

语言对比：COM DLL注册

和微软以前的COM文件不同，CLI程序集几乎不需要任何类型的注册，只需将组成一个程序的所有文件复制到一个特定的目录，然后执行程序，即可完成部署。

21.7 公共中间语言

公共语言基础结构（CLI）这个名称揭示了CIL和CLI的一个重要特点：支持多种语言在同一个应用程序内的交互（而不是源代码跨越不同操作系统的可移植性）。所以，CIL不只是C#的中间语言，还是其他许多编程语言的中间语言，比如Visual Basic .NET、Java风格的J#、Smalltalk、C++等（写作本书时，有20多种，包括各种版本的COBOL和FORTRAN）。编译成CIL的语言称为源语言（source language），而且各自都有一个自定义的编译器能将源语言转换为CIL。编译为CIL后，当初使用的是何种源语言便无关紧要了。这个强大的功能使不同的开发小组能跨越不同的单位进行协作式开发，而不必关心每个小组使用的是何种语言。这样一来，CIL就实现了语言之间的互操作性以及平台可移植性。

■ 注意

CLI的一个强大功能是支持多种语言。这就允许使用多种语言来编写一个程序，并允许用一种语言写的代码访问用另一种语言写的库。

21.8 公共类型系统

不管编程语言如何，最终生成的程序在内部都要操作数据类型。因此，CLI还包含了公共类型系统（Common Type System, CTS）。CTS定义了类型的结构及其在内存中的布局，另外还规定了与类型有关的概念和行为。除了与类型中存储的数据有关的信息，CTS还包含了类型的操作指令。由于CTS的目标是实现语言间的互操作性，所以它规定了类型在语言的外部边界处的表现及行为。“运行时”要在执行时强化CTS建立的各种规定。

在CTS内部，类型分为以下两类。

- **值（Value）**是用于表示基本类型（比如整数和字符）以及以结构的形式提供的更复杂数据的位模式（bit pattern）。每一种值类型都对应于一个单独的类型定义，这个单独的类型定义不存储在位本身中。单独的类型定义是指提供了值中每一位的含义，并对值所支持的操作进行了说明的类型定义。
- **对象（Object）**则在其本身中包含了对象的类型定义（这有助于实现类型检查）。每个对象实例都有唯一性标识。此外，对象提供了用于存储其他类型（值或者对象引用）的位置，这些位置称为槽（slot）。与值类型不同的是，更改这些槽中的内容，并不会改变对象的标识。

以上两个类别直接对应于对每种类型进行声明的C#语法。

21.9 公共语言规范

由于CTS提供了语言集成的优势，所以大多数源语言都支持CTS（对源语言的解释参见上一节）。然而，CTS语言相容性规范还存在着一个子集，称为公共语言规范（Common Language Specification, CLS）。后者侧重于库的实现。它面向的是库开发人员，为他们提供编写库的标准，以使这些库能从大多数源语言中访问——无论使用库的源语言是否相容于CTS。之所以称为公共语言规范，是因为它的另一个目的是鼓励CLI语言提供一种方式来创建可供互操作的库——或者说能从其他语言访问的库。

例如，虽然一种语言提供对无符号整数（unsigned integer）的支持是完全合理的，但这样的类型并未包含在CLS中。因此，一个类库的开发者不应对外公开无符号整数。否则，在不支持无符号整数的、与CLS规范相容的源语言中，开发者就会不愿意选用这样的库。因此，理想情况下，一个库要想能从多种语言访问，就必须遵守CLS规范。注意，CLS并不关心那些没有对外公开给程序集的类型。

注意，创建一个非CLS相容的API时，可以让编译器报告一条警告消息。为此，请使用System.CLSCompliant这个程序集特性，并为参数指定true值。

21.10 基类库

除了提供用于执行CIL代码的平台，CLI还定义了程序可能会用到的一个核心的类库集合，称为基类库（Base Class Library, BCL）。这些库提供了基本的类型和API，使程序能以一致的方式与“运行时”和底层操作系统进行交互。BCL还支持集合、简单文件访问、安全性、基本数据类型（string等）、流以及其他。

类似地，还有一个微软特有的库，称为Framework类库（Framework Class Library, FCL），它支持富客户端用户界面、Web用户界面、数据库访问和分布式通信等。

21.11 元数据

除了执行指令，CIL代码还包含与类型和程序中包含的文件有关的元数据。元数据包含以下内容：

- 程序或类库中每一个类型的描述；
- 清单信息，包括与程序本身有关的数据，以及它所依赖于的库；
- 嵌入代码中的自定义特性，提供与特性所修饰的构造有关的额外信息。

元数据并不是CIL中可有可无的东西。相反，它构成了CLI实现的一个核心部分。它提供了类型的表示和行为信息，并包含一些位置信息，描述了哪个程序集包含哪个特定的类型定义。为了保存来自编译器的数据，并使这些数据可以在运行时由调试器和“运行时”访问，它扮演了一个关键性的角色。这些数据不仅可在CIL代码中使用，还能在机器码执行期间访问，确保“运行时”能够继续执行任何必要的类型检查。

元数据为运行时提供了一种机制，可以处理本机代码和托管代码混合执行的情况。同时，它还加强了代码和代码执行的可靠性，因为它能使一个库从一个版本顺利迁移到下一个版本，用加载时的实现取代了编译时定义的绑定。

元数据的一个特殊部分是清单，它包含了与一个库及其依赖性有关的所有标头信息。

所以，元数据的清单部分使开发人员可以判断一个模块的依赖文件，其中包括与依赖文件特定版本和模块创建者签名相关的信息。在执行时，“运行时”使用清单来确定要载入哪些依赖库，库或主要程序是否被篡改以及是否丢失了程序集。

元数据还包含自定义特性，这些特性可以对代码进行额外的修饰。特性提供了与可由程序在执行时访问的CIL指令有关的额外的元数据。

元数据在执行时通过反射机制来使用。利用反射机制，我们可以在执行期间查找一个类型或者它的成员，然后调用该成员，或者判断一个特定的构造是否使用了一个特性进行修饰。这样就实现了晚期绑定——换言之，可以在执行时（而不是编译时）决定要执行的代码。反射机制甚至还用于生成文档，具体做法是遍历元数据，并将其复制到某种形式的帮助文档中（详情参见第17章）。

21.12 小结

本章介绍了许多新术语和缩写词，它们对于理解C#程序的运行环境具有重要意义。许多由3个字母组成的缩写词比较容易混淆。表21-2简单总结了CLI的部分术语和缩写词。

表21-2 常见的C#相关缩写词

缩写	定义	说明
.NET	无	这是微软所实现的CLI，其中包括CLR、CIL以及各种语言——全部都相容于CLS
BCL	基类库	CLI规范的一部分，定义了集合、线程处理、控制台以及用于生成几乎所有程序所需的其他基类
C#	无	一种编程语言。注意，C#语言规范独立于CLI标准，也得到了ECMA和ISO标准组织的认可
CIL (IL)	公共中间语言	CLI规范中的一种语言，为可在CLI的实现上执行的代码定义了指令。有时也称为中间语言（IL）或Microsoft IL（MSIL），以区别于其他中间语言。为了强调此标准的适用范围不仅是微软的产品，平时应该多说CIL，而不是说MSIL（或者IL）
CLI	公共语言基础结构	这个规范定义了中间语言、基类和行为特征，允许实现人员创建虚拟执行系统和编译器，确保不同的源语言能在公共执行环境的顶部进行互操作
CLR	公共语言运行时	微软根据CLI规定的定义实现的“运行时”
CLS	公共语言规范	CLI规范的一部分，定义了源语言必须支持的核心特性。只有支持这些特性，才能在根据CLI规范而实现的“运行时”中执行
CTS	公共类型系统	一般要由CLI兼容语言来执行的一个标准，它定义了编程语言向模块外部公开的类型的表示及行为。它包含如何对类型进行合并以构成新类型的一些概念
FCL	.NET Framework 类库	用于构成Microsoft .NET Framework的类库，包含微软实现的BCL以及用于Web开发、分布式通信、数据库访问、富客户端用户界面开发等的一个大型类库
WinRT	Windows Runtime	基于Windows 8的平台，用于运行Metro风格的应用程序。WinRT包含WinAPI，这是一套全新的、增强的Windows 8操作系统API。由于包含.NET风格的元数据，所以能从C#中无疑地访问这种API
VES（“运行时”）	虚拟执行系统	作为代理负责管理为CLI编译的程序的执行

附录A

下载和安装C#编译器与CLI平台

为了编译和运行C#程序，需要安装编译器和CLI平台。

A.1 Microsoft .NET

最主流的CLI平台就是Microsoft .NET，这是在Microsoft Windows上进行开发的首选平台。

- 最小安装（其中包括支持C# 2.0语法的编译器和.NET Framework）是.NET Framework 2.0或更高版本的可再发行包，下载地址是：<http://msdn.microsoft.com/net>。
- 要使用功能丰富的IDE，其中包括“智能感知”技术以及对项目文件的支持，请安装某个版本的Visual Studio IDE。免费和试用版的下载地址是：<http://www.microsoft.com/visualstudio/chs/downloads>。

为了在命令行上进行编译，无论当前已安装Visual Studio，还是只安装了“运行时”，必须设置PATH环境变量来包含C#编译器CSC.EXE。

为Microsoft .NET设置编译器路径

如果当前已在计算机上安装了Visual Studio 2012，请选择“开始”→“所有程序”→“Microsoft Visual Studio 2012”→“Visual Studio Tools”→“VS2012开发人员命令提示”，从而打开命令提示窗口。这会自动将CSC.EXE放到路径中，以便能从任何目录执行。

如果没有安装Visual Studio 2012，“开始”菜单不会出现任何特殊的编译器命令提示。相反，需要显式引用完整编译器路径名，或自行将其添加到路径中。编译器的位置是%Windir%\Microsoft.NET\Framework\<版本>。其中，<版本>是当前在计算机上安装的.NET Framework的版本号（v1.0.3705、v1.1.4322、v2.0.50727、v4.0.30319等），而%Windir%是指向本机Windows安装目录的特殊环境变量。为了将这个位置添加到路径

中,请使用以下命令: `Set PATH= %PATH%;%Windir%\Microsoft.NET\Framework\<版本>`。同样地,需要恰当地替换<版本>的值。输出A-1展示了一个例子。

输出A-1

```
Set PATH=%PATH%;%Windir%\Microsoft.NET\Framework\v2.0.50727
```

将.NET Framework所在的目录添加到路径中之后,就可以在任何地方直接引用.NET C#编译器CSC.EXE,不必每次都提供完整的路径名。

A.2 Mono

如果要在一个非Microsoft Windows的平台上从事CLI开发,请考虑使用Mono。可以从 www.mono-project.com 下载它。和.NET平台一样,如果C#编译器不在搜索路径中,Mono也要求每次都指定它的完整路径。在Linux上,默认安装路径是 `/usr/lib/mono/<版本>`,编译器是 `gmcs.exe` 或者 `mcs.exe`,具体取决于版本。如果将Mono安装到Microsoft Windows上,默认路径是 `%ProgramFiles%\ Mono-<版本>\lib\mono\<版本>\`。

还可以选择Monoppix作为自己的Linux开发平台,其中自带了Mono。Monoppix基于Knoppix而开发,后者是可从光盘引导的一款Linux版本。Monoppix的下载地址是 www.monoppix.com。

Mono平台上的编译器不是CSC.EXE,而是MCS.EXE或GMCS.EXE,具体要取决于编译器的版本。输出A-2展示了对HelloWorld.cs进行编译的命令。

输出A-2

```
C:\SAMPLES>mcs.exe HelloWorld.cs
```

遗憾的是,不能在Linux环境上直接运行生成的二进制程序。相反,它要求使用 `mono.exe` 来显式执行“运行时”,如输出A-3所示。

输出A-3

```
C:\SAMPLES>mono.exe HelloWorld.exe  
Hello. My name is Inigo Montoya.
```

附录B

井字棋源代码清单

代码清单B-1 井字棋程序 (Tic-Tac-Toe)

```
#define CSARP2

using System;

#pragma warning disable 1030 // Disable user-defined warnings

// The TicTacToe class enables two players to
// play tic-tac-toe.
class TicTacToeGame // Declares the TicTacToeGame class
{
    static void Main() // Declares the entry point to the program
    {
        // Stores locations each player has moved.
        int[] playerPositions = { 0, 0 };

        // Initially set the currentPlayer to Player 1;
        int currentPlayer = 1;

        // Winning player
        int winner = 0;

        string input = null;

        // Display the board and prompt the current player
        // for his next move.
        for (int turn = 1; turn <= 10; ++turn)
        {
            DisplayBoard(playerPositions);
```

```

        #region Check for End Game
        if (EndGame(winner, turn, input))
        {
            break;
        }
        #endregion Check for End Game

        input = NextMove(playerPositions, currentPlayer);

        winner = DetermineWinner(playerPositions);

        // Switch players
        currentPlayer = (currentPlayer == 2) ? 1 : 2;
    }
}

private static string NextMove(int[] playerPositions,
                                int currentPlayer)
{
    string input;

    // Repeatedly prompt the player for a move
    // until a valid move is entered.
    bool validMove;
    do
    {
        // Request a move from the current player.
        System.Console.WriteLine("\nPlayer {0} - Enter move:",
                                currentPlayer);
        input = System.Console.ReadLine();
        validMove = ValidateAndMove(playerPositions,
                                    currentPlayer, input);
    } while (!validMove);

    return input;
}

static bool EndGame(int winner, int turn, string input)
{
    bool endGame = false;
    if (winner > 0)
    {
        System.Console.WriteLine("\nPlayer {0} has won!!!!",
                                winner);
        endGame = true;
    }
    else if (turn == 10)
    {
        // After completing the 10th display of the
        // board, exit out rather than prompting the
        // user again.
        System.Console.WriteLine("\nThe game was a tie!");
        endGame = true;
    }
    else if (input == "" || input == "quit")
    {
        // Check if user quit by hitting Enter without
        // any characters or by typing "quit".
    }
}

```

```

        System.Console.WriteLine("The last player quit");
        endGame = true;
    }
    return endGame;
}

static int DetermineWinner(int[] playerPositions)
{
    int winner = 0;

    // Determine if there is a winner
    int[] winningMasks = {
        7, 56, 448, 73, 146, 292, 84, 273};

    foreach (int mask in winningMasks)
    {
        if ((mask & playerPositions[0]) == mask)
        {
            winner = 1;
            break;
        }
        else if ((mask & playerPositions[1]) == mask)
        {
            winner = 2;
            break;
        }
    }
    return winner;
}

static bool ValidateAndMove(
    int[] playerPositions, int currentPlayer, string input)
{
    bool valid = false;

    // Check the current player's input.
    switch (input)
    {
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9":
#warning "Same move allowed multiple times."
            int shifter; // The number of places to shift
                        // over in order to set a bit.
            int position; // The bit which is to be set.

            // int.Parse() converts "input" to an integer.
            // "int.Parse(input) - 1" because arrays
            // are zero-based.
            shifter = int.Parse(input) - 1;

            // Shift mask of 00000000000000000000000000000001
            // over by celLocations.
            position = 1 << shifter;

```

```

        // Take the current player cells and OR them
        // to set the new position as well.
        // Since currentPlayer is either 1 or 2 you
        // subtract one to use currentPlayer as an
        // index in a 0-based array.
        playerPositions[currentPlayer - 1] |= position;

        valid = true;
        break;

    case "":
    case "quit":
        valid = true;
        break;

    default:
        // If none of the other case statements
        // is encountered, then the text is invalid.
        System.Console.WriteLine(
            "\nERROR: Enter a value from 1-9. "
            + "Push ENTER to quit");
        break;
    }

    return valid;
}

static void DisplayBoard(int[] playerPositions)
{
    // This represents the borders between each cell
    // for one row.
    string[] borders = {
        "|", "|", "\n---+---+---\n", "|", "|",
        "\n---+---+---\n", "|", "|", ""
    };

    // Display the current board;
    int border = 0; // set the first border (border[0] = "|")

#if CSHARP2
    System.Console.Clear();
#endif

    for (int position = 1;
        position <= 256;
        position <= 1, border++)
    {
        char token = CalculateToken(
            playerPositions, position);

        // Write out a cell value and the border that
        // comes after it.
        System.Console.Write(" {0} {1}",
            token, borders[border]);
    }
}

static char CalculateToken(
    int[] playerPositions, int position)

```

```
{
    // Initialize the players to 'X' and 'O'
    char[] players = {'X', 'O'};

    char token;
    // If player has the position set,
    // then set the token to that player.
    if ((position & playerPositions[0]) == position)
    {
        // Player 1 has that position marked
        token = players[0];
    }
    else if ((position & playerPositions[1]) == position)
    {
        // Player 2 has that position marked
        token = players[1];
    }
    else
    {
        // The position is empty.
        token = ' ';
    }
    return token;
}

#line 113 "TicTacToe.cs"
// Generated code goes here
#line default
}
```

■ 附录C ■

TPL和C# 5.0之前的多线程模式

第18章讲过，多线程编程模式的目的是降低多线程编程的复杂性、监视异步操作、运行线程池、防止死锁以及实现跨操作和数据访问时的原子性/同步。在.NET 4.5和C# 5.0问世之前的十年时间里，总共出现了.NET Framework的6个版本和C#语言的4个版本，还出现了数量差不多的多线程编程模式。在此期间，人们对多线程编程进行了数不清的改进，而且经常是对框架甚至是对语言的改进。在这个过程中，老版本采用的模式逐渐不再占优。但无论占不占优，C#开发人员都有可能遇到这些模式。这要么是因为需要为.NET 4.5/C# 5.0之前的环境编程，要么是因为要使用来自其他框架的API，而这个API仍在采用老的模式。

本附录将讨论这些模式。如果当前在用最新版本的.NET和C#开发，请将本附录看成是“高级主题”，通过阅读它来熟悉过去多线程编程的细节。但是，如果当前用不了TPL和TAP及其async/await关键字，请将本附录的主题视为多线程API的重要部分。最重要的是，本附录解释了如何使用TPL和C# 5.0与过去的模式进行有效的交互。

出于演示的目的，所有例子都去掉了异常处理部分。

C.1 异步编程模型

在TPL之前，人们建立的一个特别著名的模式就是“异步编程模型”（Asynchronous Programming Model, APM）。给定一个要长时间运行的同步方法X()，APM使用一个BeginX()方法，以异步方式启动X()；并用一个EndX()方法结束它。（后文将这些方法命名为X、BeginX和EndX。）

C.1.1 调用APM

代码清单C-1演示了用System.Net.WebRequest类下载一个网页的模式。其功能与18.5

节所描述的相同。但是，假定这一次无法使用TPL和TAP，而是只能使用APM模式。为了向后兼容TPL相关异步方法还没有出现之前的模式，WebRequest还通过BeginGetResponse() (BeginX)和EndGetResponse() (EndX)方法来支持APM模式。这些方法是同步方法GetResponse() (X)的异步版本。

代码清单C-1 为WebRequest使用APM模式

```
using System;
using System.IO;
using System.Net;
using System.Linq;

public class Program
{
    public static void Main(string[] args)
    {
        string url = "http://www.IntelliTect.com";
        if (args.Length > 0)
        {
            url = args[0];
        }

        Console.Write(url);
        WebRequest webRequest =
            WebRequest.Create(url);

        IAsyncResult asyncResult =
            webRequest.BeginGetResponse(null, null);

        // Indicate busy using dots Ideally (at least in a non-Console
// implementation) should use a callback rather than a wait.
        while (
            !asyncResult.AsyncWaitHandle.WaitOne(100))
        {
            Console.Write('.');
        }

        // Retrieve the results when finished downloading
        WebResponse response =
            webRequest.EndGetResponse(asyncResult);
        using (StreamReader reader =
            new StreamReader(response.GetResponseStream()))
        {
            // Note: ReadToEnd() is blocking. A production implementation
// should offload this to another thread.
            int length = reader.ReadToEnd().Length;
            Console.WriteLine(FormatBytes(length));
        }
    }

    static public string FormatBytes(long bytes)
    {
        string[] magnitudes =
            new string[] { "GB", "MB", "KB", "Bytes" };
        long max =
            (long)Math.Pow(1024, magnitudes.Length);

        return string.Format("{1:##.##} {0}",
            magnitudes.FirstOrDefault(

```

```

        magnitude =>
            bytes > (max /= 1024) )?? "0 Bytes",
        (decimal)bytes / (decimal)max).Trim();
    }
}

```

代码清单C-1的结果如输出C-1所示。

输出C-1

```
http://www.IntelliTect.com.....29.36 KB
```

如前所述，APM的关键在于成对使用的BeginX和EndX方法，而且这些方法具有完善的签名。BeginX返回一个System.IAsyncResult对象，可通过它访问异步调用的状态，以便等待完成或轮询完成。然后，EndX方法获取这个返回的对象作为输入参数。这样才真正将两个方法配成一对，从而可以清晰地判断哪个BeginX方法调用和哪个EndX方法调用配对。APM的本质要求所有BeginX调用都必须有一个（而且只能有一个）EndX调用。因此，不可能发生两个EndX调用接受同一个IAsyncResult实例的情况。

代码清单C-1还使用了IAsyncResult的WaitHandle判断异步方法何时结束。反复轮询WaitHandle时，程序向控制台打印句点符号，指明当前正在下载。然后，程序调用EndGetResponse()。

EndX方法具有4个方面的用途。首先，调用EndX会阻塞线程继续执行，直到请求的工作成功完成（或者发生错误并引发异常）。其次，如果方法X要返回数据，这个数据可从EndX方法调用中访问。第三，如果执行请求的工作时发生异常，可在调用EndX时重新引发这个异常，确保异常会被调用代码发现——好像它是在一次同步调用中发生的一样。最后，如果任何资源需要在调用x后清理，EndX将负责清理这些资源。

1. APM签名

BeginX和EndX APM方法合并起来应与签名的同步版本匹配。因此，EndX的返回参数应该与X方法（本例是GetReponse()）的返回参数匹配。除此之外，BeginX方法的输入参数也需要匹配。在WebRequest.GetResponse()的情况下是没有参数的，但让我们考虑一个虚构的同步方法bool TryDoSomething(string url, ref string data, out string[] links)。图C-1展示了从同步方法到APM方法的参数映射。

```

System.IAsyncResult BeginTryDoSomething(
    String url, ref string data, out string[] links,
    System.AsyncCallback callback, object state)

bool TryDoSomething(
    string url, ref string data, out string[] links)

bool EndTryDoSomething(ref string data, out string[] links,
    System.IAsyncResult result);

```

图C-1 APM参数分布情况

所有输入参数都映射到BeginX方法。类似地，返回参数映射到EndX返回参数。还要注意，由于ref和out参数会返回结果，所以它们包含在EndX方法的签名中。与之相反，由于url只是一个输入参数，所以不包含在EndX方法中。

2. 用AsyncCallback实现连续传递样式

BeginX方法有两个额外的参数在同步方法中是没有的。一个是回调参数，是方法结束时要调用的一个System.AsyncCallback委托；另一个是object类型的状态参数（state）。代码清单C-2演示了如何使用它们。（输出与输出C-1相同。）

代码清单C-2 随同回调和状态调用APM

```
using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading;

public class Program
{
    public static void Main(string[] args)
    {
        string url = "http://www.intelliTecture.com";
        if (args.Length > 0)
        {
            url = args[0];
        }

        Console.WriteLine(url);
        WebRequest webRequest = WebRequest.Create(url);
        WebRequestState state =
            new WebRequestState(webRequest);
        IAsyncResult asyncResult =
            webRequest.BeginGetResponse(
                GetResponseAsyncCompleted, state);

        // Indicate busy using dots
        while (
            !asyncResult.AsyncWaitHandle.WaitOne(100))
        {
            Console.WriteLine('.');
        }
        state.ResetEvent.Wait();
    }

    // Retrieve the results when finished downloading
    private static void GetResponseAsyncCompleted(
        IAsyncResult asyncResult)
    {
        WebRequestState completedState =
            (WebRequestState)asyncResult.AsyncState;
        HttpWebResponse response =
            (HttpWebResponse)completedState.WebRequest
                .EndGetResponse(asyncResult);
        Stream stream = response.GetResponseStream();
        StreamReader reader = new StreamReader(stream);
        // Note: ReadToEnd() is blocking. A production implementation
        // should offload this to another thread.
        int length = reader.ReadToEnd().Length;

        Console.WriteLine(FormatBytes(length));
        completedState.ResetEvent.Set();
        completedState.Dispose();
    }
}
```

```

    }
    // ...
}

class WebRequestState : IDisposable
{
    public WebRequestState(WebRequest webRequest)
    {
        WebRequest = webRequest;
    }
    public WebRequest WebRequest { get; private set; }
    private ManualResetEventSlim _ResetEvent =
        new ManualResetEventSlim();
    public ManualResetEventSlim ResetEvent
    { get { return _ResetEvent; } }

    public void Dispose()
    {
        ResetEvent.Dispose();
        GC.SuppressFinalize(this);
    }
}

```

注意，代码清单C-2为BeginGetResponse()的两个参数传递了数据。第一个是System.AsyncCallback类型的委托，它获取System.AsyncResult类型的一个参数。AsyncCallback标识了异步调用结束后要执行的代码。通过登记委托，可实现一种称为连续调用样式（continuation passing style, CPS）的fire-and-forget调用模式，而不必在BeginGetResponse()下面顺序地添加EndGetResponse()和Console.WriteLine()代码。通过CPS，可以“登记”异步方法结束时要执行的代码。注意，仍然需要调用EndGetResponse()，但通过把它放到回调中，可以保证在异步调用结束时，它不会阻塞主线程。

3. 在APM方法之间传递状态

除了AsyncCallback参数，还有state参数，它用于在回调执行时向回调传递额外的数据。代码清单C-2包含一个WebRequestState类，用于向回调传递额外的数据，而且它本身包含了WebRequest，以便使用它调用EndGetResponse()。除了使用WebRequestState类本身，一个替代的方案是使用匿名方法（包括Lambda表达式），同时为额外的数据使用闭包，如代码清单C-3所示。

代码清单C-3 在匿名方法上使用闭包来传递状态

```

using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading;

public class Program
{
    public static void Main(string[] args)
    {
        string url = "http://www.intelliTecture.com";
        if (args.Length > 0)
        {
            url = args[0];
        }
    }
}

```

```

    }

    Console.Write(url);
    WebRequest webRequest = WebRequest.Create(url);
    ManualResetEventSlim resetEvent =
        new ManualResetEventSlim();
    IAsyncResult asyncResult =
        webRequest.BeginGetResponse(
            (completedAsyncResult) =>
            {
                HttpWebResponse response =
                    (HttpWebResponse)webRequest.EndGetResponse(
                        completedAsyncResult);
                Stream stream =
                    response.GetResponseStream();
                StreamReader reader =
                    new StreamReader(stream);
                int length = reader.ReadToEnd().Length;

                Console.WriteLine(FormatBytes(length));
                resetEvent.Set();
                resetEvent.Dispose();
            },
            null);

    // Indicate busy using dots
    while (
        !asyncResult.AsyncWaitHandle.WaitOne(100))
    {
        Console.Write('.');
    }
    resetEvent.Wait();
}

// ...
}

```

无论是否通过闭包传递状态，注意在AsyncCallback完成后，要用一个ManualResetEvent发出信号。这似乎很奇怪，因为IAsyncResult已经包含一个WaitHandle。然而，区别在于，IAsyncResult的WaitHandle是在异步方法完成时，在AsyncCallback执行之前设置的。如果只在IAsyncResult的WaitHandle上阻塞，那么可能在AsyncCallback执行前退出程序。因此，我们使用了一个单独的ManualResetEvent。

4. 资源清理

另一个重要的APM规则是不应发生资源泄漏，即使错误地没有调用EndX方法。由于WebRequestState拥有ManualResetEvent，所以它拥有一个需要清理的资源。为此，状态对象使用了标准的IDisposable模式（通过Dispose()方法来清理资源）。

C.1.2 使用TPL调用APM

虽然TPL大幅简化了长时间运行方法的异步调用，但通常最好是使用API提供的APM方法，而不是针对同步版本编写TPL。这是由于API开发人员知道如何编写最高效的线程处理代码，知道同步哪些数据以及要使用什么同步类型。幸好，TPL的TaskFactory有一些特殊的方法，它们专门设计用来调用APM方法。所以，如果当前的开发允许使用TPL，但同时要使用和APM相关的API，那么还是可以用TPL来调用它们的。

1. TPL和CPS的APM

TPL包含FromAsync的一组重载版本以调用APM方法。代码清单C-4展示了一个例子，它扩展了其他APM例子以支持多个URL的下载。输出C-2展示了结果。

代码清单C-4 用TPL调用APM

```
using System;
using System.IO;
using System.Net;
using System.Linq;
using System.Threading.Tasks;

public class Program
{
    static private object ConsoleSyncObject =
        new object();

    public static void Main(string[] args)
    {
        string[] urls = args;
        if (args.Length == 0)
        {
            urls = new string[]
            {
                "http://www.habitat-spokane.org",
                "http://www.partnersintl.org",
                "http://www.iassist.org",
                "http://www.fh.org",
                "http://www.worldvision.org"
            };
        }

        Task[] tasks = new Task[urls.Length];
        for (int line = 0; line < urls.Length; line++)
        {
            tasks[line] = DisplayPageSizeAsync(
                urls[line], line);
        }

        while (!Task.WaitAll(tasks, 50))
        {
            DisplayProgress(tasks);
        }
        Console.SetCursorPosition(0, urls.Length);
    }

    private static Task<WebResponse>
        DisplayPageSizeAsync(string url, int line)
    {
        WebRequest webRequest = WebRequest.Create(url);
        WebRequestState state =
            new WebRequestState(webRequest, line);
        Write(state, url + " ");
        return Task<WebResponse>.Factory.FromAsync(
            webRequest.BeginGetResponse,
            GetResponseAsyncCompleted, state);
    }

    private static WebResponse GetResponseAsyncCompleted(
```

```

    IAsyncResult asyncResult)
    {
        WebRequestState completedState =
            (WebRequestState)asyncResult.AsyncState;
        HttpWebResponse response =
            (HttpWebResponse)completedState.WebRequest
                .EndGetResponse(asyncResult);
        Stream stream =
            response.GetResponseStream();
        using (StreamReader reader =
            new StreamReader(stream))
        {
            int length = reader.ReadToEnd().Length;
            Write(
                completedState, FormatBytes(length));
        }
        return response;
    }

    private static void Write(
        WebRequestState completedState, string text)
    {
        lock (ConsoleSyncObject)
        {
            Console.SetCursorPosition(
                completedState.ConsoleColumn,
                completedState.ConsoleLine);
            Console.Write(text);
            completedState.ConsoleColumn +=
                text.Length;
        }
    }

    private static void DisplayProgress(
        Task[] tasks)
    {
        for (int i = 0; i < tasks.Length; i++)
        {
            if (!tasks[i].IsCompleted)
            {
                DisplayProgress(
                    (WebRequestState)tasks[i]
                        .AsyncState);
            }
        }
    }

    private static void DisplayProgress(
        WebRequestState state)
    {
        lock (ConsoleSyncObject)
        {
            int left = state.ConsoleColumn;
            int top = state.ConsoleLine;
            if (left >= Console.BufferWidth -
                int.MaxValue.ToString().Length)
            {
                left = state.Url.Length;
            }
        }
    }

```

```

        Console.SetCursorPosition(left, top);
        Console.Write("").PadRight(
            Console.BufferWidth -
            state.Url.Length));

        state.ConsoleColumn = left;
    }

    Write(state, ".");
}

static public string FormatBytes(long bytes)
{
    string[] magnitudes =
        new string[] { "GB", "MB", "KB", "Bytes" };
    long max =
        (long)Math.Pow(1024, magnitudes.Length);

    return string.Format("{1:##.##} {0}",
        magnitudes.FirstOrDefault(
            magnitude =>
                bytes > (max /= 1024) )?? "0 Bytes",
        (decimal)bytes / (decimal)max).Trim();
}

class WebRequestState
{
    public WebRequestState(
        WebRequest webRequest, int line)
    {
        WebRequest = webRequest;
        ConsoleLine = line;
        ConsoleColumn = 0;
    }
    public WebRequestState(WebRequest webRequest)
    {
        WebRequest = webRequest;
    }
    public WebRequest WebRequest { get; private set; }
    public string Url
    {
        get
        {
            return WebRequest.RequestUri.ToString();
        }
    }
    public int ConsoleLine { get; set; }
    public int ConsoleColumn { get; set; }
}

```

输出C-2

```

http://www.habitat-spokane.org ..9.18 KB
http://www.partnersintl.org .....14.74 KB
http://www.iassist.org ..17.12 KB
http://www.fh.org .....35.09 KB
http://www.worldvision.org .....54.56 KB

```

很容易就可以将Task和一对APM方法连接起来。代码清单C-4使用的重载版本获取3个参数。首先是BeginX方法委托（webRequest.BeginGetResponse），然后是和EndX方法匹配的一个委托。虽然EndX方法（webRequest.EndGetResponse）可直接使用，但传递一个委托（GetResponseAsync Completed）并使用CPS，允许执行额外的完成（之后的）活动。最后一个参数是状态参数，它和BeginX方法接受的参数相似。

使用TPL调用一对APM方法的优点在于，不必担心在AsyncCallback方法完成时发出信号的问题。只需监视Task来判断是否完成。所以，WebRequestState中不再需要包含一个ManualResetEventSlim。

2. 使用TPL和ContinueWith()调用APM

调用TaskFactory.FromAsync()时，另一个选择是直接传递EndX方法，然后为后续的任何代码使用ContinueWith()。结果是可以用一个对象表示任何种类的异步操作，进而可以将基于任务的操作合并到一起，即使底层的实现是基于APM的。除此之外，可以查询Task延续参数（代码清单C-5的continueWithTask）来获取结果（continueWithTask.Result），而不必通过一个异步状态对象来访问EndX方法，也不必使用闭包和匿名委托（代码清单C-4存储了WebRequest）。

代码清单C-5 使用TPL，通过ContinueWith()调用APM

```
// ...

private static Task
    DisplayPageSizeAsync(string url, int line)
{
    WebRequest webRequest = WebRequest.Create(url);
    WebRequestState state =
        new WebRequestState(webRequest, line);
    Write(state, url + " ");
    return Task<WebResponse>.Factory.FromAsync(
        webRequest.BeginGetResponse,
        webRequest.EndGetResponse, state)
        .ContinueWith(
            (antecedent, antecedentState) =>
            {
                Stream stream =
                    antecedent.Result.
                        GetResponseStream();
                using (StreamReader reader =
                    new StreamReader(stream))
                {
                    int length =
                        reader.ReadToEnd().Length;
                    Write(state,
                        FormatBytes(length).ToString());
                }
            }, state);
}

// ...
```

注意，为了将状态传给ContinueWith()返回的Task，调用ContinueWith()时，在委托中显式包含了antecedentState（除了作为参数之外）。

3. 使用TAP调用APM方法

由于TAP正是为了处理延续任务而设计的，所以一个显而易见的增强（虽然仅限于C# 5.0）是改为使用async/await而不是ContinueWith()，如代码清单C-6所示。

代码清单C-6 使用TAP调用APM

```
// ...

private async static Task
    DisplayPageSizeAsync(string url, int line)
{
    WebRequestState state =
        new WebRequestState(url, line);
    Write(state, url + " ");
    WebRequest webRequest = WebRequest.Create(url);
    WebResponse webResponse =
        await Task<WebResponse>.Factory.FromAsync(
            webRequest.BeginGetResponse,
            webRequest.EndGetResponse, state);
    Stream stream =
        webResponse.GetResponseStream();
    using (StreamReader reader =
        new StreamReader(stream))
    {
        int length = reader.ReadToEnd().Length;
        Write(state,
            FormatBytes(length).ToString());
    }
}

// ...
```

■ 初学者主题：使用lock同步Console

在代码清单C-4中，我们反复地更改控制台的光标位置，然后向控制台写入文本。由于多个线程在同时执行，它们也要向控制台写入，也可能改变光标位置，所以需要同步修改光标和向控制台写入这两个操作，确保这个操作组是原子性的。

代码清单C-4包含object类型的一个ConsoleSyncObject作为同步lock标识符。每次移动光标时，或者向控制台写入时，都在一个lock构造中使用它，从而防止在移动光标和写入控制台之间发生一次临时更新。注意，即使单行的Console.WriteLine()语句，都被lock包围起来了。虽然它们是原子性的，但我们不希望它们打断非原子性的一个不同的块。只要有多个线程执行，所有控制台更改都需要同步。

C.2 异步委托调用

APM模式的一个特殊的实现称为异步委托调用（Asynchronous Delegate Invocation），它在所有委托数据类型上使用了特殊的、由C#编译器生成的代码。例如，给定Func<string, int>的一个委托实例，可以在这个实例上使用以下APM方法对：

```
System.IAsyncResult BeginInvoke(
    string arg, AsyncCallback callback, object @object)
int EndInvoke(IAsyncResult result)
```

结果是可以使用C#编译器生成的方法来同步地调用任何委托（进而调用任何方法）。

遗憾的是，异步委托调用模式使用的基础技术是一种不再继续开发的分布式编程技术，称为远程处理（remoting）。虽然微软仍然支持异步委托调用，而且在可以预见的将来，也不会放弃对它的支持，但和其他技术相比，它的性能显得比较一般。其他技术包括Thread、ThreadPool和TPL等。因此，在开发新项目时，开发人员应尽量选用其他技术，而不要使用异步委托调用API。下面的“高级主题”对这个模式进行了进一步讨论，方便开发人员了解它是如何工作的。

■ 高级主题：异步委托调用细节

通过异步委托调用，你不是通过一个对Task或Thread的显式引用来编码。相反，是使用委托实例和编译器生成的BeginInvoke()和EndInvoke()方法——它们的实现请求来自ThreadPool的线程。来考虑代码清单C-7中的代码。

代码清单C-7 异步委托调用

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Application started....");

        Console.WriteLine("Starting thread....");
        Func<int,string> workerMethod =
            PiCalculator.Calculate;
        IAsyncResult asyncResult =
            workerMethod.BeginInvoke(500, null, null);

        // Display periods as progress bar.
        while(!asyncResult.AsyncWaitHandle.WaitOne(
            100, false))
        {
            Console.Write('.');
        }
        Console.WriteLine();

        Console.WriteLine("Thread ending....");
        Console.WriteLine(
            workerMethod.EndInvoke(asyncResult));

        Console.WriteLine(
            "Application shutting down....");
    }
}
```

代码清单C-7的结果如输出C-3所示。

输出C-3

```

Application started....
Starting thread....
.....
Thread ending....
3.141592653589793238462643383279502884197169399375105820974944592307816406286
20899862803482534211706798214808651328230664709384460955058223172535940812848
11174502841027019385211055596446229489549303819644288109756659334461284756482
33786783165271201909145648566923460348610454326648213393607260249141273724587
00660631558817488152092096282925409171536436789259036001133053054882046652138
41469519415116094330572703657595919530921861173819326117931051185480744623799
6274956735188575272489122793818301194912
Application shutting down....

```

Main()首先赋值Func<int, string>类型的一个委托，它指向PiCalculator.Calculate(int digits)。

接着，代码调用BeginInvoke()。这个方法在来自线程池的一个线程上启动PiCalculator.Calculate()方法，并立即返回。这样一来，其他代码就可以和pi计算并行运行。在本例中，我们在等待PiCalculator.Calculate()方法完成期间打印句点符号。

我们在AsyncResult上使用IAsyncResult.AsyncWaitHandle.WaitOne()来轮询委托的状态——这与APM提供的机制一样。这样一来，PiCalculator.Calculate()方法每执行一秒，代码都会向屏幕打印一个句点符号。

等待句柄发出信号以后，代码就会调用EndInvoke()。和所有APM实现一样，必须向EndInvoke()传递与调用BeginInvoke()时返回的那个IAsyncResult引用相同的引用。在本例中，EndInvoke()没有阻塞，因为我们在while循环中轮询线程状态，只有在线程结束后才调用EndInvoke()。

代码清单C-5的例子传递一个整数并返回一个字符串——Func<int, string>的签名。异步委托调用的关键之处在于，将数据传入或传出目标调用是很容易的一件事情：只需像在APM模式中那样，与同步方法签名保持一致就可以了。假定一个包含out和ref参数的委托类型，如图C-2所示。（这个例子故意没有使用更常用的Func或Action，因为泛型不允许在类型参数上使用ref和out修饰符。）

```

System.IAsyncResult UpdateHandler.BeginInvoke(
    Object[] data, ref object value, out string text
    AsyncCallback callback, object @object);

delegate bool UpdateHandlerFunc
    object[] data, ref object value, out string text);

bool UpdateHandler.EndInvoke(
    ref object value, out text);

```

图C-2 BeginInvoke()和EndInvoke()的委托参数分布情况

除了额外的AsyncCallback和object参数，BeginInvoke()方法和委托签名是匹配的。和IAsyncResult返回值相似，附加的参数对应于指定了一个回调和传递了状态对象的标准APM参数。类似地，EndInvoke()方法也和原始签名匹配，只是其中只出现了输出参数。由于object[]数据是唯一的输入，所以它不在EndInvoke()方法中出现。另外，由于EndInvoke()方法结束一个异步调用，所以它的返回值也和原始委托的返回值匹配。

由于所有委托都包含C#编译器生成且由异步委托调用模式使用的BeginInvoke()和EndInvoke()方法，所以以同步方式调用任何方法（尤其是在给定Func和Action委托的时候），都显得相当容易。除此之外，调用者可以简单地以异步方式调用一个方法，无论API程序员是否显式地实现了它。

在TPL之前，异步委托调用模式比其他替代方案容易得多，所以假如一个API没有提供显式的异步调用模式，一般都会选用它。然而，在TPL问世之后，除非是为了与.NET 3.5和早期框架版本兼容，否则异步委托调用越来越没有什么用了。

C.3 基于事件的异步模式（EAP）^①

前面一直假定异步方法返回一个任务；一旦设置任务的状态和结果，就向调用者通知异步工作完成。这样做还可能造成任务以异步方式完成。虽然该模式很常见，功能也很强大，但并不是唯一的异步模式。另一个常见的模式是**基于事件的异步模式**（Event-based Asynchronous Pattern, EAP）。该模式尤其适合长时间运行的异步工作。

使用EAP的方法通常具有以Async结尾的名称，返回void，而且没有out参数。EAP方法经常还要获取一个object或泛型参数，其中包含了由调用者确定的、与异步工作关联的状态。如果异步工作可以取消，那么有时还要获取一个取消标志。例如，假定一个EAP方法要计算pi的指定位数并用string返回，方法签名就可能是：

或者 `void CalculateAsync(int digits)`

```
void CalculateAsync(
    int digits, object state, CancellationToken ct)
```

这些签名很明显缺少了“结果”。以前讲过的异步方法都会返回一个Task<string>，可通过它来获取异步计算结果。相反，EAP方法根本没有返回值。

直到现在都还没有说到“基于事件的异步模式”的“事件”部分。方法和一个事件关联；EAP方法的调用者向关联的事件登记一个事件处理程序，然后调用方法。方法开始异步工作并返回；异步工作完成后触发事件，随后执行事件处理程序。传给处理程序的事件实参包含计算好的字符串，以及异步方法提供给侦听者的其他任何有用信息，比如由调用者提供的状态、有关任何异常的信息以及异步操作期间发生的取消等。（毫不奇怪，本来由Task对象提供的信息现在通过事件处理程序的实参来提供。）

代码清单C-8展示了如何将基于任务的异步机制作为EAP方法的实现细节来使用。EAP方法CalculateAsync<TState>()关联了CalculateCompleted事件。异步方法创建一个任务（默认在从线程池获得的线程上运行）来执行计算。任务延续在任务完成时触发事件。

和async/await相似，我们希望触发事件的延续总是在和原始异步方法一样的线程上执行。为此需要从TaskScheduler类请求同步上下文。由于这是控制台应用程序，所以当前线程没有同步上下文（造成它默认依赖于线程池），所以代码清单C-8首先展示了默认上下文的创建。

^① 参见JoeDuff著的*Concurrent Programming on Windows*（Addison-Wesley，2009），第421~426页。

代码清单C-8 基于事件的异步模式

```
using System;
using System.ComponentModel;
using System.Threading;
using System.Threading.Tasks;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

partial class PiCalculation
{
    public void CalculateAsync<TState>(
        int digits,
        CancellationToken cancellationToken
            = default(CancellationToken),
        TState userState
            = default(TState))
    {
        SynchronizationContext.
            SetSynchronizationContext(
                AsyncOperationManager.
                    SynchronizationContext);

        // Ensure the continuation runs on the current thread, and that
        // therefore the event will be raised on the same thread that
        // called this method in the first place.
        TaskScheduler scheduler =
            TaskScheduler.
                FromCurrentSynchronizationContext();
        Task.Run(
            () =>
            {
                return PiCalculator.Calculate(digits);
            }, cancellationToken)
            .ContinueWith(
                continueTask =>
                {
                    Exception exception =
                        continueTask.Exception == null ?
                        continueTask.Exception :
                        continueTask.Exception.
                            InnerException;
                    CalculateCompleted(
                        typeof(PiCalculator),
                        new CalculateCompletedEventArgs(
                            continueTask.Result,
                            exception,
                            cancellationToken.IsCancellationRequested,
                            userState));
                }, scheduler);
    }

    public event
        EventHandler<CalculateCompletedEventArgs>
        CalculateCompleted = delegate { };
}

public class CalculateCompletedEventArgs
    : AsyncCompletedEventArgs
```

```

{
    public CalculateCompletedEventArgs(
        string value,
        Exception error,
        bool cancelled,
        object userState) : base(
            error, cancelled, userState)
    {
        Result = value;
    }
    public string Result { get; private set; }
}
}

```

如前所述，EAP方法特别适合长时间运行的异步操作。这种操作不仅提供了任务完成、失败或取消时的通知，有时还提供了进度更新。如果UI需要用进度条或其他界面元素显示长时间运行的异步操作的进度，这个功能就非常有用。为此，EAP方法采取的标准方式是将方法与ProgressChangedEventHandler类型的ProgressChanged事件关联。

注意，EAP方法及其相关事件一般是实例成员，而不是静态成员。这样可以更好地支持多个并发操作，因为每个操作都可以和不同的实例关联。

C.4 Background Worker模式

能提供操作状态和取消机制的另一个模式是Background Worker模式，它是EAP的特殊实现。.NET Framework 2.0（及更高版本）专门包含了一个BackgroundWorker类来进行这种模式的编程。

代码清单C-9是这种模式的一个例子。它同样计算pi值到指定的位数。

代码清单C-9 使用Background Worker模式

```

using System;
using System.Threading;
using System.ComponentModel;
using System.Text;

public class PiCalculator
{
    public static BackgroundWorker calculationWorker =
        new BackgroundWorker();
    public static AutoResetEvent resetEvent =
        new AutoResetEvent(false);

    public static void Main()
    {
        int digitCount;

        Console.Write(
            "Enter the number of digits to calculate:");
        if (int.TryParse(
            Console.ReadLine(), out digitCount))
        {
            Console.WriteLine("ENTER to cancel");
            // C# 2.0 Syntax for registering delegates
            calculationWorker.DoWork += CalculatePi;
            // Register the ProgressChanged callback

```

```

        calculationWorker.ProgressChanged +=
            UpdateDisplayWithMoreDigits;
        calculationWorker.WorkerReportsProgress =
            true;
        // Register a callback for when the calculation completes
        calculationWorker.RunWorkerCompleted +=
            new RunWorkerCompletedEventHandler(
                Complete);
        calculationWorker.
            WorkerSupportsCancellation = true;

        // Begin calculating pi for up to digitCount digits
        calculationWorker.RunWorkerAsync(
            digitCount);

        Console.ReadLine();
        // If cancel is called after the calculation
        // has completed it doesn't matter.
        calculationWorker.CancelAsync();
        // Wait for Complete() to run.
        resetEvent.WaitOne();
    }
    else
    {
        Console.WriteLine(
            "The value entered is an invalid integer.");
    }
}

```

```

private static void CalculatePi(
    object sender, DoWorkEventArgs eventArgs)
{
    int digits = (int)eventArgs.Argument;

    StringBuilder pi =
        new StringBuilder("3.", digits + 2);
    calculationWorker.ReportProgress(
        0, pi.ToString());

    // Calculate rest of pi, if required
    if (digits > 0)
    {
        for (int i = 0; i < digits; i += 9)
        {
            // Calculate next i decimal places
            int nextDigit =
                PiDigitCalculator.StartingAt(
                    i + 1);
            int digitCount =
                Math.Min(digits - i, 9);
            string ds =
                string.Format("{0:D9}", nextDigit);
            pi.Append(ds.Substring(0, digitCount));

            // Show current progress
            calculationWorker.ReportProgress(
                0, ds.Substring(0, digitCount));

            // Check for cancellation

```

```

        if (
            calculationWorker.CancellationPending)
        {
            // Need to set Cancel if you need to
            // distinguish how a worker thread completed
            // i.e., by checking
            // RunWorkerCompletedEventArgs.Cancelled
            eventArgs.Cancel = true;
            break;
        }
    }

    eventArgs.Result = pi.ToString();
}

private static void UpdateDisplayWithMoreDigits(
    object sender,
    ProgressChangedEventArgs eventArgs)
{
    string digits = (string)eventArgs.UserState;

    Console.Write(digits);
}

static void Complete(
    object sender,
    RunWorkerCompletedEventArgs eventArgs)
{
    // ...
}

public class PiDigitCalculator
{
    // ...
}

```

C.4.1 模式的建立

建立Background Worker模式的过程如下。

(1) 为BackgroundWorker.DoWork事件注册长时间运行的方法。在本例中，要长时间运行的任务是对CalculatePi()的调用。

(2) 为了接收进展或状态通知，要为BackgroundWorker.ProgressChanged挂接一个侦听器，并将BackgroundWorker.WorkerReportsProgress设为true。在代码清单C-21中，UpdateDisplay WithMoreDigits()负责在更多的数位可用时更新显示。

(3) 为BackgroundWorker.RunWorkerCompleted事件注册一个方法(Complete())。

(4) 为WorkerSupportsCancellation属性赋值以支持取消一个操作。将true值赋给该属性以后，对BackgroundWorker.CancelAsync的调用就会设置DoWorkEventArgs.CancellationPending标志。

(5) 在DoWork提供的方法(CalculatePi())内，检查DoWorkEventArgs.CancellationPending属性值，并在它为true时退出方法。

(6) 一切都设置好之后, 调用`BackgroundWorker.RunWorkerAsync()`, 并提供要传给指定`DoWork()`方法的一个状态参数来开始工作。

分解成以上小步骤之后, `Background Worker`模式就显得很容易理解。另外, 由于它本质上是一种EAP, 所以提供了对进度通知的显式支持。缺点在于, 不能在任意方法上使用它。相反, `DoWork()`方法必须遵循一个`System.ComponentModel.DoWorkEventHandler`委托, 后者获取`object`和`DoWorkEventArgs`类型的参数。如果情况不是这样的, 就需要一个包装函数——用匿名方法可轻松达到目的。与取消和进展通知有关的方法也需要特定的签名, 不过这些由建立`Background Worker`模式的程序员进行控制。

C.4.2 异常处理

后台的工作者(worker)线程异步执行时, 如果发生未处理异常, `RunWorkerCompleted`委托的`RunWorkerCompletedEventArgs`参数的`Error`属性会被设置成`Exception`实例。因此, 代码清单C-10通过在`RunWorkerCompleted`回调内检查`Error`属性来提供异常处理机制。

代码清单C-10 从工作者线程中处理未处理的异常

```
// ...
static void Complete(
    object sender, RunWorkerCompletedEventArgs eventArgs)
{
    Console.WriteLine();
    if (eventArgs.Cancelled)
    {
        Console.WriteLine("Cancelled");
    }
    else if (eventArgs.Error != null)
    {
        // IMPORTANT: check error to retrieve any exceptions.
        Console.WriteLine(
            "ERROR: {0}", eventArgs.Error.Message);
    }
    else
    {
        Console.WriteLine("Finished");
    }
    resetEvent.Set();
}
// ...
```

代码一定要在`RunWorkerCompleted`回调内检查`eventArgs.Error`, 这一点很重要。否则, 异常会逃脱检测——甚至不会报告给`AppDomain`。

C.5 Windows UI编程

使用`System.Windows.Forms`和`System.Windows`命名空间来进行用户界面(UI)开发时, 也必须注意线程处理问题。第18章讲过, Microsoft Windows操作系统使用的是单线程的、基于消息处理的用户界面。这意味着一次只能有一个线程访问用户界面, 与任何轮换线程的交互都必须通过Windows消息泵来封送。幸好, 由于TAP在执行延续任务时使用同步上下文, 所以`await`表达式之后的调用可以自由地调用UI API, 不用担心将调用封送给UI线程的问题。遗憾的是, C#以前的版本情况并非如此。相反, 为了在UI线程上调用UI方

法，需要为Windows窗体和WPF API使用特殊的调用逻辑。

C.5.1 Windows窗体

进行Windows窗体编程时，为了检查是否允许从一个线程中发出UI调用，需要调用一个组件的InvokeRequired属性，判断是否需要封送处理（marshaling）。如果InvokeRequired返回true，表明需要封送，并可通过一个Invoke()调用来实现。尽管Invoke()在内部无论如何都会检查InvokeRequired，但更有效率的做法是提前显式地检查这个属性。代码清单C-11演示了这个模式。

代码清单C-11 通过Invoke()访问用户界面

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;

class Program : Form
{
    private System.Windows.Forms.ProgressBar _ProgressBar;

    [STAThread]
    static void Main()
    {
        Application.Run(new Program());
    }

    public Program()
    {
        InitializeComponent();
        // Use Task.Factory.StartNew for .NET 4.0
        Task task = Task.Run((Action)Increment);
    }

    void UpdateProgressBar()
    {
        if (_ProgressBar.InvokeRequired)
        {
            MethodInvoker updateProgressBar =
                UpdateProgressBar;
            _ProgressBar.BeginInvoke(updateProgressBar);
        }
        else
        {
            _ProgressBar.Increment(1);
        }
    }

    private void Increment()
    {
        for (int i = 0; i < 100; i++)
        {
            UpdateProgressBar();
            Thread.Sleep(100);
        }

        if (InvokeRequired)
```

```

        {
            // Close cannot be called directly from a non-UI thread.
            Invoke(new MethodInvoker(Close));
        }
        else
        {
            Close();
        }
    }

    private void InitializeComponent()
    {
        _ProgressBar = new ProgressBar();
        SuspendLayout();

        _ProgressBar.Location = new Point(13, 17);
        _ProgressBar.Size = new Size(267, 19);

        ClientSize = new Size(292, 53);
        Controls.Add(this._ProgressBar);
        Text = "Multithreading in Windows Forms";
        ResumeLayout(false);
    }
}

```

这个程序会显示一个窗口，其中包含一个进度条。进度条会自动变长。进度条显示到百分之百时，对话框就会关闭。

如代码清单C-11所示，必须检查InvokeRequired两次，如果返回true，就将调用封送到UI线程。在两种情况下，封送处理都涉及实例化一个MethodInvoker委托，然后将这个委托传给Invoke()。由于封送到另一个线程可能是相当慢的过程，所以可以通过BeginInvoke()和EndInvoke()来进行异步调用。

Invoke()、BeginInvoke()、EndInvoke()和InvokeRequired构成了System.ComponentModel.ISynchronizeInvoke接口的成员。该接口由System.Windows.Forms.Control实现，所有Windows窗体控件都是从这个Control类派生的。

C.5.2 Windows Presentation Foundation (WPF)

为了在WPF平台上实现相同的封送检查，需要采取稍有不同的一种方式。WPF包含名为Current的静态成员属性，它的类型是DispatcherObject，由System.Windows.Application类提供。在DispatcherObject上调用CheckAccess()，效果等同于在Windows窗体控件上调用InvokeRequired。

代码清单C-12用一个静态UIAction对象对此进行了演示。开发人员每次调用一个可能与UI进行交互的方法时，只需调用UIAction.Invoke()，并传递一个委托来指向想调用的UI代码。在Invoke()中，要检查调度器对象，判断是否需要封送处理，并作出相应的响应。

代码清单C-12 安全地调用用户界面对象

```

using System;
using System.Windows;
using System.Windows.Threading;

public static class UIAction

```

```

{
    public static void Invoke<T>(
        Action<T> action, T parameter)
    {
        Invoke(() => action(parameter));
    }
    public static void Invoke(Action action)
    {
        DispatcherObject dispatcher =
            Application.Current;
        if (dispatcher == null
            || dispatcher.CheckAccess()
            || dispatcher.Dispatcher == null
        )
        {
            action();
        }
        else
        {
            SafeInvoke(action);
        }
    }
}

// We want to catch all exceptions here so we can rethrow
private static void SafeInvoke(Action action)
{
    Exception exceptionThrown = null;
    Action target = () =>
    {
        try
        {
            action();
        }
        catch (Exception exception)
        {
            exceptionThrown = exception;
        }
    };
    Application.Current.Dispatcher.Invoke(target);
    if (exceptionThrown != null)
    {
        // Use ExceptionDispatchInfo.Throw() for .NET 4.5+
        throw exceptionThrown;
    }
}
}

```

在代码清单C-12中，UIAction的一个额外的功能是“封送”UI线程中可能发生的任何异常。SafeInvoke()将请求的所有委托调用都包装到一个try/catch块中，如果引发一个异常，就暂存异常，并在线程上下文返回调用线程时重新引发它。这样一来，UIAction就能避免引发在UI线程上的未处理异常。

附录D

C# 5.0 async/await模式之前的计时器

第19章介绍了如何将Task.Delay()作为计时器使用。.NET 4.5之前可以使用几个计时器类，包括System.Windows.Forms.Timer、System.Timers.Timer和System.Threading.Timer。

在当初设计System.Windows.Forms.Timer的时候，开发团队有意把它设计成在一个富客户端（rich client）用户界面中使用。可把它作为非可视控件拖放到窗体上，并从“属性”窗口中控制其行为。更重要的是，它总是能从一个与用户界面交互的线程中安全地触发事件。

另两个计时器非常相似。System.Timers.Timer是System.Threading.Timer的包装器，是对其功能的抽象。具体地说，System.Threading.Timer不是从System.ComponentModel.Component派生的，所以不能作为组件容器（组件容器必须实现System.ComponentModel.IContainer）中的一个组件使用。另一个区别在于，System.Threading.Timer允许从启动计时器的调用向触发计时器通知的调用传递一个状态（一个object参数）。其他区别主要反映在API的使用上，System.Timers.Timer支持一个同步对象，而且调用要显得更直观一些。无论System.Timers.Timer还是System.Threading.Timer都设计成在服务器类型的进程中使用，但System.Timers.Timer包含一个同步对象，这允许它同UI交互。除此之外，两个计时器都要使用系统线程池。表D-1对各种计时器进行了总结。

表D-1 各种计时器的功能对比

功能描述	System.Timers.Timer	System.Threading.Timer	System.Windows.Forms.Timer
支持在计时器实例化之后添加和删除侦听器	是	否	是
支持用户界面线程上的回调	是	否	是
从自线程池获取的线程进行回调	是	是	否
支持在Windows窗体设计器中拖放	是	否	是
适合在一个多线程服务器环境中运行	是	是	否

续表

功能描述	System.Timers.Timer	System.Threading.Timer	System.Windows.Forms.Timer
支持将任意状态从计时器初始化传递至回调	否	是	否
实现IDisposable	是	是	是
支持开关式回调和定期重复回调	是	是	是
可穿越应用程序域的边界访问	是	是	是
支持IComponent，可容纳在一个IContainer中	是	否	是

为用户界面编程使用System.Windows.Forms.Timer是一个相当明显的选择。唯一要注意的是，在用户界面线程上进行的一次长时间操作可能延迟一个计时器的过期时间。（理论上，依赖线程池的计时器会发生相似的延迟，因为线程池可能正好处于忙碌状态。）至于另外两个计时器，它们的差异其实并不大。假如需要容纳在一个IContainer中，那么肯定应该选择System.Timers.Timer。然而，假如不需要专门的System.Timers.Timer功能，就默认选择System.Threading.Timer，因为它是一个稍微轻量级的实现。

代码清单D-1和代码清单D-2分别演示了如何使用System.Timers.Timer和System.Threading.Timer。两段代码非常相似，都支持在using语句中实例化，因为两者都支持IDisposable。两段代码的输出完全一样，如输出D-1所示。代码的作用是显示与一个计数器关联的时间标记，从而指出计时器的触发次数。一旦完成，输出将验证计时器线程不是Main线程，同时显示最终计数。

代码清单D-1 使用System.Timers.Timer

```
using System;
using System.Timers;
using System.Threading;
// Because Timer exists in both the System.Timers and
// System.Threading namespaces, you disambiguate "Timer"
// using an alias directive.
using Timer = System.Timers.Timer;

class UsingSystemTimersTimer
{
    private static int _Count=0;
    private static readonly ManualResetEvent _ResetEvent =
        new ManualResetEvent(false);
    private static int _AlarmThreadId;

    public static void Main()
    {
        using( Timer timer = new Timer() )
        {
            // Initialize Timer
            timer.AutoReset = true;
            timer.Interval = 1000;
            timer.Elapsed +=
                new ElapsedEventHandler(Alarm);

            timer.Start();

            // Wait for Alarm to fire for the 10th time.
            _ResetEvent.WaitOne();
        }
    }
}
```

```

// Verify that the thread executing the alarm
// Is different from the thread executing Main
if(_AlarmThreadId ==
    Thread.CurrentThread.ManagedThreadId)
{
    throw new ApplicationException(
        "Thread Ids are the same.");
}
if(_Count < 9)
{
    throw new ApplicationException(
        " _Count < 9");
};

Console.WriteLine(
    "(Alarm Thread Id) {0} != {1} (Main Thread Id)",
    _AlarmThreadId,
    Thread.CurrentThread.ManagedThreadId);
Console.WriteLine(
    "Final Count = {0}", _Count);
}

```

```

static void Alarm(
    object sender, ElapsedEventArgs eventArgs)
{
    _Count++;

    Console.WriteLine("{0}:- {1}",
        eventArgs.SignalTime.ToString("T"),
        _Count);

    if (_Count >= 9)
    {
        _AlarmThreadId =
            Thread.CurrentThread.ManagedThreadId;
        _ResetEvent.Set();
    }
}
}

```

在代码清单D-1中，为System.Threading和System.Timers都使用了using指令。这使Timer类型变得不明确。所以，要用一个别名将Timer显式地与System.Timers.Timer关联起来。

System.Threading.Timer值得注意的一个特点是，它允许由构造器获取回调委托和时间间隔。

代码清单D-2 使用System.Threading.Timer

```

using System;
using System.Threading;

class UsingSystemThreadingTimer
{
    private static int _Count=0;
    private static readonly AutoResetEvent _ResetEvent =
        new AutoResetEvent(false);
    private static int _AlarmThreadId;

    public static void Main()
    {
        // Timer(callback, state, dueTime, period)
        using( Timer timer =
            new Timer(Alarm, null, 0, 1000) )
        {

```

```

// Wait for Alarm to fire for the 10th time.
_ResetEvent.WaitOne();
}

// Verify that the thread executing the alarm
// Is different from the thread executing Main
if(_AlarmThreadId ==
    Thread.CurrentThread.ManagedThreadId)
{
    throw new ApplicationException(
        "Thread Ids are the same.");
}
if(_Count < 9)
{
    throw new ApplicationException(
        " _Count < 9");
};

Console.WriteLine(
    "(Alarm Thread Id) {0} != {1} (Main Thread Id)",
    _AlarmThreadId,
    Thread.CurrentThread.ManagedThreadId);
Console.WriteLine(
    "Final Count = {0}", _Count);
}

static void Alarm(object state)
{
    _Count++;

    Console.WriteLine("{0}:- {1}",
        DateTime.Now.ToString("T"),
        _Count);

    if (_Count >= 9)
    {
        _AlarmThreadId =
            Thread.CurrentThread.ManagedThreadId;
        _ResetEvent.Set();
    }
}
}

```

输出D-1

```

12:19:36 AM:- 1
12:19:37 AM:- 2
12:19:38 AM:- 3
12:19:39 AM:- 4
12:19:40 AM:- 5
12:19:41 AM:- 6
12:19:42 AM:- 7
12:19:43 AM:- 8
12:19:44 AM:- 9
(Alarm Thread Id) 4 != 1 (Main Thread Id)
Final Count = 9

```

在System.Threading.Timer实例化之后,可以通过Change()方法更改计时器的启动时间和方法调用间隔。但是,不能在实例化之后更改回调侦听器。要进行这样的更改,必须创建一个新实例。