple<int, int>)obj;
        Console.WriteLine("Task {0} starts calculation", Task.Current.Id);
        Thread.Sleep(new Random().Next(3000));
        Result = data.Item1 + data.Item2;

        // signal the event-completed!
        Console.WriteLine("Task {0} is ready", Task.Current.Id);
        mEvent.Set();
    }
}
```

代码段 EventSample/Program.cs

程序的 `Main()` 方法定义了包含 4 个 `ManualResetEventSlim` 对象的数组和包含 4 个 `Calculator` 对象的数组。每个 `Calculator` 在构造函数中用一个 `ManualResetEventSlim` 对象初始化，这样每个任务在完成时都有自己的事件对象来发信号。现在使用 `TaskFactory` 类，让不同的任务执行计算任务。

```
class Program
{
    static void Main()
    {
        const int taskCount = 4;

        var mEvents = new ManualResetEventSlim[taskCount];
        var waitHandles = new WaitHandle[taskCount];
        var calcs = new Calculator[taskCount];

        TaskFactory taskFactory = new TaskFactory();
        for (int i = 0; i < taskCount; i++)
        {
            mEvents[i] = new ManualResetEventSlim(false);
            waitHandles[i] = mEvents[i].WaitHandle;
            calcs[i] = new Calculator(mEvents[i]);

            taskFactory.StartNew(calcs[i].Calculation, Tuple.Create(i + 1, i + 3));
        }
    }
}
```

```

    }
    //...

```

`WaitHandle` 类现在用于等待数组中的任意一个事件。`WaitAny()`方法等待向任意一个事件发信号。与 `ManualResetEvent` 对象不同, `ManualResetEventSlim` 对象不派生自 `WaitHandle` 类。因此有一个 `WaitHandle` 对象的集合, 它从 `ManualResetEventSlim` 类的 `WaitHandle` 属性中填充。从 `WaitAny()`方法返回的 `index` 值匹配传递给 `WaitAny()`方法的事件数组的索引, 以提供发信号的事件的相关信息, 使用该索引可以从这个事件中读取结果。



可从
wtox.com
下载源代码

```

        for (int i = 0; i < taskCount; i++)
        {
            int index = WaitHandle.WaitAny(mEvents);
            if (index == WaitHandle.WaitTimeout)
            {
                Console.WriteLine("Timeout!!");
            }
            else
            {
                mEvents[index].Reset();
                Console.WriteLine("finished task for {0}, result: {1}",
                                index, calcs[index].Result);
            }
        }
    }
}

```

代码段 EventSample/Program.cs

启动应用程序, 可以看到线程在进行计算, 设置事件, 以通知主线程, 它可以读取结果了。在任意时间, 依据是调试版本还是发布版本和硬件的不同, 会看到按照不同的顺序, 线程池中有不同数量的线程在执行任务。这里重用了线程池中的第 4 个线程, 完成了两个任务, 因为它比较快, 能第一个完成计算。

```

Task 1 starts calculation
Task 2 starts calculation
Task 2 is ready
Task 3 starts calculation
finished task for 0, result: 4
Task 4 starts calculation
Task 3 is ready
finished task for 2, result: 8
Task 1 is ready
finished task for 1, result: 6
Thread 4 is ready
finished task for 3, result: 10

```

在一个类似的场景中, 为了把一些工作分支到多个任务中, 并在以后合并结果, 使用新的 `CountdownEvent` 类很有用。不需要为每个任务创建一个单独的事件对象, 而只需创建一个事件对象。`CountdownEvent` 类为所有设置了事件的任务定义了一个初始数字, 在到达该计数后, 就向 `CountdownEvent` 类发信号。

修改 `Calculator` 类, 以使用 `CountdownEvent` 类替代 `ManualResetEvent` 类。不使用 `Set()`方法设置

信号, 而使用 `CountdownEvent` 类定义 `Signal()` 方法。



可从
wrox.com
下载源代码

```
public class Calculator
{
    private CountdownEvent cEvent;

    public int Result { get; private set; }

    public Calculator(CountdownEvent ev)
    {
        this.cEvent = ev;
    }

    public void Calculation(object obj)
    {
        Tuple<int, int> data = (Tuple<int, int>)obj;
        Console.WriteLine("Task {0} starts calculation", Task.Current.Id);
        Thread.Sleep(new Random().Next(3000));
        Result = data.Item1 + data.Item2;

        // signal the event-completed!
        Console.WriteLine("Task {0} is ready", Task.Current.Id);
        cEvent.Signal();
    }
}
```

代码段 EventSample/Calculator.cs

`Main()` 方法现在可以简化, 从而使它只需等待一个事件了。如果不像前面那样单独处理结果, 这个新版本就很不错。



可从
wrox.com
下载源代码

```
const int taskCount = 4;
var cEvent = new CountdownEvent(taskCount);
var calcs = new Calculator[taskCount];

var taskFactory = new TaskFactory();
for (int i = 0; i < taskCount; i++)
{
    calcs[i] = new Calculator(cEvent);

    taskFactory.StartNew(calcs[i].Calculation,
        Tuple.Create(i + 1, i + 3));
}

cEvent.Wait();
Console.WriteLine("all finished");
for (int i = 0; i < taskCount; i++)
{
    Console.WriteLine("task for {0}, result: {1}", i, calcs[i].Result);
}
```

代码段 EventSample/Program.cs

20.9.9 Barrier 类

.NET 4 为同步提供了新的 `Barrier` 类。`Barrier` 类非常适用于其中工作有多个任务分支且以后又

需要合并工作的情况。**Barrier** 类用于需要同步的参与者。激活一个作业时，就可以动态地添加其他参与者，例如，从父任务中创建子任务。参与者在继续之前，可以等待所有其他参与者完成其工作。

示例应用程序使用一个包含 2 000 000 个字符串的集合。使用多个任务遍历该集合，并统计以 a、b、c 等开头的字符串个数。

FillData()方法创建一个集合，并用随机字符串填充它：



可从
wrox.com
下载源代码

```
public static IEnumerable<string> FillData(int size)
{
    List<string> data = new List<string>(size);
    Random r = new Random();
    for (int i = 0; i < size; i++)
    {
        data.Add(GetString(r));
    }
    return data;
}

private static string GetString(Random r)
{
    StringBuilder sb = new StringBuilder(6);
    for (int i = 0; i < 6; i++)
    {
        sb.Append((char)(r.Next(26) + 97));
    }
    return sb.ToString();
}
```

代码段 BarrierSample/Program.cs

CalculationInTask()方法定义了任务执行的作业。通过参数接收一个包含 4 项的元组。第 3 个数是对 **Barrier** 实例的引用。任务完成其作业时，任务就会使用 **RemoveParticipant()**方法从 **Barrier** 类中删除它自己。

```
static int[] CalculationInTask(object p)
{
    var pl = p as Tuple<int, int, Barrier, List<string>>;
    Barrier barrier = pl.Item3;
    List<string> data = pl.Item4;

    int start = pl.Item1 * pl.Item2;
    int end = start + pl.Item2;
    Console.WriteLine("Task {0}: partition from {1} to {2}",
        Task.Current.Id, start, end);
    int[] charCount = new int[26];
    for (int j = start; j < end; j++)
    {
        char c = data[j][0];
        charCount[c - 97]++;
    }
    Console.WriteLine("Calculation completed from task {0}. {1} " +
        "times a, {2} times z", Task.Current.Id, charCount[0],
        charCount[25]);

    barrier.RemoveParticipant();
}
```

```

        Console.WriteLine("Task {0} removed from barrier, " +
            "remaining participants {1}", Task.Current.Id,
            barrier.ParticipantsRemaining);
        return charCount;
    }

```

在 `Main()` 方法中创建一个 `Barrier` 实例。在构造函数中，可以指定参与者的数量。在该示例中，这个数量是 3，因为该示例创建了两个任务，`Main()` 方法本身也是一个参与者。使用 `TaskFactory` 创建两个任务，把遍历集合的任务分为两个部分。启动该任务后，使用 `SignalAndWait()` 方法，`Main()` 方法在完成时发出信号，并等待所有其他参与者或者发出完成的信号，或者从 `Barrier` 类中删除它们。一旦所有的参与者都准备好，就提取任务的结果，并使用 `Zip()` 扩展方法把它们合并起来。

```

static void Main()
{
    const int numberTasks = 2;
    const int partitionSize = 1000000;

    var data = new List<string>(FillData(partitionSize * numberTasks));

    var barrier = new Barrier(numberTasks + 1);
    var taskFactory = new TaskFactory();
    var tasks = new Task<int[]>(numberTasks);
    for (int i = 0; i < participants; i++)
    {
        tasks[i] = taskFactory.StartNew<int[]>(CalculationInTask,
            Tuple.Create(i, partitionSize, barrier, data));
    }

    barrier.SignalAndWait();
    var resultCollection = tasks[0].Result.Zip(tasks[1].Result, (c1, c2) =
    {
        return c1 + c2;
    });

    char ch = 'a';
    int sum = 0;
    foreach (var x in resultCollection)
    {
        Console.WriteLine("{0}, count: {1}", ch++, x);
        sum += x;
    }

    Console.WriteLine("main finished {0}", sum);
    Console.WriteLine("remaining {0}", barrier.ParticipantsRemaining);
}

```

20.9.10 ReaderWriterLockSlim 类

为了使锁定机制允许锁定多个读取器(而不是一个写入器)访问某个资源，可以使用 `ReaderWriterLockSlim` 类。这个类提供了一个锁定功能，如果没有写入器锁定资源，就允许多个读取器访问资源，但只能有一个写入器锁定该资源。

`ReaderWriterLockSlim` 类的属性可获得读取阻塞或不阻塞的锁定，如 `EnterReadLock()` 和 `TryEnterReadLock()` 方法。还可以使用 `EnterWriteLock()` 和 `TryEnterWriteLock()` 方法获得写入锁定。如果任

务先读取资源，之后写入资源，它就可以使用 `EnterUpgradableReadLock()` 或 `TryEnterUpgradableReadLock()` 方法获得可升级的读取锁定。有了这个锁定，就可以获得写入锁定，而无需释放读取锁定。

这个类的几个属性提供了当前锁定的相关信息，如 `CurrentReadCount`、`WaitingReadCount`、`WaitingUpgradableReadCount` 和 `WaitingWriteCount`。

下面的示例程序创建了一个包含 6 项的集合和一个 `ReaderWriterLockSlim` 对象。`ReaderMethod()` 方法获得一个读取锁定，读取列表中的所有项，并把它们写到控制台中。`WriterMethod()` 方法试图获得一个写入锁定，以改变集合的所有值。在 `Main()` 方法中，启动 6 个线程，以调用 `ReaderMethod()` 或 `WriterMethod()` 方法。



可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Threading
{
    class Program
    {
        private static List<int> items = new List<int>() { 0, 1, 2, 3, 4, 5};
        private static ReaderWriterLockSlim rwl =
            new ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

        static void ReaderMethod(object reader)
        {
            try
            {
                rwl.EnterReadLock();

                for (int i = 0; i < items.Count; i++)
                {
                    Console.WriteLine("reader {0}, loop: {1}, item: {2}",
                        reader, i, items[i]);
                    Thread.Sleep(40);
                }
            }
            finally
            {
                rwl.ExitReadLock();
            }
        }

        static void WriterMethod(object writer)
        {
            try
            {
                while (!rwl.TryEnterWriteLock(50))
                {
                    Console.WriteLine("Writer {0} waiting for the write lock",
                        writer);
                    Console.WriteLine("current reader count: {0}",
                        rwl.CurrentReadCount);
                }
                Console.WriteLine("Writer {0} acquired the lock", writer);
            }
        }
    }
}
```

```

        for (int i = 0; i < items.Count; i++)
        {
            items[i]++;
            Thread.Sleep(50);
        }
        Console.WriteLine("Writer {0} finished", writer);
    }
    finally
    {
        rwl.ExitWriteLock();
    }
}

static void Main()
{
    var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
        TaskContinuationOptions.None);
    var tasks = new Task[6];
    tasks[0] = taskFactory.StartNew(WriterMethod, 1);
    tasks[1] = taskFactory.StartNew(ReaderMethod, 1);
    tasks[2] = taskFactory.StartNew(ReaderMethod, 2);
    tasks[3] = taskFactory.StartNew(WriterMethod, 2);
    tasks[4] = taskFactory.StartNew(ReaderMethod, 3);
    tasks[5] = taskFactory.StartNew(ReaderMethod, 4);

    for (int i = 0; i < 6; i++)
    {
        tasks[i].Wait();
    }
}
}

```

代码段 ReaderWriterSample/Program.cs

运行这个应用程序，第一个写入器先获得锁定。第二个写入器和所有的读取器需要等待。接着，读取器可以同时工作，而第二个写入器仍在等待资源。

```

Writer 1 acquired the lock
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 2 waiting for the write lock
current reader count: 0
Writer 1 finished
reader 4, loop: 0, item: 1
reader 1, loop: 0, item: 1
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 0, item: 1
reader 3, loop: 0, item: 1
reader 4, loop: 1, item: 2
reader 1, loop: 1, item: 2

```

```
reader 3, loop: 1, item: 2
reader 2, loop: 1, item: 2
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 2, item: 3
reader 1, loop: 2, item: 3
reader 2, loop: 2, item: 3
reader 3, loop: 2, item: 3
Writer 2 waiting for the write lock
current reader count: 4
reader 4, loop: 3, item: 4
reader 1, loop: 3, item: 4
reader 2, loop: 3, item: 4
reader 3, loop: 3, item: 4
reader 4, loop: 4, item: 5
reader 1, loop: 4, item: 5
Writer 2 waiting for the write lock
current reader count: 4
reader 2, loop: 4, item: 5
reader 3, loop: 4, item: 5
reader 4, loop: 5, item: 6
reader 1, loop: 5, item: 6
reader 2, loop: 5, item: 6
reader 3, loop: 5, item: 6
Writer 2 waiting for the write lock
current reader count: 4
Writer 2 acquired the lock
Writer 2 finished
```

20.10 Timer 类

.NET Framework 提供了几个 Timer 类，用于在某个时间间隔后调用某个方法。表 20-1 列出了 Timer 类及其名称空间和功能。

表 20-1

名 称 空 间	说 明
System.Threading	System.Threading 名称空间中的 Timer 类提供了核心功能。在构造函数中，可以传递一个委托，该委托应按照指定的时间间隔调用
System.Timers	System.Timers 名称空间中的 Timer 类是一个组件，因为它派生自 Component 基类。因此，可以把它从工具箱拖放到服务器应用程序(如 Windows 服务)的设计界面上。 这个 Timer 类使用 System.Threading.Timer，但提供了基于事件的机制，而不是基于委托的机制
System.Windows.Forms	使用 System.Threading 和 System.Timers 名称空间中的 Timer 类，可以从不是主调线程的另一个线程中调用回调方法或事件方法。Windows 窗体控件绑定到创建它的线程上。对这个线程的回调通过 System.Windows.Forms 名称空间中的 Timer 类完成
System.Web.UI	System.Web.UI 名称空间中的 Timer 类是一个 AJAX 扩展，该扩展可以用于 Web 页面
System.Windows.Threading	System.Windows.Threading 名称空间中的 DispatcherTimer 类由 WPF 应用程序使用。 DispatcherTimer 类运行在 UI 线程上

使用 `System.Threading.Timer` 类, 可以把要调用的方法作为构造函数的第一个参数传递。这个方法必须满足 `TimeCallback` 委托的要求, 该委托定义一个 `void` 返回类型和一个 `object` 参数。通过第二个参数, 可以传递任意对象, 用回调方法中的 `object` 参数接收对应的对象。例如, 可以传递 `Event` 对象, 向调用者发送信号。第 3 个参数指定第一次调用回调方法时的时间段。最后一个参数指定了回调的重复时间间隔。如果计时器应只触发一次, 就把第 4 个参数设置为 `-1`。

如果创建 `Timer` 对象后应改变时间间隔, 就可以用 `Change()` 方法传递新值:



可从
wrox.com
下载源代码

```
private static void ThreadingTimer()
{
    var t1 = new System.Threading.Timer(
        TimeAction, null, TimeSpan.FromSeconds(2),
        TimeSpan.FromSeconds(3));

    Thread.Sleep(15000);

    t1.Dispose();
}

static void TimeAction(object o)
{
    Console.WriteLine("System.Threading.Timer {0:T}", DateTime.Now);
}
```

代码段 TimerSample/Program.cs

`System.Timer` 名称空间中的 `Timer` 类的构造函数只需要一个时间间隔。经过该时间间隔后应调用的方法用 `Elapsed` 事件指定。这个事件需要一个 `ElapsedEventHandler` 类型的委托, 这个委托需要的对象和 `ElapsedEventArgs` 参数与 `TimeAction()` 方法相同。`AutoReset` 属性指定计时器是否重复触发。如果把这个属性设置为 `false`, 事件就只触发一次。调用 `Start()` 方法允许计时器触发事件。除了调用 `Start()` 方法之外, 还可以把 `Enabled` 属性设置为 `true`。在后台, `Start()` 方法什么也不做。`Stop()` 方法把 `Enabled` 属性设置为 `false`, 以停止计时器。

```
private static void TimersTimer()
{
    var t1 = new System.Timers.Timer(1000);
    t1.AutoReset = true;
    t1.Elapsed += TimeAction;
    t1.Start();
    Thread.Sleep(10000);
    t1.Stop();

    t1.Dispose();
}

static void TimeAction(object sender, System.Timers.ElapsedEventArgs e)
{
    Console.WriteLine("System.Timers.Timer {0:T}", e.SignalTime);
}
```

20.11 基于事件的异步模式

本章前面介绍了基于 `IAsyncResult` 接口的异步模式。在异步回调中，回调线程不同于主调线程。使用 Windows 窗体或 WPF 时，这是一个问题，因为 Windows 窗体和 WPF 控件绑定到一个线程上。对于每个控件，都只能从创建该控件的线程中调用方法。也就是说，如果有一个后台线程，就不能直接从这个线程中访问 UI 控件。

通过 Windows 窗体控件，唯一可以从除了创建它的线程之外的线程中调用的是 `Invoke()`、`BeginInvoke()`、`EndInvoke()` 方法和 `InvokeRequired` 属性。`BeginInvoke()` 和 `EndInvoke()` 方法是 `Invoke()` 方法的异步版本。这些方法会切换到创建控件的线程上，以调用赋予一个委托参数的方法，该委托参数可以传递给这些方法。这些方法的使用并不简单，这就是 .NET 2.0 新组件和新异步模式一起开发的原因，即基于事件的异步模式。

在基于事件的异步模式中，异步组件提供了后缀为 `Async` 的方法。例如，同步方法 `DoATask()` 的异步版本是 `DoATaskAsync()`。为了得到结果信息，组件还需要定义一个后缀为 `Completed` 的事件，如 `DoATaskCompleted`。`DoATaskAsync()` 方法中的操作在后台线程中运行时，会在主调线程中触发 `DoATaskCompleted` 事件。

通过基于事件的异步模式，异步组件可以选择性地支持取消操作，并提供进度信息。对于取消操作，方法的名称应是 `CancelAsync()`。对于进度信息，应提供后缀为 `ProgressChanged` 的事件，例如，`DoATaskProgressChanged`。

20.11.1 BackgroundWorker 类

`BackgroundWorker` 类是异步事件模式的一种实现方案。`RunWorkerAsync()` 方法启动与 `DoWork` 事件异步关联的方法。工作完成后，就触发 `RunWorkerCompleted` 事件。这里使用的名称完全符合前面介绍的基于事件的异步模式。

如果把 `WorkerSupportsCancellation` 属性设置为 `true`，`BackgroundWorker` 类就可以也支持取消功能。此时，工作可以通过调用 `CancelAsync()` 方法来取消。但已完成的工作需要使用 `CancellationPending` 属性验证它是否应停止其工作。与新的取消架构类似，`BackgroundWorker` 类也支持协作式取消。

在显示进度信息方面，`BackgroundWorker` 类也是有用的。`WorkerReportsProgress` 属性指定正在进行的任务是否返回进度信息。如果返回，就可以指定 `ProgressChanged` 事件，来接收进度变化信息。完成该工作的任务需要调用 `ReportProgress()` 方法，报告已完成任务的进度百分比信息。



另一个实现异步事件模式的类是 `System.Net` 名称空间中的 `WebClient` 组件。这个类使用 `WebRequest` 和 `WebResponse` 类，但提供了一个易于使用的接口。`WebRequest` 和 `WebResponse` 类也提供了异步编程方式，它基于包含 `IAsyncResult` 接口的异步模式。

下面的示例应用程序说明了 `BackgroundWorker` 控件在 WPF 应用程序中的用法，具体方法是执行一个需要一定时间的任务。新建一个 WPF 应用程序，在窗体上添加 3 个标签控件、3 个文本框控件、两个按钮控件和一个进度条控件，如图 20-3 所示。

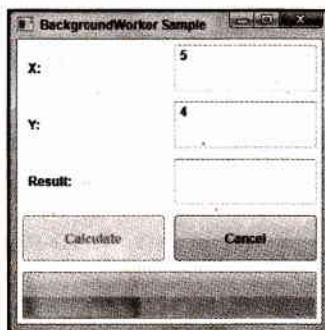


图 20-3

按表 20-2 配置控件的属性。

表 20-2

控 件	属性和事件	值
标签	Content	X:
文本框	x:Name	textX
标签	Content	Y:
文本框	x:Name	textY
标签	Content	Result:
文本框	x:Name	textResult
按钮	x:Name	buttonCalculate
	Text	Calculate
	Click	OnCalculate
按钮	x:Name	buttonCancel
	Text	Cancel
	IsEnabled	False
	Click	OnCancel
进度条	x:Name	progressBar

在代码隐藏中，给 BackgroundWorkerWindow 添加一个 BackgroundWorker 对象，把 DoWork 事件赋予 OnDoWork() 方法。对于 BackgroundWorker 类，需要导入 System.ComponentModel 名称空间。



可从
wrox.com
下载源代码

```
public partial class BackgroundWorkerWindow : Window
{
    private BackgroundWorker backgroundWorker;

    public BackgroundWorkerWindow()
    {
        InitializeComponent();
        backgroundWorker = new BackgroundWorker();
        backgroundWorker.DoWork += OnDoWork;
    }
}
```

代码段 BackgroundWorkersSample/Window1.xaml.cs

`OnCalculate()`方法是按钮控件 `buttonCalculate` 的 `Click` 事件的事件处理程序。在实现过程中,因为 `buttonCalculate` 按钮被禁用,所以在计算完成之前,用户不能再次单击该按钮。要启动 `BackgroundWorker` 控件,可调用 `RunWorkerAsync()`方法。`BackgroundWorker` 使用线程池中的一个线程来完成计算。`RunWorkerAsync()`方法需要将输入参数传递给赋予 `DoWork` 事件的处理程序。

```
private void OnCalculate(object sender, RoutedEventArgs e)
{
    this.buttonCalculate.IsEnabled = false;
    this.textResult.Text = String.Empty;
    this.buttonCancel.IsEnabled = true;
    this.progressBar.Value = 0;

    backgroundWorker.RunWorkerAsync(Tuple.Create<int, int>(int.Parse(textX.Text),
        int.Parse(textY.Text)));
}
```

`OnDoWork()`方法连接到 `BackgroundWorker` 控件的 `DoWork` 事件上。在 `DoWorkEventArgs` 中,通过 `Argument` 属性接收输入参数。其实现代码模拟的功能需要一定的执行时间和 5 秒的睡眠时间。在睡眠时间过后,将计算的结果写入 `DoEventArgs` 的 `Result` 属性中。如果将计算和睡眠时间添加到 `OnCalculate()`方法中,在用户输入时,Windows 应用程序就不能获得用户的输入。但是,这里使用一个单独的线程,用户界面仍是激活的。

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
    var t = e.Argument as Tuple<int, int>;

    Thread.Sleep(5000);
    e.Result = t.Item1 + t.Item2;
}
```

`OnDoWork()`方法完成后, `BackgroundWorker` 控件就触发 `RunWorkerCompleted` 事件。`OnWorkCompleted()`方法与这个事件相关。这里从 `RunWorkerCompletedEventArgs` 参数的 `Result` 属性中接收结果,把该结果写入文本框控件 `result` 中。在触发该事件时, `BackgroundWorker` 控件会把控制权交给创建它的线程,所以不需要使用 WPF 控件的 `Invoke()`方法,而可以直接调用 UI 控件的属性和方法。

```
private void OnWorkCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    this.textResult.Text = e.Result.ToString();

    this.buttonCalculate.IsEnabled = true;
    this.buttonCancel.IsEnabled = false;
}
```

现在可以测试应用程序,会发现计算过程独立于 UI 线程,UI 仍是激活的,窗体可以四处移动。但是,取消功能和进度条功能仍需要实现。

20.11.2 启用取消功能

要启用取消功能，以停止正在运行的线程的进度，必须把 `BackgroundWorker` 控件的 `WorkerSupportsCancellation` 属性设置为 `true`。接着，必须实现与 `buttonCancel` 控件的 `Click` 事件相连接的 `OnCancel` 处理程序。`BackgroundWorker` 控件的 `CancelAsync()` 方法可以取消正在进行的异步任务。



可从
wrox.com
下载源代码

```
private void OnCancel(object sender, RoutedEventArgs e)
{
    backgroundWorker.CancelAsync();
}
```

代码段 BackgroundWorkersSample/Window1.xaml.cs

异步任务不会自动取消。在执行异步任务的 `OnDoWork()` 处理程序中，必须修改其实现代码，以检查 `BackgroundWorker` 控件的 `CancellationPending` 属性。这个属性在调用 `CancelAsync()` 方法时设置。如果某个取消操作被挂起，就把 `DoWorkEventArgs` 的 `Cancel` 属性设置为 `true`，并退出处理程序。

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
    var t = e.Argument as Tuple<int, int>;

    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);

        if (backgroundWorker.CancellationPending)
        {
            e.Cancel = true;
            return;
        }

        e.Result = t.Item1 + t.Item2;
    }
}
```

如果异步方法成功地完成或被取消，就调用完整的处理程序 `OnWorkCompleted()`。如果取消了该方法，就不能访问 `Result` 属性，因为这会抛出一个 `InvalidOperationException` 异常，并显示操作被取消的信息。所以必须检查 `RunWorkerCompletedEventArgs` 的 `Cancelled` 属性，并根据不同的情况执行不同的操作：

```
private void OnWorkCompleted(object sender,
                             RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        this.textResult.Text = "Cancelled";
    }
    else
    {
        this.textResult.Text = e.Result.ToString();
    }
    this.buttonCalculate.IsEnabled = true;
}
```

```

        this.buttonCancel.IsEnabled = false;
    }

```

再次运行应用程序，就可以从用户界面中取消异步进度功能。

20.11.3 启用进度功能

为了获得用户界面的进度信息，必须将 `BackgroundWorker` 控件的 `WorkerReportsProgress` 属性设置为 `true`。

在 `OnDoWork()` 方法中，可以用 `ReportProgress()` 方法报告 `BackgroundWorker` 控件的进度。



可从
wrox.com
下载源代码

```

private void OnDoWork(object sender, DoWorkEventArgs e)
{
    var t = e.Argument as Tuple<int, int> ;

    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);
        backgroundWorker.ReportProgress(i * 10);
        if (backgroundWorker.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
    }

    e.Result = t.Item1 + t.Item2;
}

```

代码段 BackgroundWorkersSample/Window1.xaml.cs

`ReportProgress()` 方法触发 `BackgroundWorker` 控件的 `ProgressChanged` 事件，这个事件会将控制权交给 UI 线程。

在 `ProgressChanged` 事件中添加 `OnProgressChanged()` 方法，在其实现代码中，给进度条控件设置一个从 `ProgressChangedEventArgs` 的 `ProgressPercentage` 属性中接收的新值。

```

private void OnProgressChanged(object sender, ProgressChangedEventArgs e)
{
    this.progressBar.Value = e.ProgressPercentage;
}

```

在 `OnWorkCompleted()` 事件处理程序中，进度条最终设置为 100%。

```

private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        this.textBoxResult.Text = "Canceled";
    }
    else
    {
        this.textBoxResult.Text = e.Result.ToString();
    }
    this.buttonCalculate.Enabled = true;
}

```

```

        this.buttonCancel.Enabled = false;
        this.progressBar.Value = 100;
    }

```

20.11.4 创建基于事件的异步组件

要创建一个支持基于事件的异步模式的自定义组件，需要做更多的工作。为了用一个简单的情形说明这个过程，`AsyncComponent` 类只在某个时间段后返回一个转换后的输入字符串，这与前面的同步方法 `LongTask()` 相同。为了提供异步支持，公共接口提供了异步方法 `LongTaskAsync()` 和事件 `LongTaskCompleted`。这个事件的类型是 `EventHandler<LongTaskCompletedEventArgs>`，它定义了类型为 `object` 和 `LongTaskCompletedEventArgs` 的参数。`LongTaskCompletedEventArgs` 类型在后面创建，调用者可以从中读取异步操作的结果。

另外，还需要一些帮助方法，如 `DoLongTask()` 和 `CompletionMethod()`，如下所述。



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
    public delegate void LongTaskCompletedEventHandler(object sender,
        LongTaskCompletedEventArgs e);

    public partial class AsyncComponent: Component
    {
        private Dictionary<object, AsyncOperation> userStateDictionary =
            new Dictionary<object, AsyncOperation>();
        private SendOrPostCallback onCompletedDelegate;

        public event EventHandler<LongTaskCompletedEventArgs> LongTaskCompleted;

        public AsyncComponent()
        {
            InitializeComponent();
            InitializeDelegates();
        }

        public AsyncComponent(IContainer container)
        {
            container.Add(this);

            InitializeComponent();
            InitializeDelegates();
        }

        private void InitializeDelegates()
        {
            onCompletedDelegate = LongTaskCompletion;
        }

        public string LongTask(string input)
        {
            Console.WriteLine("LongTask started");
            Thread.Sleep(5000);
        }
    }
}

```

```

        Console.WriteLine("LongTask finished");
        return input.ToUpper();
    }

    public void LongTaskAsync(string input, object taskId)
    {
        //.
    }

    private void LongTaskCompletion(object operationState)
    {
        //.
    }

    protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
    {
        //.
    }

    // running in a background thread
    private void DoLongTask(string input, AsyncOperation asyncOp)
    {
        //.
    }

    private void CompletionMethod(string output, Exception ex,
        bool cancelled, AsyncOperation asyncOp)
    {
        //.
    }
}

public class LongTaskCompletedEventArgs: AsyncCompletedEventArgs
{
    //.
}
}

```

代码段 AsyncComponent/AsyncComponent.cs

`LongTaskAsync()`方法需要异步地启动同步的操作。如果组件允许同时启动几次异步任务，客户就需要选择把不同的结果映射到启动的任务上。因此 `LongTaskAsync()`方法的第二个参数必须是一个 `taskId`，客户机可以使用它映射结果。当然，在组件内部，必须记住任务 ID，才能映射结果。.NET 提供了 `AsyncOperationManager` 类，来创建 `AsyncOperationObjects`，从而有助于跟踪操作的状态。`AsyncOperationManager` 类有一个方法 `CreateOperation()`，可以给它传递一个任务标识符，该方法就返回一个 `AsyncOperation` 对象。这个操作保存为前面创建的 `userStateDictionary` 字典中的一项。

接着，创建一个 `Action<string, AsyncOperation>` 类型的委托，把 `DoLongTask()` 方法赋予委托实例。`BeginInvoke()` 是委托的一个方法，通过它可以使用线程池中的一个线程异步启动地 `DoLongTask()` 方法。这个委托需要的参数是调用者的所有输入参数和 `AsyncOperation` 参数，它们用于获得状态，并映射操作的结果。

```

public void LongTaskAsync(string input, object taskId)
{
    AsyncOperation asyncOp = AsyncOperationManager.CreateOperation(taskId);

```

```

lock (userStateDictionary)
{
    if (userStateDictionary.ContainsKey(taskId))
        throw new ArgumentException("taskId must be unique", "taskId");

    userStateDictionary[taskId] = asyncOp;
}

Action<string, AsyncOperation> longTaskDelegate = DoLongTask;
longTaskDelegate.BeginInvoke(input, asyncOp, null, null);
}

```

DoLongTask()方法现在使用委托来异步地调用。可以调用同步方法 **LongTask()**，来获得输出值。

因为在同步方法中可能出现的异常不应影响到后台线程，所以应捕获任何异常，并用 **Exception** 类型的变量 **e** 保存它。最后，调用 **CompletionMethod()**方法，向调用者通报结果。

```

// running in a background thread
private void DoLongTask(string input, AsyncOperation asyncOp)
{
    Exception e = null;
    string output = null;
    try
    {
        output = LongTask(input);
    }
    catch (Exception ex)
    {
        e = ex;
    }

    this.CompletionMethod(output, e, false, asyncOp);
}

```

通过 **CompletionMethod()**方法的实现代码，删除操作时清除了 **userStateDictionary**。**AsyncOperation** 对象的 **PostOperationCompleted()**方法结束了异步操作的生存期，并使用 **onCompletedDelegate()**方法通知调用者。这个方法确保在线程上根据应用程序类型的需要调用委托。要给调用者通报信息，应创建一个 **LongTaskCompletedEventArgs** 类型的对象，并把它传递给 **PostOperationCompleted()**方法。

```

private void CompletionMethod(string output, Exception ex,
    bool cancelled, AsyncOperation asyncOp)
{
    lock (userStateDictionary)
    {
        userStateDictionary.Remove(asyncOp.UserSuppliedState);
    }

    // results of the operation
    asyncOp.PostOperationCompleted(onCompletedDelegate,
        new LongTaskCompletedEventArgs(output, ex, cancelled,
            asyncOp.UserSuppliedState));
}

```

为了给调用者传递信息，**LongTaskCompletedEventArgs** 类应派生自 **AsyncCompletedEventArgs** 基类，并添加一个包含输出信息的属性。在构造函数中，调用基类的构造函数，来传递异常信息、

取消信息和用户状态信息。

```
public class LongTaskCompletedEventArgs: AsyncCompletedEventArgs
{
    public LongTaskCompletedEventArgs(string output, Exception e,
        bool cancelled, object state)
        : base(e, cancelled, state)
    {
        this.output = output;
    }

    private string output;

    public string Output
    {
        get
        {
            RaiseExceptionIfNecessary();
            return output;
        }
    }
}
```

`asyncOp.PostOperationCompleted()`方法使用了 `onCompletedDelegate()`方法。初始化这个委托以便引用 `LongTaskCompletion()`方法。`LongTaskCompletion()`方法必须满足 `SendOrPostCallbackDelegate()`方法的参数要求。实现代码把参数强制转换为 `LongTaskCompletedEventArgs` 类型，这是传递给 `PostOperationCompleted()`方法的对象的类型，再调用 `onLongTaskCompleted()`方法。

```
private void LongTaskCompletion(object operationState)
{
    var e = operationState as LongTaskCompletedEventArgs;

    OnLongTaskCompleted(e);
}
```

接着，`onLongTaskCompleted()`方法触发 `LongTaskCompleted`事件，给调用者返回 `LongTaskCompletedEventArgs`。

```
protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
{
    if (LongTaskCompleted != null)
    {
        LongTaskCompleted(this, e);
    }
}
```

创建了该组件后，使用它确实很容易。把 `LongTaskCompleted`事件赋予 `Comp_LongTaskCompleted()`方法，并调用 `LongTaskAsync()`方法。在一个简单的控制台应用程序中，事件处理程序 `Comp_LongTaskCompleted`从不是主线程的另一个线程中调用(这不同于 Windows 窗体应用程序，如下所述)。

```
static void Main()
{
```

```

        Console.WriteLine("Main thread: {0}",
            Thread.CurrentThread.ManagedThreadId);

        AsyncComponent comp = new AsyncComponent();
        comp.LongTaskCompleted += Comp_LongTaskCompleted;
        comp.LongTaskAsync("input", 33);

        Console.ReadLine();
    }

    static void Comp_LongTaskCompleted(object sender,
        LongTaskCompletedEventArgs e)
    {
        Console.WriteLine("completed, result: {0}, thread: {1}", e.Output,
            Thread.CurrentThread.ManagedThreadId);
    }

```

代码段 AsyncComponent/Program.cs

通过一个 Windows 窗体应用程序把 `SynchronizationContext` 设置为 `WindowsFormsSynchronizationContext`，因此在同一个线程中调用事件处理程序：

```

WindowsFormsSynchronizationContext syncContext =
    new WindowsFormsSynchronizationContext();
SynchronizationContext.SetSynchronizationContext(syncContext);

```

20.12 小结

本章介绍了如何通过 `System.Threading` 名称空间编写多线程应用程序，和如何通过 `System.Threading.Tasks` 名称空间编写多任务应用程序。在应用程序中使用多线程要仔细规划。太多的线程会导致资源问题，线程不足又会使应用程序执行缓慢，且执行效果也不好。使用任务可以获得线程的抽象。这个抽象有助于创建非常多的线程，因为线程是在池中重用的。

我们探讨了创建多线程的各种方法，如使用委托、计时器、`ThreadPool` 和 `Thread` 类。还探讨了各种同步技术，如简单的 `lock` 语句、`Monitor`、`Semaphore` 和 `Event` 类。学习了如何用 `IAsyncResult` 接口编写异步模式，以及基于事件的异步模式。

.NET Framework 中的 `System.Threading` 名称空间提供了处理线程的多种方式，但这并不意味着 .NET Framework 完成多线程中所有困难的任务。我们必须考虑线程的优先级和同步问题。本章讨论了这些问题，并介绍了如何在 C# 应用程序中为它们编码。还论述了与死锁和争用条件相关的问题。

如果要在 C# 应用程序中使用多线程功能，就必须仔细规划。

下面是有关线程的几条规则：

- 尽力使同步要求最低。同步很复杂，且会阻塞线程。如果尝试避免共享状态，就可以避免同步。当然，这不总是可行。
- 类的静态成员应是线程安全的。通常，.NET Framework 中的类满足这个要求。
- 实例状态不需要是线程安全的。为了得到最佳性能，最好在类的外部使用同步功能，且不对类的每个成员使用同步功能。.NET Framework 类的实例成员一般不是线程安全的。在 MSDN 库中，对于 Framework 的每个类在“线程安全性”部分中可以找到相应的归档信息。

下一章介绍另一个 .NET 核心主题：安全性。

第21章

安全性

本章内容:

- 身份验证和授权
- 加密
- 资源的访问控制
- 代码访问安全性

为了确保应用程序的安全,安全性有几个重要方面需要考虑。一是应用程序的用户,访问应用程序的是一个真正的用户,还是伪装成用户的某个人?如何确定这个用户是可以信任的?如本章所述,用户首先需要进行身份验证,再进行授权,以验证该用户是否可以使用需要的资源。

对于在网络上存储或发送的数据呢?例如,有人可以通过网络嗅探器访问这些数据吗?这里,数据的加密很重要。一些技术,如 Windows Communication Foundation(WCF),通过简单的配置提供了加密功能,所以可以看到后台执行了什么操作。

另一方面是应用程序本身。如果应用程序驻留在 Web 提供程序上,该怎么办?如何禁止应用程序执行对服务器有害的操作?

本章将讨论.NET 中有助于管理安全性的一些特性,其中包括.NET 怎样避开恶意代码、怎样管理安全性策略,以及怎样通过编程访问安全子系统等。

21.1 身份验证和授权

安全性的两个基本支柱是身份验证和授权。身份验证是标识用户的过程,授权在验证了所标识用户是否可以访问特定资源之后进行。

21.1.1 标识和 Principal

使用标识可以验证运行应用程序的用户。WindowsIdentity 类表示一个 Windows 用户。如果没有用 Windows 账户标识用户,也可以使用实现了 IIdentity 接口的其他类。通过这个接口可以访问用户名、该用户是否通过身份验证,以及验证类型等信息。

Principal 是一个包含用户的标识和用户的所属角色的对象。IPrincipal 接口定义了 Identity 属性和 IsInRole()方法,Identity 属性返回 IIdentity 对象;在 IsInRole()方法中,可以验证用户是否是指定角色的一个成员。角色是有相同安全权限的用户集合,同时它是用户的管理单元。角色可以是

Windows 组或自己定义的一个字符串集合。

.NET 中的 `Principal` 类有 `WindowsPrincipal` 和 `GenericPrincipal`。还可以创建实现了 `IPrincipal` 接口的自定义 `Principal` 类。

下面创建一个控制台应用程序，它可以访问某个应用程序中的主体，以便允许用户访问底层的 Windows 账户。这里需要导入 `System.Security.Principal` 和 `System.Threading` 名称空间。首先，必须指定 .NET 使用底层的 Windows 账户自动挂起主体。因为从安全的角度考虑，.NET 不会自动填充线程的 `CurrentPrincipal` 属性。完成这项工作的代码如下所示：



可从
wrox.com
下载源代码

```
using System;
using System.Security.Principal;
using System.Threading;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {
            AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
        }
    }
}
```

代码段 `WindowsPrincipal/Program.cs`

使用 `WindowsIdentity.GetCurrent()` 方法可以访问 Windows 账户的详细信息，但是，这个方法最好在只访问主体一次时使用。如果要多次访问主体，比较有效的方法是设置策略，以便当前的线程能够访问主体。如果使用 `SetPrincipalPolicy()` 方法时，就指定当前线程中的主体应保存一个 `WindowsIdentity` 对象。所有的标识类(如 `WindowsIdentity`)都实现 `IIdentity` 接口，该接口包含 3 个属性(`AuthenticationType`、`IsAuthenticated` 和 `Name`)，便于所有的派生标识类实现它们。

下面添加一些代码，从 `Thread` 对象中访问主体的属性：

```
WindowsPrincipal principal =
    (WindowsPrincipal)Thread.CurrentPrincipal;
WindowsIdentity identity = (WindowsIdentity)principal.Identity;
Console.WriteLine("IdentityType: " + identity.ToString());
Console.WriteLine("Name: {0}", identity.Name);
Console.WriteLine("'Users'? {0} ",
    principal.IsInRole(WindowsBuiltInRole.User));
Console.WriteLine("'Administrators'? {0}",
    principal.IsInRole(WindowsBuiltInRole.Administrator));
Console.WriteLine("Authenticated: {0}", identity.IsAuthenticated);
Console.WriteLine("AuthType: {0}", identity.AuthenticationType);
Console.WriteLine("Anonymous? {0}", identity.IsAnonymous);
Console.WriteLine("Token: {0}", identity.Token);
}
```

从控制台应用程序中输出的结果类似如下文本行。当然，根据计算机的配置和与账户相关的角色(在该账户下用户进行签名)，实际输出的结果也不尽相同：

```
IdentityType: System.Security.Principal.WindowsIdentity
Name: farabove\christian
```

```
'Users'? True
'Administrators'? False
Authenticated: True
AuthType: NTLM
Anonymous? False
Token: 416
```

很明显,如果能很容易访问当前用户及其角色的详细信息,然后使用那些信息决定允许或拒绝用户执行某些动作,这就非常有好处。利用角色和 Windows 用户组,管理员可以完成使用标准用户管理工具所能完成的工作,这样,在用户的角色改变时,通常可以避免更改代码。下面将详细讨论角色。

21.1.2 角色

基于角色的安全性可以很好地解决资源的访问问题。例如,在金融行业中,员工的角色决定了他们能够访问的信息和它们能够执行的操作。

此外,基于角色的安全性最好也与 Windows 账户或自定义用户目录一起使用,以便管理基于 Web 的资源的访问权限。例如,Web 站点可以限制用户对其内容的访问,直到用户用那个站点注册为止,而且只有用户成为那个 Web 站点的付费订阅者之后,才能访问站点上的特殊内容。在众多的方法中,只有 ASP.NET 能更容易实现基于角色的安全性,因为许多代码都基于服务器。

例如,如果来实现一个需要身份验证的 Web 服务,就可以使用 Windows 的账户子系统,并以这种方式编写一个 Web 方法。但是,要确保用户在访问该方法的功能之前,成为某一特定 Windows 用户组的成员。

设想有一个依赖于 Windows 账户的内联网应用程序的情景。系统有一个 Manager 组和一个 Assistant 组,根据用户在公司中的角色,把用户分配到其中的一个组中。假设应用程序包含一个显示员工信息的特性,且只允许 Managers 组中的成员访问这个特性。很容易使用代码检查当前的用户是否是 Manager 组的成员,以此来决定是否允许该用户访问该特性。

但是,如果以后决定重新安排账户组,并引入一个新组 Personnel,这个组的成员也可以访问员工的详细信息,就会出问题。此时就必须仔细检查并更新所有代码,以包括这个新组的规则。

更好的解决方案是创建权限(如 ReadEmployeeDetails),把它赋予需要权限的组。如果代码对 ReadEmployeeDetails 权限应用了某项检查,更新应用程序以允许 Personnel 组中的成员访问员工信息就变得非常简单,只需创建一个组,并把用户放到那个组中,然后把 ReadEmployeeDetails 权限赋予那个组即可。

21.1.3 声明基于角色的安全性

如同代码访问的安全性一样,可以使用命令式的请求,方法是调用 IPincipal 类的 IsInRole()方法或使用属性,实现基于角色的安全性请求(用户必须是 Administrator 组中的成员)。在类或方法级别上,可以使用[PincipalPermission]特性声明性地说明权限的需求,其代码如下所示:



可从
wrox.com
下载源代码

```
using System;
using System.Security;
using System.Security.Principal;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
```

```

{
    class Program
    {
        static void Main()
        {
            AppDomain.CurrentDomain.SetPrincipalPolicy(
                PrincipalPolicy.WindowsPrincipal);
            try
            {
                ShowMessage();
            }
            catch (SecurityException exception)
            {
                Console.WriteLine("Security exception caught ({0})", exception.Message);
                Console.WriteLine("The current principal must be in the local"
                    + "Users group");
            }
        }

        [PrincipalPermission(SecurityAction.Demand, Role = "BUILTIN\\Users")]
        static void ShowMessage()
        {
            Console.WriteLine("The current principal is logged in locally ");
            Console.WriteLine("(member of the local Users group)");
        }
    }
}

```

代码段 RoleBasedSecurity/Program.cs

除非在 Windows 本地 Users 组的用户环境中执行应用程序，否则 ShowMessage() 方法将抛出一个异常。对于 Web 应用程序，运行 ASP.NET 代码的账户必须处于组中，但在实际应用中一定不会把这个账户添加到管理员组中。

如果使用本地 User 组中的账户运行上面的代码，则输出结果如下所示：

```

The current principal is logged in locally
(member of the local Users group)

```

21.1.4 客户端应用程序服务

Visual Studio 很容易对 ASP.NET Web 应用程序使用以前建立的身份验证服务。使用这个服务可以对 Windows 应用程序和 Web 应用程序使用相同的身份验证机制。这是一个提供程序模型，它主要基于 System.Web.Security 名称空间中的 Membership 和 Roles 类。使用 Membership 类可以验证、创建、删除和查找用户，改变密码以及与用户相关的许多其他操作。使用 Roles 类可以添加和删除角色，给用户获取角色，以及改变用户的角色。角色和用户的存储位置取决于提供程序。ActiveDirectoryMembershipProvider 访问 Active Directory 中的用户和角色，SqlMembershipProvider 使用 SQL Server 数据库。对于客户端应用程序服务，.NET 4 有两个提供程序：ClientFormsAuthenticationMembershipProvider 和 ClientWindowsAuthenticationMembershipProvider。

下面使用客户端应用程序服务和 Forms 身份验证。为此，首先需要启动一个应用程序服务器，然后才能从 Windows 窗体或 WPF 中使用这个服务。

1. 应用程序服务

要使用客户端应用程序服务，可以创建一个 ASP.NET Web 服务项目，它提供了应用程序服务。

该项目需要一个成员提供程序。不仅可以使已有的成员提供程序，也可以创建自定义提供程序。这里的示例代码定义了 `SampleMembershipProvider` 类，它派生自基类 `MembershipProvider`，该基类在 `System.Web.ApplicationServices` 程序集的 `System.Web.Security` 名称空间中定义。必须重写基类中的所有抽象方法。对于登录，只需实现 `ValidateUser()` 方法。所有其他方法都可以用 `ApplicationName` 属性抛出一个 `NotSupportedException` 异常。这里的示例代码使用包含用户名和密码的 `Dictionary<string, string>`。当然，也可以改为使用自己的实现代码，如从数据库中读取用户名和密码。



```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Web.Security;

namespace Wrox.ProCSharp.Security
{
    public class SampleMembershipProvider: MembershipProvider
    {
        private Dictionary<string, string> users = new Dictionary<string, string>();
        internal static string ManagerUserName = "Manager".ToLowerInvariant();
        internal static string EmployeeUserName = "Employee".ToLowerInvariant();

        public override void Initialize(string name, NameValueCollection config)
        {
            users = new Dictionary<string, string>();
            users.Add(ManagerUserName, "secret@Pa$$w0rd");
            users.Add(EmployeeUserName, "s0me@Secret");

            base.Initialize(name, config);
        }

        public override string ApplicationName
        {
            get
            {
                throw new NotImplementedException();
            }
            set
            {
                throw new NotImplementedException();
            }
        }

        // override abstract Membership members
        // ...

        public override bool ValidateUser(string username, string password)
        {
            if (users.ContainsKey(username.ToLowerInvariant()))
            {
                return password.Equals(users[username.ToLowerInvariant()]);
            }
        }
    }
}
```

```

    }
    return false;
}
}
}

```

代码段 AppServices/SampleMembershipProvider.cs

为了使用角色，还需要实现一个角色提供程序。SampleRoleProvider 类派生自基类 RoleProvider，并实现 GetRolesForUser()和 IsUserInRole()方法：



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Specialized;
using System.Web.Security;

namespace Wrox.ProCSharp.Security
{
    public class SampleRoleProvider: RoleProvider
    {
        internal static string ManagerRoleName = "Manager".ToLowerInvariant();
        internal static string EmployeeRoleName = "Employee".ToLowerInvariant();

        public override void Initialize(string name, NameValueCollection config)
        {
            base.Initialize(name, config);
        }

        public override void AddUsersToRoles(string[] usernames,
            string[] roleNames)
        {
            throw new NotImplementedException();
        }

        // override abstract RoleProvider members
        // ...

        public override string[] GetRolesForUser(string username)
        {
            if (string.Compare(username,
                SampleMembershipProvider.ManagerUserName, true) == 0)
            {
                return new string[] { ManagerRoleName };
            }
            else if (string.Compare(username,
                SampleMembershipProvider.EmployeeUserName, true) == 0)
            {
                return new string[] { EmployeeRoleName };
            }
            else
            {
                return new string[0];
            }
        }

        public override bool IsUserInRole(string username, string roleName)
        {
            string[] roles = GetRolesForUser(username);

```

```

foreach (var role in roles)
{
    if (string.Compare(role, roleName, true) == 0)
    {
        return true;
    }
}
return false;
}
}

```

代码段 AppServices/SampleRoleProvider.cs

身份验证服务必须在 Web.Config 文件中配置。在产品系统上,从安全性角度来看,最好用包含应用程序服务的服务器配置 SSL:



可从
wrox.com
下载源代码

```

<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true" requireSSL="false"/>
      <roleService enabled="true"/>
    </webServices>
  </scripting>
</system.web.extensions>

```

代码段 AppServices/web.config

在<system.web>部分中, membership 和 rolemanager 元素必须配置为引用实现了成员和角色提供程序的类:

```

<system.web>
  <membership defaultProvider="SampleMembershipProvider">
    <providers>
      <add name="SampleMembershipProvider"
        type="Wrox.ProCSharp.Security.SampleMembershipProvider"/>
    </providers>
  </membership>
  <roleManager enabled="true" defaultProvider="SampleRoleProvider">
    <providers>
      <add name="SampleRoleProvider"
        type="Wrox.ProCSharp.Security.SampleRoleProvider"/>
    </providers>
  </roleManager>

```

在调试时,可以用项目属性的 Web 选项卡指定端口号和虚拟路径。示例应用程序使用端口 55555 和虚拟路径/AppServices。如果使用其他值,就需要修改客户端应用程序的配置。

现在从客户端应用程序使用应用程序服务。

2. 客户端应用程序

通过客户端应用程序使用 WPF。Visual Studio 有一个项目设置 Services, 它允许使用客户端应用程序服务。这里可以设置 Forms 身份验证, 把身份验证和角色服务的位置设置为前面定义的地址 http://localhost:55555/AppServices。从项目配置中, 所做的全部工作就是引用 System.Web 和

System.Web.Extensions 程序集, 并修改应用程序的配置文件, 以配置那些使用 ClientAuthenticationMembershipProvider 和 ClientRoleProvider 类的成员提供程序和角色提供程序, 以及这些提供程序使用的 Web 服务的地址。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf - 8"?>
<configuration>
  <system.web>
    <membership defaultProvider="ClientAuthenticationMembershipProvider">
      <providers>
        <add name="ClientAuthenticationMembershipProvider"
          type="System.Web.ClientServices.Providers.
            ClientFormsAuthenticationMembershipProvider,
            System.Web.Extensions, Version=4.0.0.0, Culture=neutral,
            PublicKeyToken=31bf3856ad364e35" serviceUri=
            "http://localhost:55555/AppServices/Authentication_JSON_AppService.axd" />
      </providers>
    </membership>
    <roleManager defaultProvider="ClientRoleProvider" enabled="true">
      <providers>
        <add name="ClientRoleProvider"
          type="System.Web.ClientServices.Providers.ClientRoleProvider,
            System.Web.Extensions, Version=4.0.0.0, Culture=neutral,
            PublicKeyToken=31bf3856ad364e35" serviceUri=
            "http://localhost:55555/AppServices/Role_JSON_AppService.axd"
            cacheTimeout="86400" />
      </providers>
    </roleManager>
  </system.web>
</configuration>
```

代码段 AuthenticationServices/app.config

Windows 应用程序仅使用标签、文本框、密码框和按钮控件, 如图 21-1 所示。只有登录成功, 才会显示标签的内容 User Validated。



图 21-1

Button.Click 事件的处理程序调用 Membership 类的 ValidateUser()方法。因为配置了提供程序 ClientAuthenticationMembershipProvider, 所以该提供程序首先调用 Web 服务, 接着调用

`SampleMembershipProvider` 类的 `ValidateUser()` 方法, 验证登录是否成功。如果成功, 标签 `labelValidatedInfo` 就可见; 否则弹出一个消息框:



可从
wrox.com
下载源代码

```
private void OnLogin(object sender, RoutedEventArgs e)
{
    try
    {
        if (Membership.ValidateUser(textUsername.Text,
            textPassword.Password))
        {
            // user validated!
            labelValidatedInfo.Visibility = Visibility.Visible;
        }
        else
        {
            MessageBox.Show("Username or password not valid",
                "Client Authentication Services", MessageBoxButton.OK,
                MessageBoxImage.Warning);
        }
    }
    catch (WebException ex)
    {
        MessageBox.Show(ex.Message, "Client Application Services",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

代码段 `AuthenticationServices/MainWindow.xaml.cs`

21.2 加密

机密数据应得到保护, 从而使未授权的用户不能读取它们。这对于在网络中发送的数据或存储的数据都有效。可以用对称或不对称密钥来加密这些数据。

通过对称密钥, 可以使用同一个密钥进行加密和解密。与不对称的加密相比, 加密和解密使用不同的密钥: 公钥/私钥。如果使用一个公钥进行加密, 就应使用对应的私钥进行解密, 而不是使用公钥解密。同样, 如果使用一个私钥加密, 就应使用对应的公钥解密, 而不是使用私钥解密。

公钥/私钥总是成对创建。公钥可以由任何人使用, 它甚至可以放在 Web 站点上, 但私钥必须安全地加锁。为了说明加密过程, 下面看看使用公钥和私钥的例子。

如果 Alice 给 Bob 发了一封电子邮件, 如图 21-2 所示, 并且 Alice 希望能保证除了 Bob 外, 其他人都不能阅读该邮件, 所以她就使用 Bob 的公钥。邮件是使用 Bob 的公钥加密的。Bob 打开该邮件, 并使用他秘密存储的私钥解密。

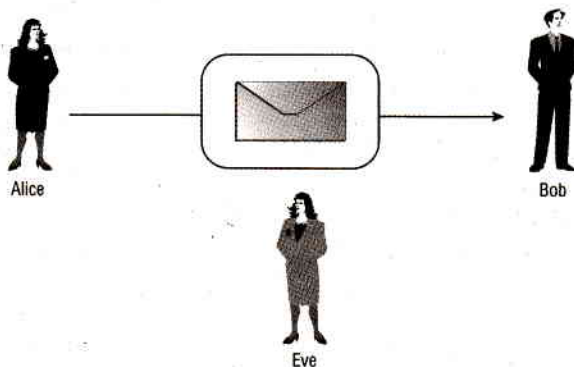


图 21-2

这种方式可以保证除了 Bob 外，其他人都不能阅读 Alice 的邮件。

但这还有一个问题：Bob 不能确保邮件是 Alice 发送来的。Eve 可以使用 Bob 的公钥加密发送给 Bob 的邮件并假装是 Alice。我们使用公钥/私钥把这条规则扩展一下。下面再次从 Alice 给 Bob 发送电子邮件开始。在 Alice 使用 Bob 的公钥加密邮件之前，她添加了自己的签名，再使用自己的私钥加密该签名。然后使用 Bob 的公钥加密邮件。这样就保证了除 Bob 外，其他人都不能阅读该邮件。在 Bob 解密邮件时，他检测到一个加密的签名。这个签名可以使用 Alice 的公钥来解密。而 Bob 可以访问 Alice 的公钥，因为这个密钥是公钥。在解密了签名后，Bob 就可以确定是 Alice 发送了电子邮件。

使用对称密钥的加密和解密算法比使用非对称密钥的算法快得多。对称密钥的问题是密钥必须以安全的方式互换。在网络通信中，一种方式是先使用非对称的密钥进行密钥互换，再使用对称密钥加密通过网络发送的数据。

在 .NET Framework 中，可以使用 System.Security.Cryptography 名称空间中的类来加密。它实现了几个对称算法和非对称算法。有几个不同的算法类用于不同的目的。.NET 3.5 中的一些新类以 Cng 作为前缀或后缀。Cng 是 Cryptography Next Generation 的简称，它用于 Windows Vista 和 Windows Server 2008 以及以后的版本。这个 API 可以使用基于提供程序的模型，编写独立于算法的程序。如果也面向 Windows Server 2003，就需要注意使用了什么加密类。

表 21-1 列出了 System.Security.Cryptography 名称空间中的加密类及其功能。没有 Cng、Managed 或 CryptoServiceProvider 后缀的类是抽象基类，如 MD5。Managed 后缀表示这个算法用托管代码实现，其他类可能封装了本地 Windows API 调用。CryptoServiceProvider 后缀用于实现了抽象基类的类，Cng 后缀用于利用新 Cryptography CNG API 的类。

表 21-1

类 别	类	说 明
散列	MD5、MD5Cng	散列算法的目标是从任意长度的二进制字符串中创建一个长度固定的散列值。这些算法和数字签名一起用于保证数据的完整性。如果再次散列相同的二进制字符串，会返回相同的散列结果。MD5(Message Digest Algorithm 5, 消息摘要算法 5)由 RSA 实验室开发，比 SHA1 快。SHA1 在抵御暴力攻击方面比较强大。SHA 算法由美国国家安全局(NSA)设计。MD5 使用 128 位的散列长度，SHA1 使用 160 位。其他 SHA 算法在其名称中包含了散列长度。SHA512 是这些算法中最强大的，其散列长度为 512 位，它也是最慢的
	SHA1、SHA1Managed、	
	SHA1Cng	
	SHA256、SHA256Managed、	
	SHA256Cng	
	SHA384、SHA384Managed、	
	SHA384Cng	
对称	SHA512、SHA512Managed、	对称密钥算法使用相同的密钥进行数据的加密和解密。现在认为 DES(Data Encryption Standard, 数据加密标准)是不安全的，因为它只使用 56 位的密钥长度，可以在不超过 24 小时的时间内破解。Triple-DES 是 DES 的继承者，其密钥长度是 168 位，但它提供的有效安全性只有 112 位。AES(Advanced Encryption Standard, 高级加密标准)的密钥长度是 128、192 或 256 位。Rijandel 非常类似于 AES，它只是在密钥长度方面的选项较多。AES 是美国政府采用的加密标准
	SHA512Cng	
	DES、DESCryptoServiceProvider	
	TripleDES、	
	TripleDESCryptoServiceProvider	
	Aes, AesCryptoServiceProvider、	
	AesManaged	
	RC2、RC2CryptoServiceProvider	
	Rijandel、RijandelManaged	

(续表)

类 别	类	说 明
非对称	DSA、DSACryptoServiceProvider ECDsa、ECDsaCng ECDiffieHellman、 ECDiffieHellmanCng RSA、RSACryptoServiceProvider	非对称算法使用不同的密钥进行加密和解密。RSA(Rivest, Shamir, Adleman)是第一个用于签名和加密的算法。这个算法广泛用于电子商务协议。DSA(Digital Signature Algorithm, 数字签名算法)是用于数字签名的一个美国联邦政府标准。ECDSA(Elliptic Curve DSA, 椭圆曲线数字签名算法)和 ECDiffieHellman 使用基于椭圆曲线组的算法。这些算法比较安全, 且使用较短的密钥长度。例如, DSA 的密钥长度为 1024 位, 其安全性类似于 ECDSA 的 160 位。因此, ECDSA 比较快。ECDiffieHellman 算法用于以安全的方式在公共信道中交换私钥

下面用例子说明这些算法如何通过编程使用。

21.2.1 签名

第一个例子说明了如何使用 ECDSA 算法进行签名。Alice 创建了一个签名, 它用 Alice 的私钥加密, 可以使用 Alice 的公钥访问。因此保证该签名来自于 Alice。

首先, 看看 Main()方法中的主要步骤: 创建 Alice 的密钥, 给字符串“Alice”签名, 最后使用公钥验证该签名是否真的来自于 Alice。要签名的消息使用 Encoding 类转换为一个字节数组。要把加密的签名写入控制台, 包含该签名的字节数组应使用 Convert.ToBase64 String()方法转换为一个字符串。



千万不要使用 Encoding 类把加密的数据转换为字符串。Encoding 类验证和转换 Unicode 不允许使用的无效值, 因此把字符串转换回字节数组会得到另一个结果。



可从
wrox.com
下载源代码

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        internal static CngKey aliceKeySignature;
        internal static byte[] alicePubKeyBlob;

        static void Main()
        {
            CreateKeys();

            byte[] aliceData = Encoding.UTF8.GetBytes("Alice");
            byte[] aliceSignature = CreateSignature(aliceData, aliceKeySignature);
            Console.WriteLine("Alice created signature: {0}",
                Convert.ToBase64String(aliceSignature));
            if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))
            {
                Console.WriteLine("Alice signature verified successfully");
            }
        }
    }
}
```

CreateKeys()方法为 Alice 创建新的密钥对。因为这个密钥对存储在一个静态字段中，所以可以从其他方法中访问它。CngKey 类的 Create()方法把该算法作为一个参数，为算法定义密钥对。通过 Export()方法，导出密钥对中的公钥。这个公钥可以提供给 Bob，来验证签名。Alice 保留其私钥。除了使用 CngKey 类创建密钥对之外，还可以打开存储在密钥存储器中的已有密钥。通常 Alice 在其私有存储器中有一个证书，其中包含了一个密钥对，该存储器可以用 CngKey.Open()方法访问。

```
static void CreateKeys()
{
    aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP256);
    alicePubKeyBlob = aliceKeySignature.Export(CngKeyBlobFormat.GenericPublicBlob);
}
```

有了密钥对，Alice 就可以使用 ECDsaCng 类创建签名了。这个类的构造函数从 Alice 那里接收包含公钥和私钥的 CngKey 类。再使用私钥，通过 SignData()方法给数据签名。

```
static byte[] CreateSignature(byte[] data, CngKey key)
{
    var signingAlg = new ECDsaCng(key);
    byte[] signature = signingAlg.SignData(data);
    signingAlg.Clear();

    return signature;
}
```

要验证签名是否真的来自于 Alice，Bob 使用 Alice 的公钥检查签名。包含公钥 blob 的字节数组可以用静态方法 Import()导入 CngKey 对象。然后使用 ECDsaCng 类，调用 VerifyData()方法来验证签名。

```
static bool VerifySignature(byte[] data, byte[] signature, byte[] pubKey)
{
    bool retValue = false;
    using (CngKey key = CngKey.Import(pubKey, CngKeyBlobFormat.GenericPublicBlob))
    {
        var signingAlg = new ECDsaCng(key);
        retValue = signingAlg.VerifyData(data, signature);
        signingAlg.Clear();
    }
    return retValue;
}
```

21.2.2 交换密钥和安全传输

下面是一个比较复杂的例子，它使用 Diffie Hellman 算法交换一个对称密钥，以进行安全的传输。在 Main()方法中，可以看到其主要功能。Alice 创建了一条加密的消息，并把它发送给 Bob。在此之前，要先为 Alice 和 Bob 创建密钥对。Bob 只能访问 Alice 的公钥，Alice 也只能访问 Bob 的公钥。



```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
```

```
namespace Wrox.ProCSharp.Security
```

```
{
```

```
    class Program
```

```
    {
```

```
        static CngKey aliceKey;
```

```
        static CngKey bobKey;
```

```
        static byte[] alicePubKeyBlob;
```

```
        static byte[] bobPubKeyBlob;
```

```
        static void Main()
```

```
        {
```

```
            CreateKeys();
```

```
            byte[] encryptedData = AliceSendsData("secret message");
```

```
            BobReceivesData(encryptedData);
```

```
        }
```

代码段 SecureTransfer/Program.cs

在 `CreateKeys()` 方法的实现代码中, 使用 EC Diffie Hellman 256 算法创建密钥。

```
private static void CreateKeys()
```

```
{
```

```
    aliceKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
```

```
    bobKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
```

```
    alicePubKeyBlob = aliceKey.Export(CngKeyBlobFormat.EccPublicBlob);
```

```
    bobPubKeyBlob = bobKey.Export(CngKeyBlobFormat.EccPublicBlob);
```

```
}
```

在 `AliceSendsData()` 方法中, 包含文本字符的字符串使用 `Encoding` 类转换为一个字节数组。创建一个 `ECDiffieHellmanCng` 对象, 用 Alice 的密钥对初始化它。Alice 调用 `DeriveKeyMaterial()` 方法, 从而使用其密钥对和 Bob 的公钥创建一个对称密钥。返回的对称密钥使用对称算法 AES 加密数据。`AesCryptoServiceProvider` 需要密钥和一个初始化矢量(IV)。IV 从 `GenerateIV()` 方法中动态生成, 对称密钥用 EC Diffie Hellman 算法交换, 但还必须交换 IV。从安全性角度来看, 在网络上传输未加密的 IV 是可行的——只是密钥交换必须是安全的。IV 存储为内存流中的第一项内容, 其后是加密的数据, 其中, `CryptoStream` 类使用 `AesCryptoServiceProvider` 类创建的 `encryptor`。在访问内存流中的加密数据之前, 必须关闭加密流。否则, 加密数据就会丢失最后的位。

```
private static byte[] AliceSendsData(string message)
```

```
{
```

```
    Console.WriteLine("Alice sends message: {0}", message);
```

```
    byte[] rawData = Encoding.UTF8.GetBytes(message);
```

```
    byte[] encryptedData = null;
```

```
    using (var aliceAlgorithm = new ECDiffieHellmanCng(aliceKey))
```

```
    using (CngKey bobPubKey = CngKey.Import(bobPubKeyBlob,
```

```
        CngKeyBlobFormat.EccPublicBlob))
```

```
    {
```

```
        byte[] symmKey = aliceAlgorithm.DeriveKeyMaterial(bobPubKey);
```

```
        Console.WriteLine("Alice creates this symmetric key with " +
```

```

        "Bobs public key information: {0}",
        Convert.ToBase64String(symmKey));

var aes = new AesCryptoServiceProvider();
aes.Key = symmKey;
aes.GenerateIV();
using (ICryptoTransform encryptor = aes.CreateEncryptor())
using (MemoryStream ms = new MemoryStream())
{
    // create CryptoStream and encrypt data to send
    var cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write);

    // write initialization vector not encrypted
    ms.Write(aes.IV, 0, aes.IV.Length);
    cs.Write(rawData, 0, rawData.Length);
    cs.Close();
    encryptedData = ms.ToArray();
}
aes.Clear();
}
Console.WriteLine("Alice: message is encrypted: {0}",
    Convert.ToBase64String(encryptedData));
Console.WriteLine();
return encryptedData;
}

```

Bob 从 `BobReceivesData()` 方法的参数中接收加密数据。首先，必须读取未加密的初始化矢量。`AesCryptoServiceProvider` 类的 `BlockSize` 属性返回块的位数。位数除以 8，就可以计算出字节数。最快的方式是把数据右移 3 位。右移 1 位就是除以 2，右移 2 位就是除以 4，右移 3 位就是除以 8。在 `for` 循环中，包含未加密 IV 的原字节的前几个字节写入数组 `iv` 中。接着用 Bob 的密钥对实例化一个 `ECDiffieHellmanCng` 对象。使用 Alice 的公钥，从 `DeriveKeyMaterial()` 方法中返回对称密钥。比较 Alice 和 Bob 创建的对称密钥，可以看出所创建的密钥值相同。使用这个对称密钥和初始化矢量，来自 Alice 的消息就可以用 `AesCryptoServiceProvider` 类解密。

```

private static void BobReceivesData(byte[] encryptedData)
{
    Console.WriteLine("Bob receives encrypted data");
    byte[] rawData = null;

    var aes = new AesCryptoServiceProvider();

    int nBytes = aes.BlockSize / 8;
    byte[] iv = new byte[nBytes];
    for (int i = 0; i < iv.Length; i++)
        iv[i] = encryptedData[i];

    using (var bobAlgorithm = new ECDiffieHellmanCng(bobKey))
    using (CngKey alicePubKey = CngKey.Import(alicePubKeyBlob,
        CngKeyBlobFormat.EccPublicBlob))
    {
        byte[] symmKey = bobAlgorithm.DeriveKeyMaterial(alicePubKey);
        Console.WriteLine("Bob creates this symmetric key with " +
            "Alices public key information: {0}",
            Convert.ToBase64String(symmKey));
    }
}

```