PROGRAMMER TO PROGRAMMER™

# C# Threading Handbook

# C#
# 线程参考手册

(美) Tobin Titus
Fabio Claudio Ferracchiati 等著
王 敏 译

清华大学出版社

PROGRAMMER TO PROGRAMMER™

# C# Threading Handbook

本书介绍了 Windows 和.NET 程序的一个基础单元——线程。开发人员在开发高效的 C# 程序时，应很好地理解线程在程序执行过程中的作用，多个线程应如何交互才能提高程序的效率，以及开发多线程应用程序时应注意的问题。

线程是 GUI 和服务器应用程序执行的基本方式。如果代码在 GUI 中运行，说明这些代码就是线程代码，ASP.NET 页面也运行在线程环境中。本书旨在介绍.NET 中使用线程的内容，特别是 C# 开发人员应如何使用线程。线程并不容易掌握，而掌握.NET 平台的编程技术是学习线程的必要一步。

## 本书主要内容

- .NET 应用程序的执行方式
- 线程在.NET 中的生命周期
- .NET Framework 使用线程的方式
- 线程在事件驱动的环境中如何工作
- 如何避免资源的竞争和死锁
- 如何同步执行多线程
- 如何调试多线程应用程序

## 本书读者对象

本书面向的是需要在应用程序中使用.NET Framework 的线程功能的 C# 开发人员。要阅读本书，读者应熟悉 C# 语法，并曾使用 C# 编写过代码。不过，读者不需要具备任何有关线程方面的知识。

ASP Today
www.asptoday.com

C# Today
www.csharptoday.com

p2p.wrox.com
The programmer's resource center

wrox base
www.wroxbase.com

ISBN 7-302-07403-8

9 787302 074038 >

定价：33.00 元

Recommended Computer Book Categories

Programming

.NET, C#

wrox direct

# C#线程参考手册

(美) Tobin Titus
Fabio Claudio Ferracchiati 等著
王　敏　　译

清华大学出版社

北　京

# 内 容 简 介

本书自上而下地介绍了.NET 执行 C#代码的方法。首先描述了 Windows 线程的定义，它们与.NET 进程、应用程序域的关系以及线程之间的关系。讨论了线程的调度(操作系统如何确定下一个要处理的线程)，接着论述了如何编写.NET 代码来处理线程。之后介绍了线程的同步，让多个线程安全地访问同一资源。本书还介绍了多线程应用程序使用的一些典型的体系结构，尤其是线程池，并阐述了如何调试多线程代码。最后用一个完整的例子来说明如何利用线程来建立可伸缩的、高性能的网络服务器。

本书适合从事.NET 开发的 C#程序员阅读，不要求读者具备任何线程方面的知识。

EISBN: 1-86100-829-5
C# Threading Handbook
Tobin Titus　Fabio Claudio Ferracchiati , et al.
Copyright©2003 by Wrox Press Ltd.
Original English Language Edition Published by Wrox Press Ltd.
All Rights Reserved.

本书中文简体字版由英国乐思出版公司授权清华大学出版社出版。来经出版者书面许可，不得以任何方式复制或抄袭本书内容。

**版权所有，翻印必究。**

**本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。**

**北京市版权局著作权合同登记号：01-2002-6525**

**图书在版编目(CIP)数据**

C# 线程参考手册/(美)泰特斯(Titus, T.)等著；王敏译.—北京：清华大学出版社，2003
书名原文：C# Threading Handbook
ISBN 7-302-07403-8

Ⅰ. C…　Ⅱ. ①泰…②王…　Ⅲ. C 语言—程序设计—技术手册　Ⅳ. TP312-62
中国版本图书馆 CIP 数据核字(2003)第 092471 号

**出 版 者**：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
**社 总 机**：010-62770175　　**客户服务**：010-62776969
**组稿编辑**：曾康
**文稿编辑**：李阳
**封面设计**：康博
**版式设计**：康博
**印 刷 者**：北京大中印刷厂
**装 订 者**：三河市兴旺装订有限公司
**发 行 者**：新华书店总店北京发行所
**开　　本**：185×260　**印张**：16.25　**字数**：337 千字
**版　　次**：2003 年 11 月第 1 版　2003 年 11 月第 1 次印刷
**书　　号**：ISBN 7-302-07403-8/TP・5468
**印　　数**：1～4000
**定　　价**：33.00 元

# 前　言

多线程就是允许复杂的应用程序在同一时刻执行多项任务。这些程序可以响应用户事件，同时访问网络资源或文件系统。根据平台和操作系统提供的对进程的不同控制，编写这类并发应用程序的方式也不尽相同。例如，Visual Basic 6 并没有提供这类控制，而是在后台实现线程的，所以当触发事件时，它将在某个线程模型中运行相关的处理代码，但应用程序的程序员并不需要考虑这个线程。而 Visual C++开发人员必须充分考虑到 Windows 线程和处理模型的复杂性，同时拥有这种线程模型的强大功能：C++程序员可以轻松地创建出多线程的程序，但必须学习如何使用很多复杂的技巧，以确保线程在自己的控制之下。

.NET Framework 的托管编码环境提供了一个完整而功能强大的线程模型，该模型允许编程人员精确控制在一个线程中运行的内容，线程何时退出，以及它将访问多少数据等。但是，Common Language Runtime 除了负责程序员不必考虑的内存管理之外，它还负责管理和清理线程。所以，在.NET 中，既提供了 C++的强大功能，又具有 Visual Basic 的简单性。这就是说，多线程应用程序引入了单线程程序从来都不会碰到的一整套编程问题。

本书介绍如何利用.NET Framework 所提供的线程功能，学习线程所提供的各种特性，并指出如何在使用线程的过程中避免可能会遇到的陷阱。

什么时候使用线程？实际上，所有的程序都是在线程中执行的，所以理解.NET 和 Windows 如何执行线程，将有助于理解程序在运行期间的执行情况。Windows Forms 应用程序使用事件循环线程来处理用户界面事件。各个窗体在不同的线程上执行，如果需要在 Windows Forms 之间交互，就需要在线程之间交互。ASP.NET 页面在 IIS 的多线程环境中执行——同一页面上的不同请求可以在不同的线程中执行，同一页面可以同时在多个线程中执行。在访问 ASP.NET 页面上的共享资源时，就会遇到线程问题。

除了编写在这类多线程环境下运行的代码之外，还常常需要创建和控制我们自己的线程。例如，需要创建一个应用程序，该应用程序在处理数据时从来或很少等待，但总是可以响应用户和事件。只有建立一个多线程的应用程序，才能实现这一功能。网上有许多这方面的文章，其他图书也介绍了如何利用.NET Framework 创建线程，如何执行基本的操作，不过，实现代码只完成了一部分工作。当使用多线程应用程序时，有一些操作通常会阻塞应用程序，例如文件系统操作，此时理想的解决方法就是使用线程。当同一时刻有多个线程在同一个文件上进行操作时，就会出现同步问题或可伸缩性问题。本书除了介绍如何创建和操纵线程之外，还将讲解如何设计应用程序，应用合适类型的锁，并在它等候其他操作完成时不阻塞线程，以避免可能遇到的这些问题。

## 本书读者对象

本书适合于研究.NET 平台的所有功能的 C#开发人员。如果希望理解 C#代码在.NET Runtime 中的执行情况，编写可以在多线程系统中安全运行的代码，以及在自己的代码中创建

和控制线程，那么阅读本书将会大有帮助。

## 本书内容

本书自上而下详细介绍了.NET 如何执行 C#代码。首先论述什么是 Windows 线程，线程与.NET 过程、应用程序域的关系以及线程之间的关系。然后介绍线程的调度(操作系统如何确定下一个要处理的线程)，讨论如何编写.NET 代码来操作线程。接着介绍线程的同步，以便让多个线程安全地访问同一资源。之后，阐述多线程程序使用的一些典型体系结构，尤其要详细介绍线程池。本书还将探讨如何调试多线程代码。最后用一个完整的例子来说明如何利用线程来建立可伸缩的、高性能的网络服务器。

下面是本书每一章讲述的内容。

**第 1 章　定义线程**

本章讲述什么是线程，线程在.NET 中的作用，如何在操作系统中创建、执行和中断线程。

**第 2 章　.NET 中的线程**

本章将介绍第 1 章中探讨的概念如何在.NET 中实现。讨论 C#代码如何创建线程、如何访问其状态和生命周期等信息以及如何执行基本的操作，例如睡眠、停止和中断。

**第 3 章　使用线程**

本章深入介绍了如何在应用程序中使用多线程，讨论如何实现同步和锁定，确保每次只有一个线程对数据进行独占式访问，并说明死锁的危险性，以及如何避免它。

**第 4 章　线程设计原则**

本章将介绍在多线程代码中使用的一些常见模式—— 这些体系结构您可以放心地使用，只要按照其中的提示和经过测试的原则来实现它们，就可以避免死锁。

**第 5 章　线程应用程序的伸缩**

不能无休止地创建线程。有了线程，反复的次数就会减少。但是，在不同的线程上同时执行多个任务时，利用线程池就可以达到相同的效果，而不必创建太多的线程。本章介绍.NET 的线程池，以及如何实现自己的线程池。

**第 6 章　调试与跟踪线程**

多线程应用程序调试起来很复杂。本章将讨论.NET 中的一些最有效的调试工具，并解释如何使用它们调试多线程代码。

**第 7 章　联网与线程**

联网操作在单线程程序中非常慢。这种单线程应用程序把许多时间都浪费在等待网络上传输的信息，在这段等待的时间里，它什么也不做。因此，多线程就成为网络应用程序中的一个普遍要求，即在等待网络信息时能执行其他操作。本章将介绍如何利用线程建立快速的、可伸缩的网络服务器。

## 本书的使用要求

要使用本书，需要能编译和执行 C#代码，也就是说，需要安装如下软件：

- .NET Framework SDK，可从 Microsoft 的 MSDN 站点(http://msdn.microsoft.com)的 Software Development 目录上获得。本书出版时，该下载页面可以通过下面的 URI 访问：http://msdn.microsoft.com/download/sample.asp?url=msdn-files/027/000/976/msdncompositedoc.xml
- 包含 Visual C# .NET 的 Visual Studio .NET 版本。Visual C# .NET IDE 的 2002 版包含在下列 Microsoft 产品中：
  - ♦ Microsoft Visual C# .NET Standard
  - ♦ Microsoft Visual Studio .NET Enterprise Architect
  - ♦ Microsoft Visual Studio .NET Enterprise Developer
  - ♦ Microsoft Visual Studio .NET Professional

这些产品的主页站点是 http://msdn.microsoft.com/vstudio/。

还有几个用于其他平台的 .NET 版本，支持 Linux、UNIX 和 Windows 的 C#版本由 Mono 项目提供(http://www.go-mono.com)。Mono 代码不能访问所有的 Microsoft .NET 类库，但其语法规则与 Microsoft 的 C#相同。其线程模型与.NET 不完全相同，由于本书介绍的类和功能的实现是 Mono 平台的部分目标，所以本书的内容也可以用于 Mono 平台。但是，本书的代码没有在 Mono 平台上测试过。

# 目　录

# 第1章 定 义 线 程

线程技术是指开发架构将应用程序的一部分分离为“线程”，使线程与程序其余部分的执行步骤不一致。在绝大多数编程语言中，都有一个相当于 Main()的方法，该方法中的每一行都按顺序执行，下一行代码只能在前一行代码执行完之后才能执行。线程是一种特殊的对象，是操作系统执行多任务的一部分，它允许应用程序的一部分独立于其他对象而单独运行，因此也就脱离了应用程序常规的执行顺序。稍后还会讨论多任务的不同类型。

另一个概念是“自由线程”，对于大多数 C++或 Java 开发人员来说，自由线程并不是一个新概念。本章将定义这个术语，进一步介绍 C#对自由线程所提供的支持，并把自由线程模型和其他模型(如 Visual Basic 6.0 的单元线程模型)进行简单的比较。我们不会对它们之间的区别进行深入的介绍，毕竟本书介绍的重点不在于此。然而，了解这些线程模型的区别有助于了解自由线程的优点。本章所介绍的概念是本书其余部分的基础，本章主要介绍以下内容：

- 线程的概念
- 各种多任务和线程模型之间的比较
- 线程存在的位置以及如何为它们分配处理器时间
- 如何使用中断和优先级来控制和管理线程
- 应用程序域的概念，以及在应用程序的安全上应用程序域是如何提供比在简单进程环境中更精细的控制粒度

在学习本书其他章节所讲述的实现线程的具体内容之前，通过理解线程的诸多概念以及它们在.NET 中的构建方式，可以对如何在应用程序中实现这些功能有更好的编程决策。

## 1.1 线程的定义

通过本节的介绍，可以理解以下内容：

- 什么是多任务，以及多任务的不同类型
- 进程的概念
- 线程的概念
- 什么是主线程

- 什么是辅线程

## 1.1.1　多任务

多任务这个术语是指操作系统一次运行多个应用程序的能力。例如，在作者编写本章的内容时，打开了两个 Microsoft Word 窗口和一个 Microsoft Outlook 窗口。另外，系统面板显示出在系统后台还运行有其他的应用程序。当来回切换应用程序时，就可以看出在同一时刻所有这些应用程序都在执行着。在这里使用“应用程序”这个词并不是很恰当，其实我们指的是进程。在本章后面的内容里，将更清楚地定义“进程”这个词。

传统上，多任务有两种不同的风格。目前，Windows 在线程中只使用了一种格式，本书会详细讨论这一内容。我们也会介绍多任务早先的类型，以更好地理解当前方法与以前类型的不同以及优点。

在 Windows 的早期版本(如 Windows 3.x)以及其他一些操作系统中，系统允许运行一个程序，直到该程序将占用的处理器资源释放给正在运行的其他应用程序为止，从而实现协作。因为这种方式是由应用程序与其他正在运行的程序协作，所以称为协作式多任务。这种多任务的缺点是，如果一个程序不释放处理器资源，其他应用程序就会被锁定。而实际上，正在运行的应用程序挂起，其他的程序按次序等候。这就像在银行里排队一样，出纳员一次只对一个顾客提供服务，在完成所有的事务处理之前，顾客不可能离开出纳窗口。完成了这个顾客的所有事务处理之后，出纳员才能为队列中的下一位顾客提供服务。即使该顾客只是想存一张支票，也必须排队等候，直到他前面的顾客将手里的 5 件事务处理完之后才行。

现在，使用 Windows 的当前版本(2000 和 XP)就不会遇到这样的问题了。操作系统现在处理多任务的方法有很大的不同。操作系统允许一个应用程序执行一段很短的时间，然后强制中断它，让另一个应用程序执行。这种中断式的多任务风格称为抢先式多任务机制(pre-emptive multitasking)。抢先式可定义为中断一个应用程序，允许另一个应用程序执行。要注意的是一个应用程序可能还没有完成它的任务，操作系统就让另一个应用程序获得处理器时间。前面的银行出纳员的例子在这里就不适合了。在现实世界中，这就像银行的出纳员暂停处理一个顾客的事务，让另一个顾客开始办理他的业务一样。这并不意味着下一个顾客就可以完成他的事务。出纳员可能在未完成下一个顾客的业务时就中断，开始下一个客户的业务——最终恢复被中断的第一个客户的事务。这种情况很像是人们的大脑处理社交和各种其他任务。抢先式多任务机制解决了处理器被锁定的问题，但同时带来了其他问题。一些应用程序会共享诸如数据库连接和文件之类的资源。如果两个应用程序同时访问同一资源会如何呢？一个程序修改了数据，然后被中断，让另一个程序也对这个数据进行修改。现在两个应用程序都修改了同一个

数据，它们都假定自己是以独占的方式来访问数据，如图 1-1 所示。

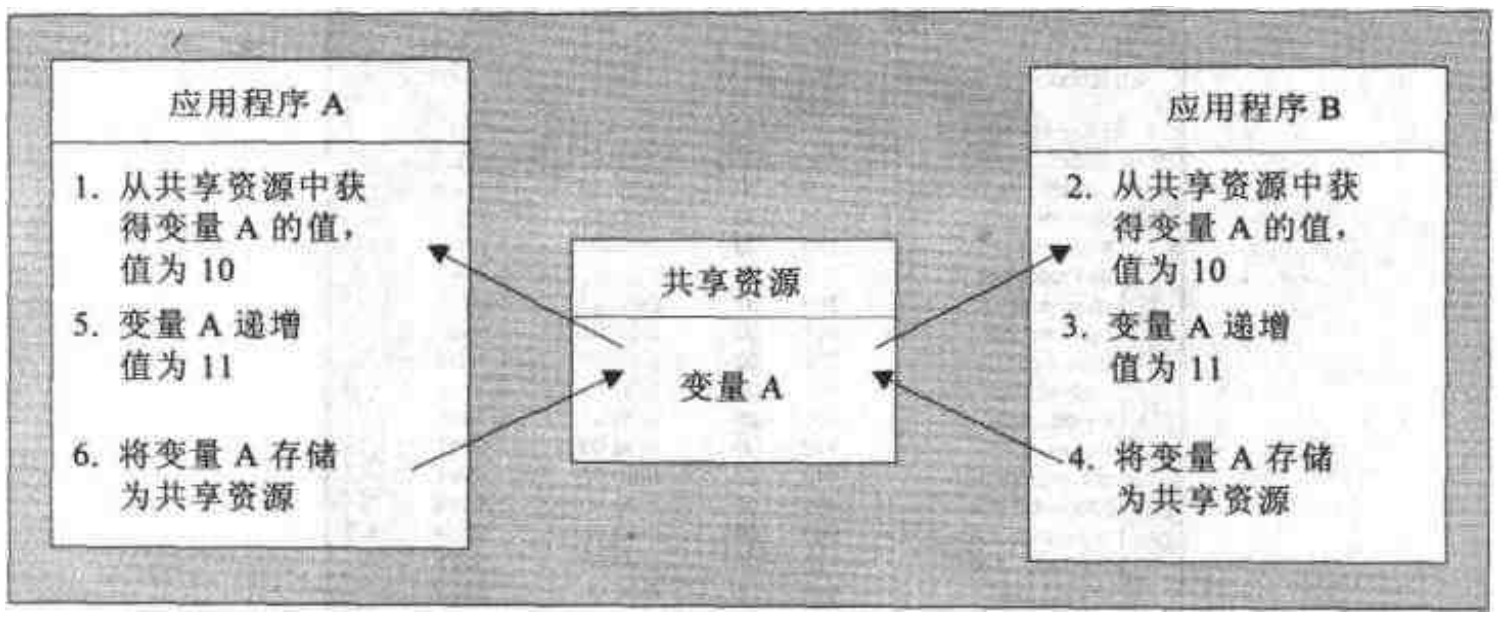


图　1-1

在步骤 1 中，应用程序 A 从一个数据存储器中获取了一个整数值，并将它放到内存中。该整数变量设置为 10。接着，应用程序 A 被中断，强制等候应用程序 B。然后，开始执行步骤 2，应用程序 B 获取同样的整数值 10。在步骤 3 中，应用程序 B 将该数值递增为 11，并在随后的步骤 4 中将变量存储到内存中。在步骤 5 中，应用程序 A 也递增这个值。然而，由于两个应用程序引用这个值时，这个值都是 10，在应用程序 A 完成其递增例程时，该值仍旧是 11。而我们希望这个值设置为 12。两个应用程序都不知道对方也在访问这个资源，于是它们试图递增的那个变量就有了一个错误的值。如果这是一个引用计数器或是一个预订飞机票的售票代理，又会发生什么事情呢？

使用同步技术可以解决抢先式多任务的相关问题，该内容详见第 3 章。

## 1.1.2　进程

当启动应用程序时，系统就会为该应用程序分配所需的内存以及其他资源。内存和资源的物理分离叫作进程。当然，应用程序可以启动多个进程。注意，“应用程序”和“进程”并不是同义词。分配给进程的内存与为其他进程分配的内存被隔离，只有所属的那个进程才可以访问它。

在 Windows 中，通过访问 Windows 任务管理器，可以看到当前正在运行的进程。右击任务栏中的空白空间，选择 Task Manager，打开如图 1-2 所示的 Windows Task Manager 窗口，它包含了 3 个标签：Applications、Processes 和 Performance。Processes 选项卡显示了进程的名称，进程的 ID(PID)号，CPU 使用率、迄今为止线程占用的处理器时间、应用程序使用的内存量。基于方便的原因，在 Windows Task Manager 窗口中应用程序和进程在不同的选项卡中显示。应用程序可能包含一个或多个进程。每个进程都有自己独立的数据、执行代码和系统资源。

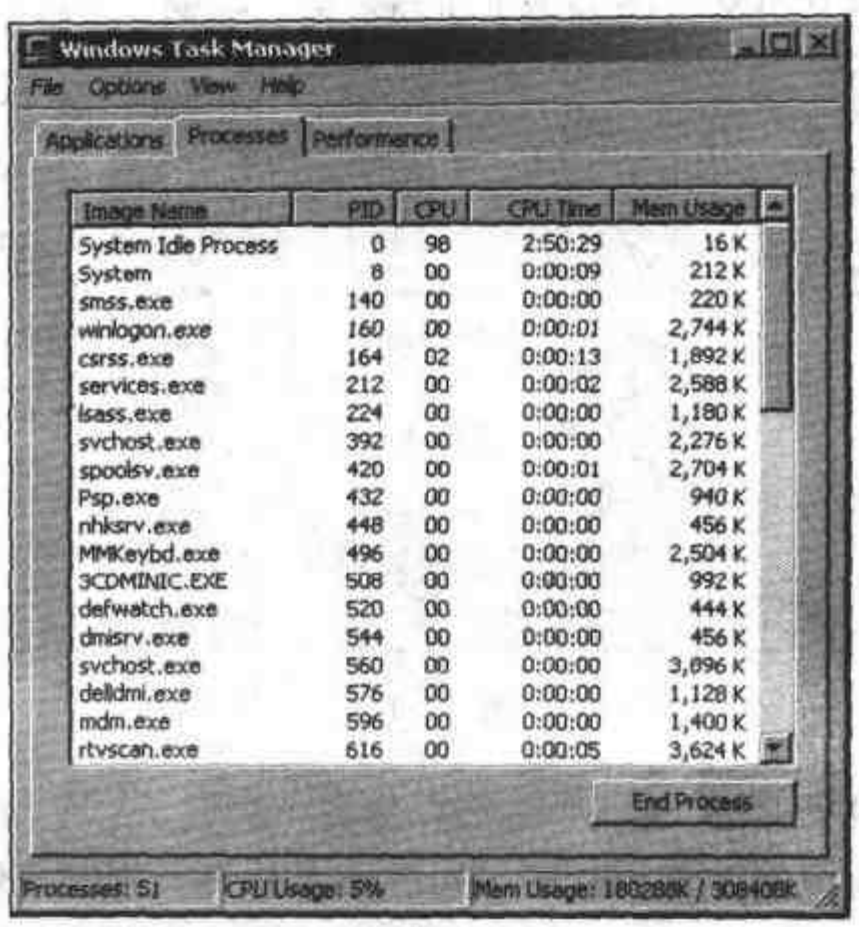


图 1-2

## 1.1.3 线程

在 Windows Task Manager 窗口中，还包含了进程使用 CPU 的汇总信息。这是因为进程也有一个由计算机的处理器使用的执行次序。这个执行次序就称为线程。进程由寄存器定义，记录 CPU 的使用情况、线程使用的堆栈以及跟踪线程当前状态的容器。这个容器就叫作线程本地存储区(Thread Local Storage)。对于惯常处理诸如内存分配这样的底层问题的用户来说，应该比较熟悉寄存器和堆栈的概念。然而，这里需要知道的是，.NET Framework 中的堆栈就是内存的一个区域，用于快速访问，存储值类型或对象的指针、方法参数以及每个方法调用在本地使用的其他数据。

### 1. 单线程的进程

如前所述，每个进程至少有一个执行顺序或线程。创建一个进程包括在指令中的某一点启动进程。这个最初的线程称为基本线程或主线程。线程的实际执行顺序是由应用程序中的方法代码来决定的。例如，在一个简单的.NET Windows Forms 应用程序中，主线程是在项目的静态方法 Main()中启动的。它最先以调用 Application.Run()开始。

清楚了什么是进程，并知道每个进程至少有一个线程后，下面在图 1-3 中给出这种关系的可视模型。

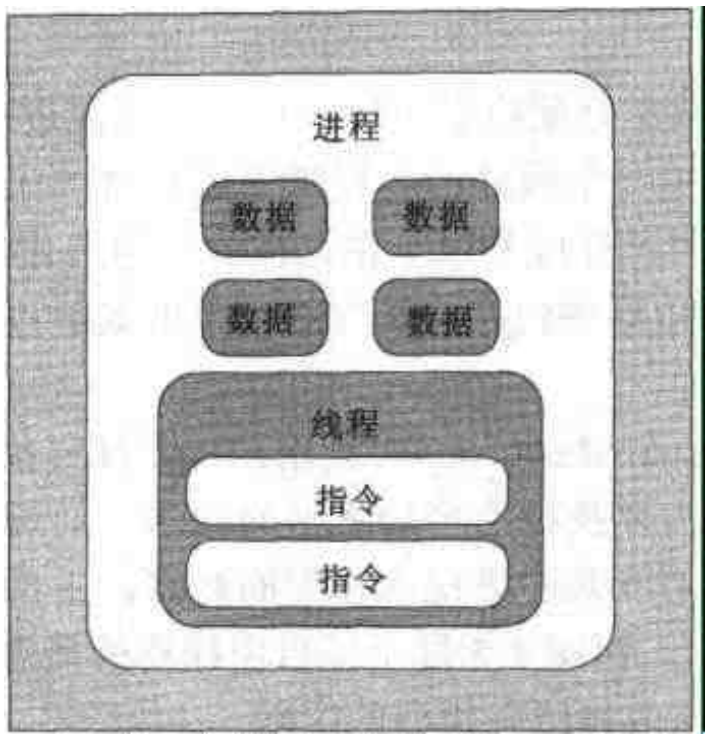


图 1-3

在图 1-3 中，线程和数据一样放在同一隔离区域，说明在这个进程中声明的数据可以通过该线程访问。在需要的时候，线程在处理器上执行并使用进程中的数据。这些看起来都很简单。进程是被物理隔离开了，其他的进程都不能修改其数据。就该进程而言，它是运行在系统上的惟一进程。对于进程的正常运行来说，我们不需要知道其他的进程以及与它们相关联的线程的过多细节。

**注意：**

*更准确地讲，线程其实是指向进程的指令流部分的一个指针。线程实际上不包含指令，只是指出了当前和将来可能要使用的路径，而这是通过数据和分支判断确定的指令来完成的。*

### 2. 时间片

当讨论多任务时，操作系统为每个应用程序都授权了一个时间段，让应用程序在这个时间段中运行，之后就中断该应用程序的执行，让另一个应用程序执行。这样说不太准确，实际上是处理器给进程授予时间。进程能够执行的时段称为时间片或时间量。程序员并不知道这个时间片，除操作系统之外，谁也无法预计时间片。程序员不应将时间片看作应用程序中的一个常量。每个操作系统和处理器可能被分配了不同的时间片。

不过，本章在前面提到过并发的一个潜在问题，如果每个进程都进行物理隔离，就应考虑如何安排它们的执行。这是问题的开始，也是本书剩余部分的重点。一个进程至少要有一个执行线程。进程可能有多个任务需要在一个时间点上执行。例如，在绘制用户界面的同时，可能还需要通过网络访问 SQL Server 数据库。

### 3. 多线程的进程

可以将进程分解，以共享分配给它的时间片。通过在进程中产生额外的执行线程，就可以分解进程。可以产生一个额外的线程来完成后台的工作，例如访问网络或查询数据库。这些辅线程常常用于完成某项工作，因此称为工作线程(worker thread)。这些线程将共享进程的内存空间(这些内存空间与系统上的其他进程隔离开来)。在进程中产生的新线程就称为自由线程。

自由线程的概念和 Visual Basic 6.0 中使用的单元线程模型有很大的区别。在单元线程中，每个进程都拥有需要执行的全局数据的副本。而每个生成的线程都是在它本身的进程中产生的，所以不能共享进程内存中的数据。下面比较一下这两个模型。图 1-4 演示了单元线程的概念，而图 1-5 展示了自由线程的概念。我们不打算在这方面进行过多的介绍，只是简述这两种线程模型的区别。

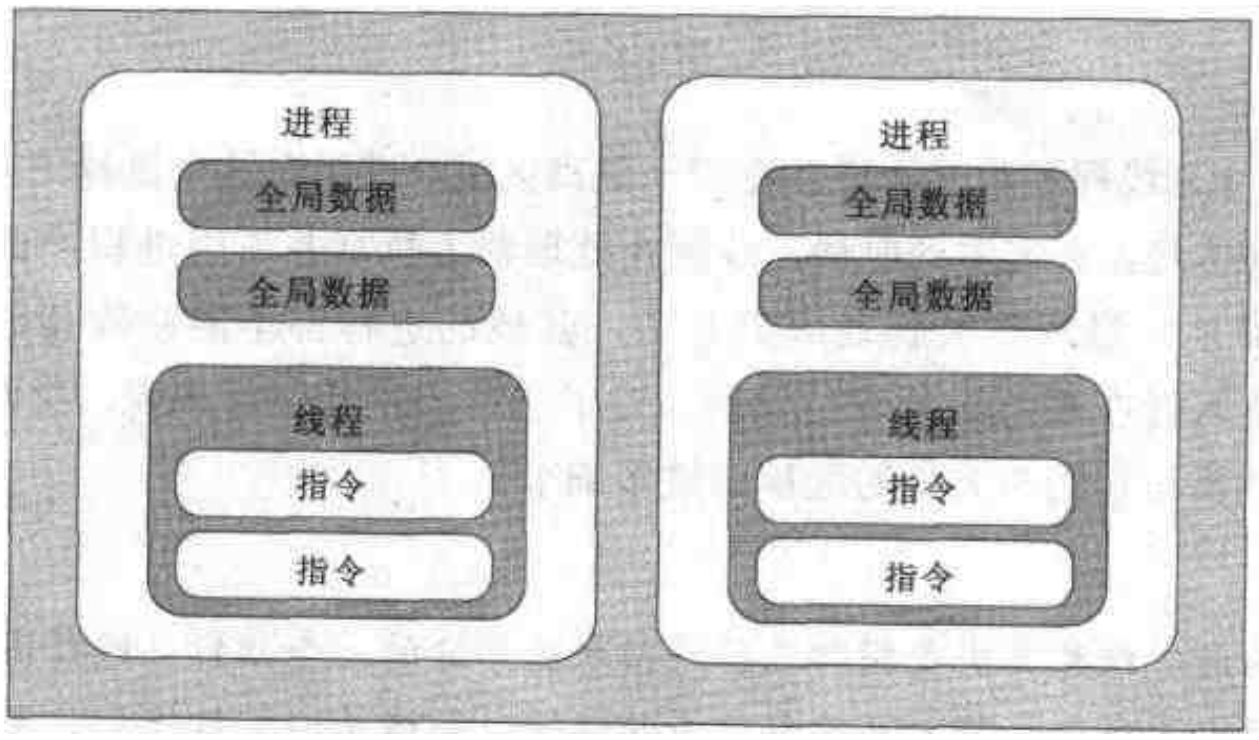


图　1-4

每次希望做一些后台工作时，这些工作都是在其本身的进程中完成的。这就是称之为进程外运行的原因。这种模型与图 1-5 所示的自由进程模型有着很大的区别。

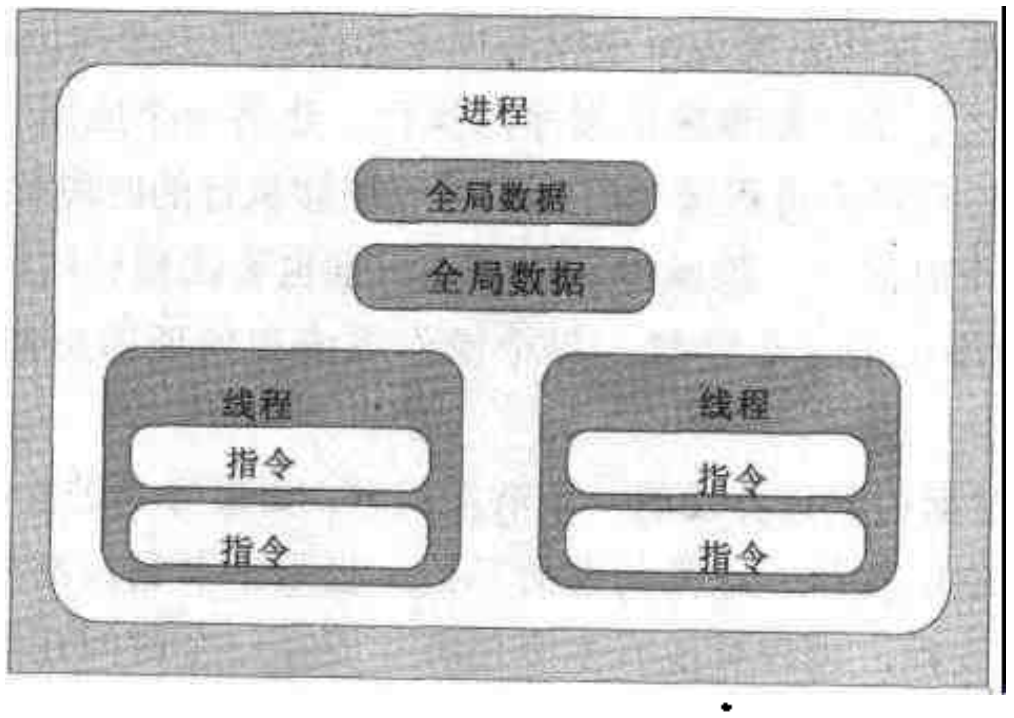


图　1-5

可以让 CPU 使用同一个进程的数据执行一个额外的线程。这远远优于单线程的单元，除了拥有一个额外的线程之外，还可以共享相同的数据。但要注意，处理器一次只能执行一个线程。该进程中的每个线程都被分配了一部分执行时间，以完成它的工作。图 1-6 显示了线程的工作原理。

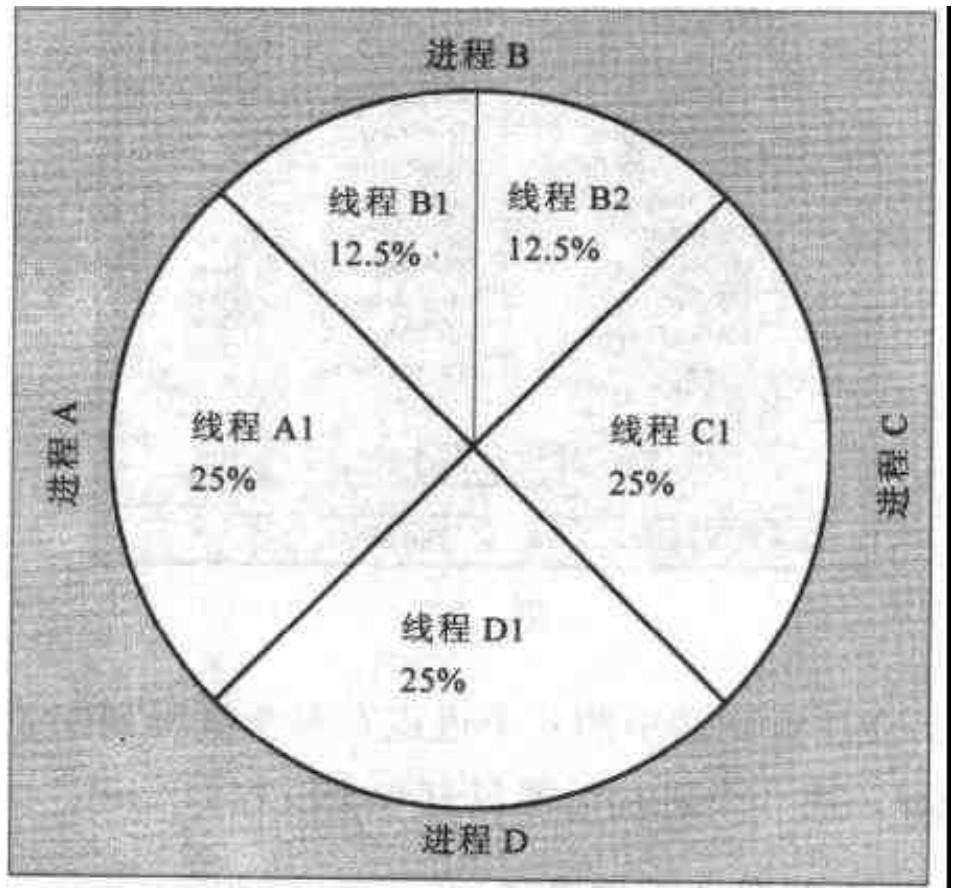


图 1-6

在本书中，所有的例子和图表都假定计算机只有一个处理器。但是，如果计算机使用了多个处理器，使用多线程的应用程序将获得更高的性能。操作系统目前有两处地方可以发送线程的执行。在前面的那个银行例子中，这就类似于另一个出纳员对另一个队列营业。操作系统负责确定哪些线程在哪个处理器上执行。不过，如果程序员有多个 CPU 可供选择，.NET 平台还可以控制进程使用哪个 CPU。通过 System.Diagnostics 命名空间的 Process 类的 ProcessorAffinity 属性，就可以完成这项工作。这是在进程级别上设置的，该进程中的所有线程都将在同一个处理器上执行。

这些线程的调度要比图 1-6 所显示的复杂得多，不过对于要说明的问题来说，这个模型就足够了。每个线程都是依次执行的，很容易让人想起排成一队，等候银行出纳员进行业务处理。但是，这些线程会在很短的一段时间之后中断，接着就执行下一个线程，这可能是同一个进程中的线程，也可能是另一进程中的线程。下面介绍一下 Task Manager。

启动 Task Manager，返回 Processes 选项卡。之后，打开 View | Select Columns 菜单，在 Task Manager 中就会显示一组列。这里我们只关心一个额外的列，即 Thread Count 选项。选中 Thread Count 复选框，如图 1-7 所示。

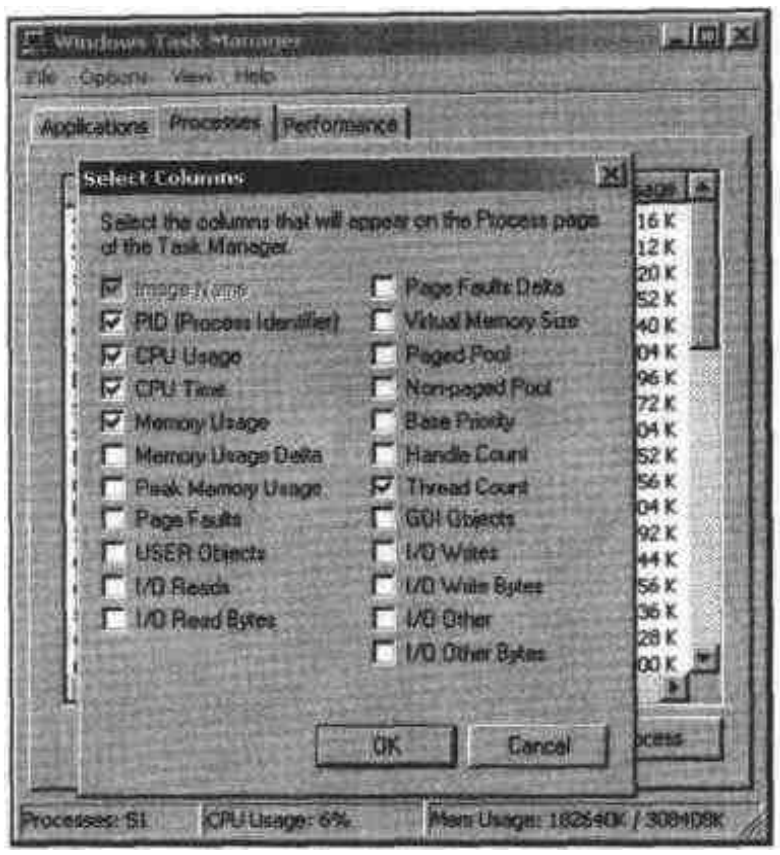


图　1-7

单击 OK 按钮后，会注意到其中的几个进程有多个线程列在 Thread Count 中。这说明程序可以有许多线程，而一个进程可能只有一个线程。

**4. 中断与线程本地存储器的工作原理**

当一个线程运行的时间超出了为它分配的时间片时，它并不是停下来等待。每个处理器一次只能处理一个任务，所以当前的线程必须摆脱这种情况。但是，在当前线程再次跳出队列之前，必须存储状态信息，才能在下次执行时从本次中断的位置开始。这是线程本地存储器(Thread Local Storage，TLS)的一项功能。线程的 TLS 包括了寄存器、堆栈指针、调度信息、内存中的地址空间和其他资源的使用信息。存储在 TLS 中的一个寄存器是程序计数器，该计数器可以告诉线程接下来应执行哪条指令。

**中断**

前面说过，进程并不需要知道同一台计算机上的其他进程。如果是这种情况的话，线程如何知道自己应该给另一个进程让路呢？这个调度决定主要由操作系统来处理。Windows 本身(它是正在处理器上运行的另一个程序)有一个主线程，即系统线程，它负责其他线程的调度。

Windows 知道何时需要使用中断来作出线程调度的决定。我们已经习惯使用中断这个词了，下面精确地定义一下何谓中断。中断是一种机制，它能够使 CPU 指令的正常顺序执行转向计算机内存中的其他地方，而不需要知道目前正在执行什么程序。Windows 决定了线程要执行多长时间，并在当前线程的执行序列中放置一条指令。这个时段在不同的系统上甚至是同一系统上的不同线程之间也会有所不同。中断指令明显地放在指令集中，所以称为软件中断。不要和硬件中断搞混了，硬件中断在所执行的

指令外部发生。一旦放置了中断，Windows 就允许线程执行。当线程发生中断时，Windows 就使用一个特殊的函数，即中断处理程序将这个线程的状态存储在 TLS 中。对于中断之前存储的那个线程来说，当前的程序计数器也存储在 TLS 中。这个程序计数器仅仅是当前执行的指令的地址。一旦线程的执行超时，它就会移到线程队列的末尾，并基于给定的优先级等候再次轮到自己执行。图 1-8 为中断进程的示意图。

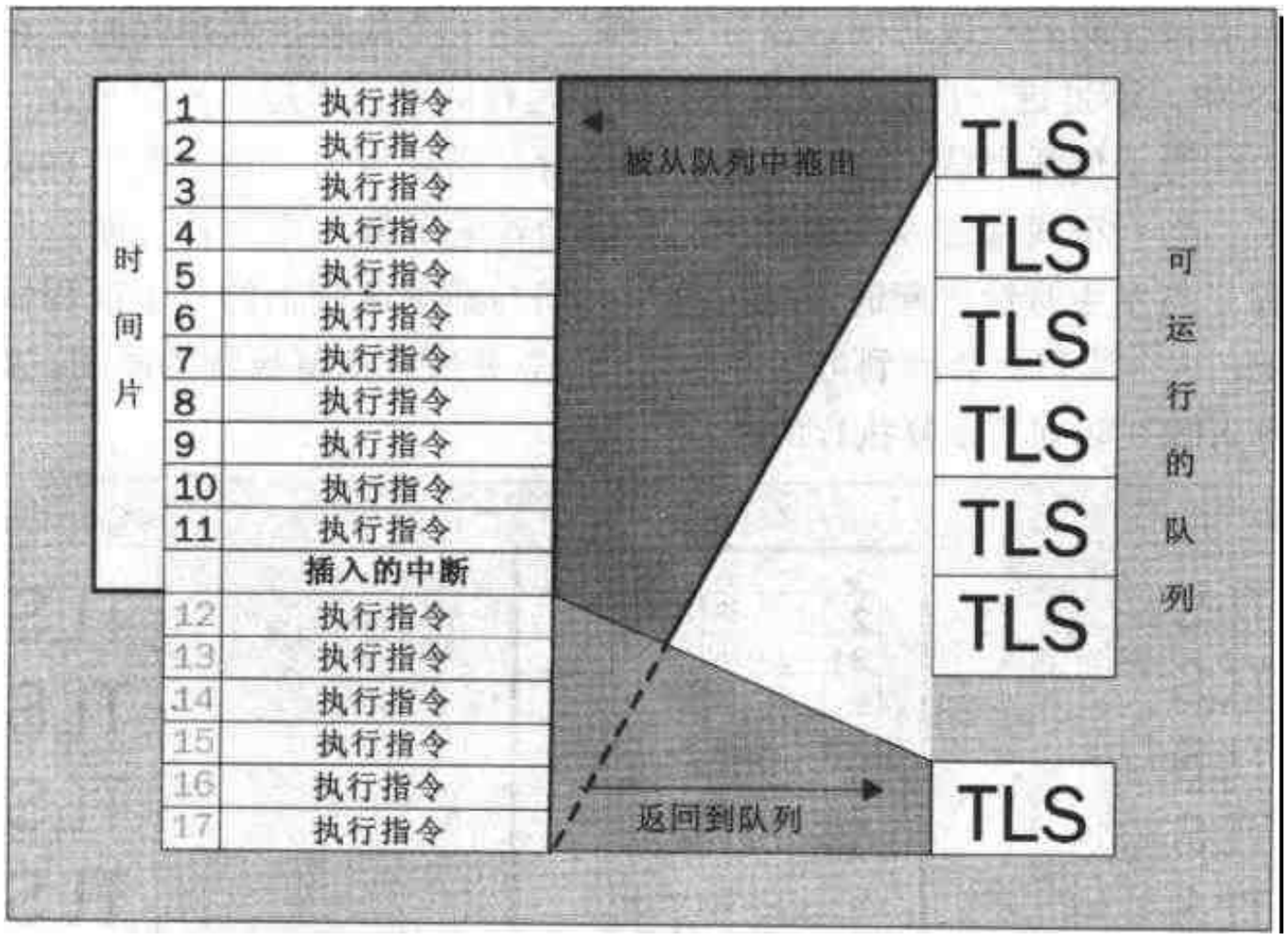


图 1-8

**注意：**

TLS 实际上并没有保存到队列中；而是保存在包含这个线程的进程的内存中。实际上保存到队列中的是指向该内存的指针。

如果线程还没有做任何工作，或线程需要继续执行，这当然好了。可是，如果线程不需要使用分配给它的所有执行时间会怎样呢？在关联切换中的进程(即从一个线程切换到另一个线程)最初有点不同，不过结果是一样的。一个线程在再次执行之前需要等候一个资源，所以，它会给另一个线程让出自己的执行时间。这应由程序员和操作系统来决定，程序员给线程发出信号让它退让，线程则清除掉 Windows 已放在它堆栈中的所有中断，这就模拟了一个软件中断。线程存储在 TLS 中，并像以前那样移到队列的末尾。这非常容易理解，与图 1-8 非常相似，这里就不用图来说明这个概念了。只是要注意，Windows 已经在线程的堆栈上放置了一个中断，在线程被打包之前必须清除它，否则当线程再次执行时，它可能会被过早地中断。当然，其中的细节很抽象。程序员不必亲自清除这些中断。

**线程睡眠与时钟中断**

如前所述，程序可以将自己的执行时间让给另一个线程，以等待某些外部资源。可是，当线程下一次恢复执行时，资源可能还是无法使用。事实上，在线程接下来的 10 次或 20 次执行中，资源可能仍旧无法使用。此时，程序员就希望使线程退出执行队列较长时间，这样，处理器就不必在从一个线程到另一个线程的来回切换中浪费时间。线程自动退出执行队列一段时间，就称为睡眠。当一个线程进入睡眠时，它会再次被打包到 TLS 中，不过这一次，TLS 并没有放在运行队列的末尾，而是放在一个独立的睡眠队列中。为了使睡眠队列上的线程再次运行，需要用另一种中断来标记它们，称为时钟中断。当一个线程进入睡眠队列时，就会给它设置一个时钟中断，安排何时唤醒这个线程。当发生时钟中断时，如果这个时间与睡眠队列上的一个线程恢复运行的时间相匹配，这个线程就会回到可运行的队列中，并安排恢复执行的时间。如图 1-9 所示为线程进入睡眠队列与恢复执行的过程。

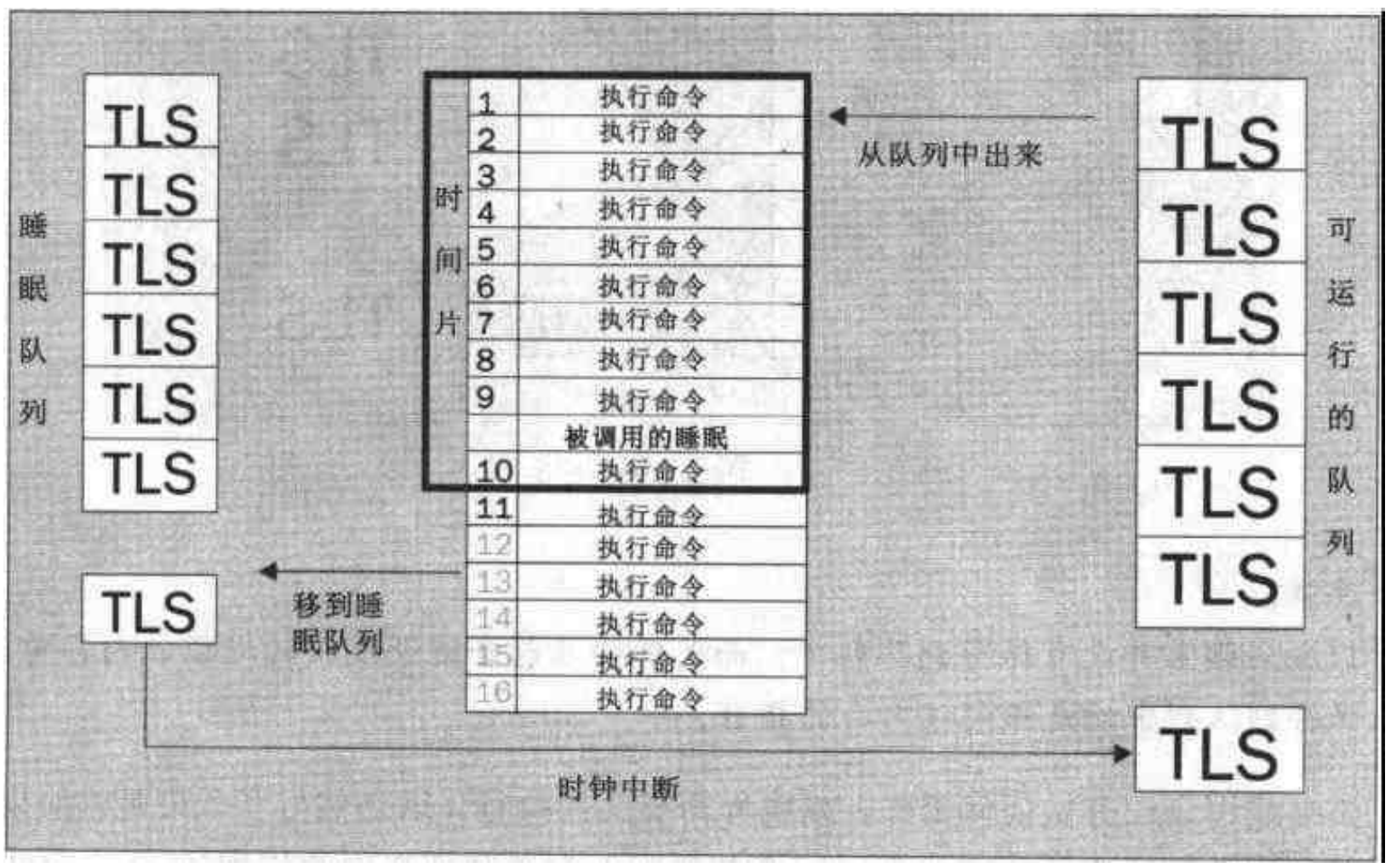


图　1-9

**线程的终止**

现在我们已经了解了线程的中断和线程睡眠。可是，像生活中的其他事物一样，线程也必须终止。在执行另一个线程的过程中，可以用一个请求显式停止线程的执行。当以这种方式结束线程时，就称为终止。当执行到线程的结尾时，线程也会停止。无论哪种情况，当线程结束时，线程的 TLS 就会释放其内存。但线程所使用的进程中的数据仍然存在，除非进程也结束了，这一点很重要，因为进程中可能有多个线程访问该数据。线程不能自我终止，必须通过另一个线程来终止。

**线程的优先级**

前面介绍了如何中断一个线程，以便执行另一个线程。还介绍了线程如何通过退出执行或使自己进入睡眠来让出自己的执行时间。另外，还介绍了线程的终止。最后一个有关线程的基本概念是，线程如何设定优先级。通过生活中一些类似的经验，我们明白，有一些任务是优先于其他任务的。例如，对于本书的作者来说，完成这本书的创作有一个紧迫的截止期限，但是作者也要吃东西啊。吃东西优先于编写本书，因为人是铁，饭是钢。另外，如果作者为编写本书而开夜车，睡觉的优先级就要提高。其他人也可能会给作者指派额外的任务。但是，这些额外的任务不会具有最高的优先级。某人可以强调某个任务非常重要，但最终还是由接受任务的人来确定哪个任务最重要，哪个任务可以稍后再做。

上述信息包含了很多理论和类比，这与线程的概念有着非常紧密的联系。有些线程需要高一点的优先级。吃东西和睡觉具有很高的优先级，因为它们使我们能正常地工作，系统任务具有较高的优先级，因为计算机需要它们执行所需的功能。Windows 将线程的优先级分为 0～31 个级别，数字越大表示优先级越高。

只有系统才能设置 0 优先级，它说明线程是空闲的。Windows 系统的用户可以设置 1～15 的优先级。只有系统管理员才能设置高于 15 的优先级。稍后介绍系统管理员如何设置高于 15 的优先级。以 16～31 之间的优先级运行的线程是在实时运行的线程。所谓“实时”，指的是优先级非常高，可以抢占以较低优先级运行的线程的执行时间。这种抢占可以使线程立即执行。需要实时运行的项目有设备驱动程序、文件系统和输入设备等进程。设想一下，如果键盘和鼠标输入在系统中没有高的优先级将会发生什么事情。用户级线程的默认优先级是 8。

最后要注意的是，线程继承了它们所属的进程的优先级。图 1-10 说明了这一点，将来还将使用这个图进一步分解这些数字。

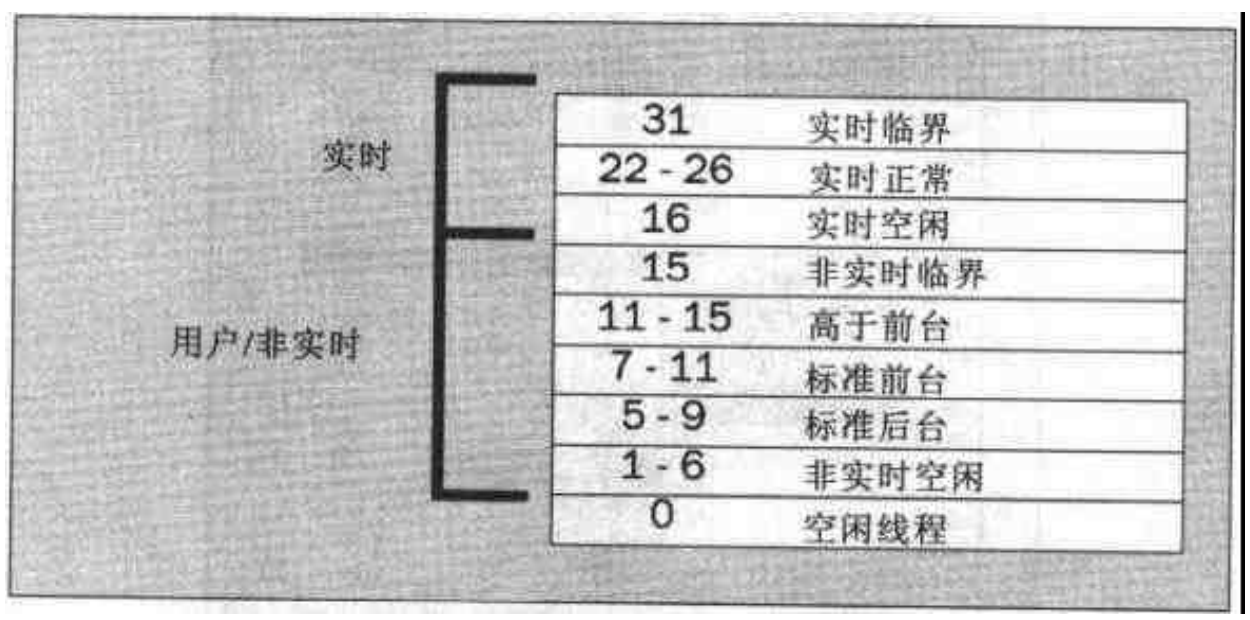


图 1-10

在一些操作系统中，例如 Windows，只要存在一个优先级较高的线程，就不会安

排优先级较低的线程执行。处理器将以优先级最高则最先执行的原则来调度所有的线程。同一优先级的线程将以循环的方式依次执行。优先级最高的所有线程都执行完之后，就执行仅次于其优先级的线程。如果一个优先级较高的线程再次可用了，就会抢占优先级较低的线程，把处理器的使用时间分配给这个优先级较高的线程。

**管理优先级**

基于我们对优先级的了解，可以为某个进程设置较高的优先级，使该进程中生成的线程执行的可能性更高。Windows 提供了几种方式来编程设置任务的优先级，并能对这些优先级进行管理。下面主要讨论如何管理优先级。使用 Task Manager、pview(和 Visual Studio 一起安装)和 pviewer(安装为 Windows NT 的一个资源工具集或直接和 Windows XP 企业版一起安装)等工具可以完成这项工作。使用 Windows 的性能监控器还可以查看当前进程的优先级。这里不深入介绍这些工具，只是简要地介绍一下如何为进程设置普通的优先级。在第一次介绍进程时，启动了 Task Manager，来查看当前运行在系统上的所有进程。而在 Windows Task Manager 窗口中，就可以提高某个进程的优先级。

接下来试着修改进程的优先级。首先，打开一个应用程序实例，如 Microsoft Excel。现在启动 Task Manager，进入如图 1-11 所示的 Processes 选项卡。在该选项卡中，可以看到 Excel 实例运行为一个进程。在列表框中右击 Excel.EXE，从弹出的快捷菜单中选择 Set Priority 命令，可以根据自己的需要修改优先级别。将 Excel 的优先级设置为 high 并没有意义，只是可以这样设置。每个进程都有一个优先级，操作系统不会告诉用户该设置什么优先级，如果用户要做一些不合逻辑的事，操作系统就会提出警告，不过选择什么优先级仍由用户决定。

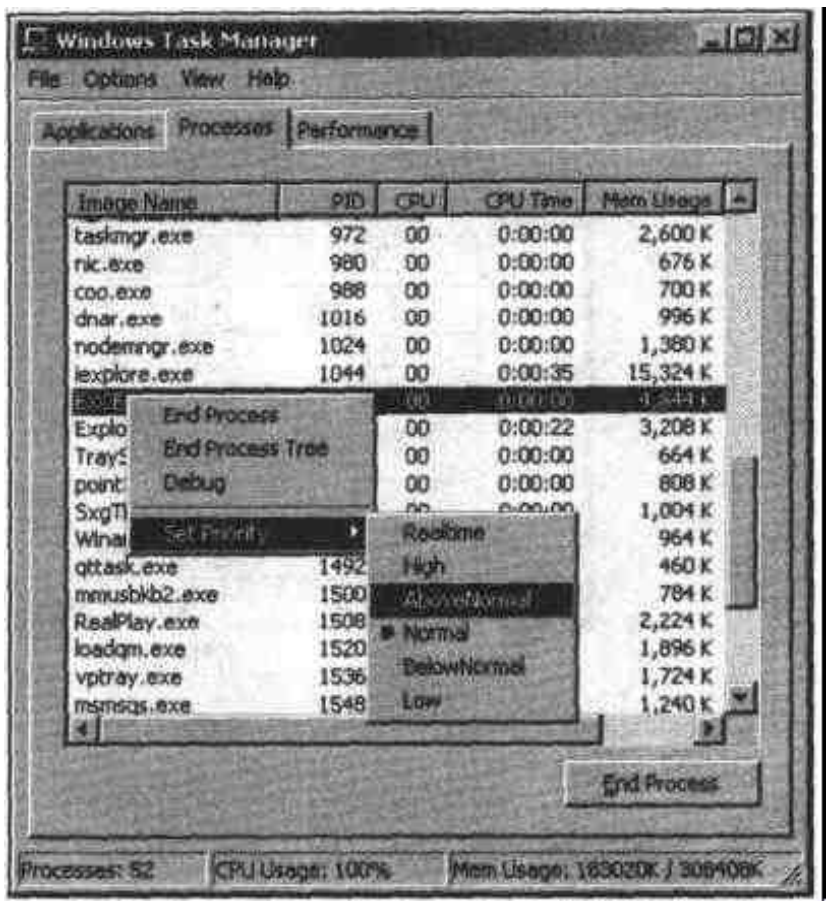


图　1-11

在图 1-11 中，在一个优先级旁边有一个标记。这个标记表示进程当前的优先级。当为一个进程设置优先级时，实际上只是为应用程序的一个实例设置这个标记。这意味着，同一个应用程序的其他当前运行的实例保持其默认的进程级别。另外，该进程今后启动的实例也具有默认的进程级别。

## 1.2 .NET 和 C#对线程的支持

.NET Framework 支持自由线程，所以在所有的.NET 语言中都可以使用自由线程，包括 C#和 VB.NET。下一节将介绍 Microsoft 是如何提供对线程的支持的，深入论述线程的工作原理，并讨论 Microsoft 提供的一些额外支持，以便更好地把线程与进程区分开。

通过本节的介绍可以了解以下内容：

- 什么是 System.AppDomain 类，可以使用该类做什么
- .NET 运行库是如何监控线程的

### 1.2.1 System.AppDomain 类

本章前面在介绍进程时说过，进程是内存和资源的一个物理独立的部分。后来又提到，一个进程至少有一个线程。当 Microsoft 设计.NET Framework 时，新增了一个隔离层，称为应用程序域或 AppDomain。应用程序域不像进程那样是一个物理独立的部分，而是进程内部的一个逻辑独立部分。在一个进程中可以存在多个应用程序域，这有一些重要的优点。通常，标准的进程不使用代理，就不可能访问其他进程的数据。使用代理会耗费很多系统开销，编码也会变得较复杂。可是，如果引入应用程序域的概念，就可以在同一个进程中启动好几个应用程序。进程所提供的同一个隔离区域可以和应用程序域一起使用。线程可以跨多个应用程序域来执行，且不需要耗费因线程内部通信而产生的系统开销。这些额外的进程内边界的另一个好处是，它们提供了对所包含的数据的类型检查功能。

Microsoft 把这些应用程序域的所有功能都封装在一个名为 System.AppDomain 的类中。Microsoft .NET 程序集与这些应用程序域有着非常密切的关系。只要将程序集加载到应用程序中，它就会载入 AppDomain。除非特别指出，否则程序集就会被加载到调用代码的 AppDomain 中。应用程序域与线程也有直接的关系，它们可以保存一个或多个线程，就像进程一样。可是不同之处在于，应用程序域可以在进程内部创建，但不创建新线程。应用程序域和进程以及线程的关系如图 1-12 所示。

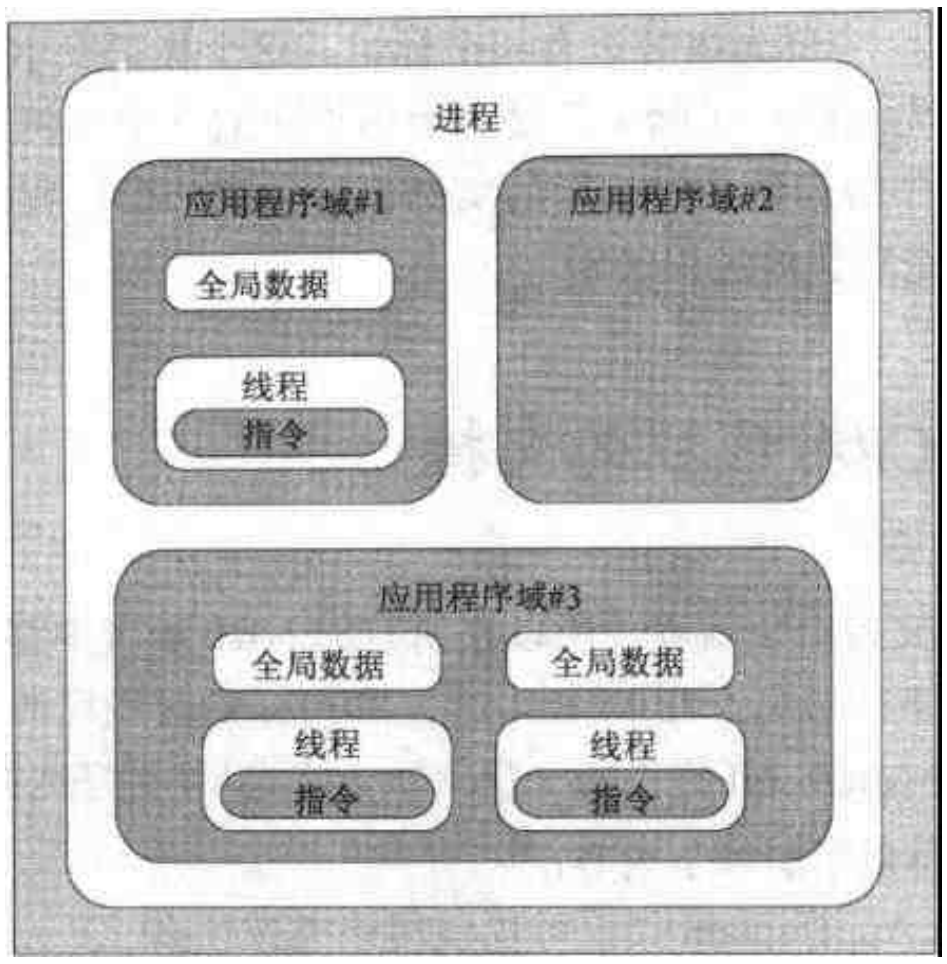


图　1-12

**注意：**

在.NET 中，出于安全的原因，不能继承 AppDomain 类和 Thread 类。

每个应用程序都包含了一个或多个 AppDomain。每个 AppDomain 都可以创建并执行多个线程。如图 1-13 所示，机器 X 中有两个正在运行的 OS(操作系统)进程 Y 和 Z。OS 进程 Y 有 4 个正在运行的 AppDomain，即 A、B、C 和 D。OS 进程 Z 则包含了两个 AppDomain，即 A 和 B。

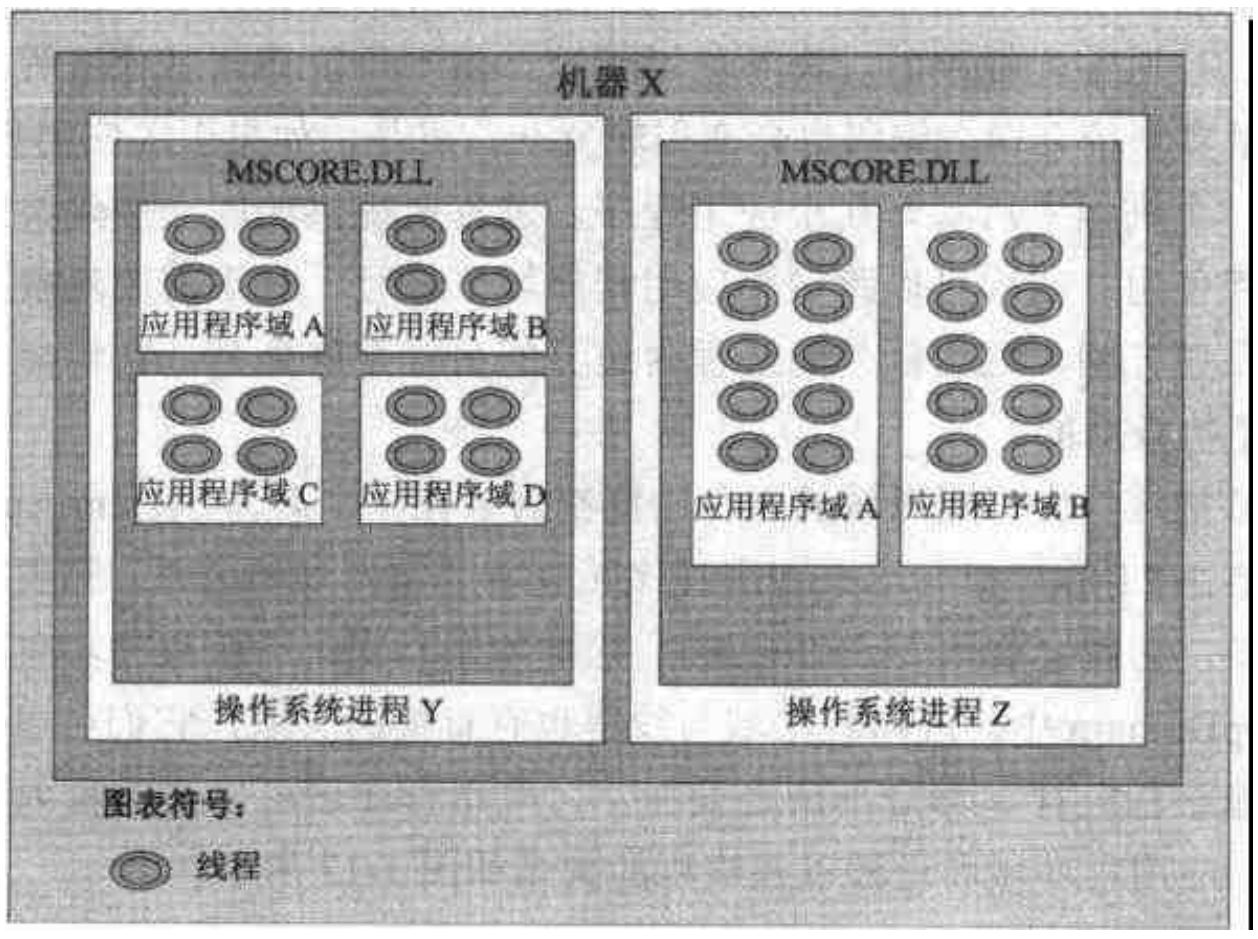


图　1-13

### 1. 设置 AppDomain 数据

了解了有关的理论和模型后，现在介绍一些实际的代码。下面的示例将使用 AppDomain 来设置数据、检索数据以及标识 AppDomain 在执行的线程。创建一个新的类文件 appdomain.cs，并输入如下代码：

```
using System;

public class MyAppDomain
{
   public AppDomain Domain;
   public int ThreadId;

public void SetDomainData(string vName ,string vValue)
{
   Domain.SetData(vName, (object)vValue) ;
   ThreadId = AppDomain.Get CurrentThreadId();
}

public string GetDomainData(string name)
{
   return (string)Domain.GetData(name) ;
}
public static void Main()
{
   string DataName = "MyData";
   string DataValue = "Some Data to be stored";

   Console.WriteLine("Retrieving current domain");
   MyAppDomain Obj = new MyAppDomain();
   Obj.Domain = AppDomain.CurrentDomain;

   Console.WriteLine("Setting domain data");
   Obj.SetDomainData(DataName, DataValue);

   Console.WriteLine("Getting domain data");
   Console.WriteLine("The Data found for key ' " + DataName
                     + " ' is ' " + Obj.GetDomainData(DataName)
                     + " ' running on thread id: " + Obj.ThreadId);
  }
}
```

执行程序后，得到如下输出：

```
Retrieving current domain
Setting domain data
Getting domain data
The Data found for key 'MyData' is 'Some Data to be stored' running on thread id: 1372
```

这对于即使没有经验的 C#开发人员来说也非常简单。下面看一下代码，说明发生了什么。这个类中第一段重要的代码如下所示：

```
public void SetDomainData (string vName , string vValue)
{
  Domain.SetData(VName, (object)vValue);
 ThreadId = AppDomain.GetCurrentThreadId();
}
```

这个方法的参数是要设置的数据名称及其值。注意当 SetData()方法传入参数时，做了一些与众不同的工作。这里将 String 类型的值强制转换为 object 数据类型，因为 SetData()方法把 object 数据类型作为其第二个参数。该方法只使用了一个 String，而 String 由 System.Object 派生而来，所以不用将这个变量强制转换为 object 数据类型。不过，要存储其他数据，就没这么容易处理了，进行这个转换只是提醒一下。在这个方法的最后，调用 AppDomain 对象的 GetCurrentThreadId 属性，就可以得到当前运行的 ThreadId。

接下来的一个方法：

```
public string GetDomainData(string name)
{
  return (string)Domain.GetData(name) ;
}
```

这个方法非常简单，这里使用 AppDomain 类的 GetData()方法，根据键值获取数据。在这个例子中，把参数从 GetDomainData()方法传递给 GetData()方法，再把该方法的结果传递回调用方法。

最后看看 Main()方法：

```
public static void Main()
{
  string DataName = "MyData";
  string DataValue = "Some Data to be stored";

  Console.WriteLine("Retrieving current domain");
```

```
    MyAppDomain Obj = new MyAppDomain();
    Obj.Domain = AppDomain.CurrentDomain;

    Console.WriteLine("Setting domain data");
    Obj.SetDomainData(DataName, DataValue) ;

    Console.WriteLine("Getting domain data") ;
    Console.WriteLine("The Data found for key ' " + DataName
                      + " ' is ' " + Obj.GetDomainData(DataName)
                      + " ' running on thread id: " + Obj.ThreadId);
}
```

首先初始化要在 AppDomain 中存储的名称和值对，在控制台上写一行代码，表示方法开始运行了。接着，引用当前执行的 AppDomain 对象(就是在 Main()方法中执行的 AppDomain 对象)，设置类的 Domain 字段。然后调用方法，把两个参数传送给 SetDomainData()方法：

```
obj.SetDomainData(DataName, DataValue);
```

之后，把一个参数传递给 GetDomainData()方法，获取刚才设置的数据，并把它插入控制台输出流。再输出类的 ThreadId 属性，看看在所调用的方法中正在执行的 ThreadId 是什么。

### 2. 在指定的应用程序域中执行代码

现在介绍如何创建新的应用程序域，着重介绍在新建的 AppDomain 中创建线程时的有关行为。下面是 create_appdomains.cs 文件中的代码：

```
using System;

public class CreateAppDomains
{
   public static void Main()
   {
      AppDomain DomainA;
      DomainA = AppDomain.CreateDomain("MyDomainA");
      string StringA = "DomainA Value";
      DomainA.SetData("DomainKey", StringA);

      CommonCallBack();
```

```
    CrossAppDomainDelegate delegateA =
        new CrossAppDomainDelegate (CommonCallBack) ;
    DomainA.DoCallBack(delegateA);
}

public static void CommonCallBack()
{
    AppDomain Domain;
    Domain = AppDomain.CurrentDomain;

    Console.WriteLine("The value ' " + Domain.GetData("DomainKey") +
                      " ' was found in " + Domain.FriendlyName.ToString() +
                      " running on thread id: " +
                      AppDomain.GetCurrentThreadId().ToString()) ;
  }
}
```

编译该类后的输出结果如下所示：

```
The Value " was found in create_appdomain.exe running on thread id: 1372
The Value 'DomainA Value' was found in MyDomainA running on thread id: 1372
```

在这个示例中，创建了两个应用程序域。为此，调用 AppDomain 类的静态方法 CreateDomain()。构造函数采用的参数是它所创建的 AppDomain 实例的友好名称。以后，经由一个只读属性就可以访问这个友好名称。下面是创建 AppDomain 实例的代码：

```
AppDomain DomainA;
DomainA = AppDomain.CreateDomain("MyDomainA");
```

接下来，调用 SetData()方法。这个方法在上一个示例中介绍过了，这里不再列出其代码。接下来需要解释的是如何让代码在给定的 AppDomain 中执行。为此，可以使用 AppDomain 类的 DoCallBack()方法。该方法使用 CrossAppDomainDelegate 作为参数。在这个示例中，通过给构造函数传递要执行的方法名，来创建 CrossAppDomainDelegate 的一个实例：

```
CommonCallBack();

CrossAppDomainDelegate delegateA =
    new CrossAppDomainDelegate(CommonCallBack);
DomainA.DoCallBack(delegateA);
```

这里首次调用了 CommonCallBack()方法。该方法在主 AppDomain 中执行。在输出中，主 AppDomain 的 FriendlyName 属性是可执行程序的名称。

最后看一下 CommonCallBack()方法的代码:

```
public static void CommonCallBack ()
{
   AppDomain Domain;
   Domain = AppDomain.CurrentDomain;

   Console.WriteLine("The Value ' " + Domain.GetData("DomainKey") +
        " ' was found in " + Domain.FriendlyName.ToString() +
        " running on thread id: " +
        AppDomain.GetCurrentThreadId().ToString()) ;
}
```

这段代码非常通用，无论在什么实例中运行它，它都可以正常工作。这里再次使用 CurrentDomain 属性来获取正在执行代码的域的引用。然后使用 FriendlyName 属性标识正在使用的 AppDomain。

最后，再次调用 GetCurrentThreadId()方法。在程序的输出结果中，无论在哪个 AppDomain 中执行，得到的线程 ID 都是一样的。这一点很重要，因为这不仅意味着 AppDomain 可以有零个或多个线程，而且能够跨不同的域来执行线程。

## 1.2.2 线程管理与.NET 运行库

.NET Framework 不仅提供了对自由线程的进程和逻辑应用程序域的支持，还提供了处理器线程的对象表示。这些对象表示都是 System.Threading.Thread 类的实例，第 2 章将对此进行深入介绍。不过，在进入第 2 章之前，必须先了解托管线程与非托管线程的关系。也就是说，非托管线程(在.NET 外部创建)是如何与托管 Thread 类的实例(表示运行在.NET CLR 内部的线程)发生关联的。

.NET 运行库监控着由.NET 代码创建的所有线程。它还监控着所有可以执行托管代码的非托管线程。因为托管代码可以由能通过 COM 调用的包装器提供，所以非托管线程能够很好地在.NET 运行库中工作。

当非托管代码在托管线程中执行时，运行库就会检查 TLS，确定是否存在托管的 Thread 对象。如果找到了一个托管线程，运行库就会使用那个线程。如果没有找到托管线程，运行库就会创建一个并使用它。这个过程非常简单，却非常必要。无论线程来自何处，我们都希望得到它的对象表示。如果运行库不为这类内部调用管理和创建线程，就不能在托管环境中标识线程，更不用说控制它了。

最后要注意的是，一旦非托管调用返回给非托管代码，线程就不再受运行库的监控了。

## 1.3　本章小结

本章讨论了许多主题，介绍了多任务处理的基本概念以及如何使用线程来完成多任务处理，说明了多任务处理和自由线程并不是一回事。此外，还描述了进程的概念以及它们如何将数据与其他应用程序隔离开来。同时也讲述了在像 Windows 这样的操作系统中线程的功能。Windows 通过中断线程来为其他的线程安排一个较短的执行时间，这段时间称为时间片或时间量。随后，描述了线程优先级的功能以及优先级的不同级别，默认情况下，线程将继承其父进程的优先级。

本章还描述了.NET 运行库如何监控在.NET 环境中创建的线程，以及执行托管代码的非托管线程。我们讲述了.NET Framework 对线程所提供的支持。System.AppDomain 类在物理进程数据隔离区域之上提供了一个额外的逻辑数据隔离层。还描述了线程是如何从一个 AppDomain 跨越到另一个 AppDomain。除此之外，AppDomain 不必像所有的进程那样必须有自己的线程。

# 第2章　.NET中的线程

第 1 章说明了线程是什么，介绍了很多人已经熟悉的基本知识，论述了线程的一些重要内容。本章将介绍如何实现一些基本的线程；但理解何时使用线程同样也很重要。

本章的主要内容如下：

- System.Threading 命名空间
- 在线程的使用中存在的设计问题
- 线程使用的资源
- 使用线程的好时机
- 使用线程时需避免什么错误

## 2.1　System.Threading 命名空间

前面提过，托管代码中的线程由 System.Threading.Thread 类的实例表示。本节将深入讨论 System.Threading 命名空间及其内容。System.Threading 命名空间中可用的类如表 2-1 所示。

表 2-1

| 类 | 说　明 |
| --- | --- |
| AutoResetEvent | 该事件通知一个或多个正等待的线程已发生了一个事件 |
| Interlocked | 该类为多个线程共享的变量提供原子操作，以防止错误的产生 |
| ManualResetEvent | 在通知一个或多个等待的线程已发生了一个事件时，触发该事件 |
| Monitor | 该类提供了同步访问对象的机制 |
| Mutex | 一种同步原语，将对共享资源的排他访问授予惟一的一个线程。它还可以用于进程之间的同步 |
| ReaderWriterLock | 该类定义了一个锁，允许单个写程序和多个读程序有相关联的语义 |
| RegisteredWaitHandle | 该类代表一个在调用方法 RegisterWaitForSingleObject()时已注册的句柄 |
| SynchronizationLockException | 在未同步的代码块中调用同步方法时，会抛出这个异常 |
| Thread | 该类创建并控制线程，设置其优先级，并获得其状态 |
| ThreadAbortException | 当调用 Abort()方法时会抛出这个异常 |

(续表)

| 类 | 说　明 |
| --- | --- |
| ThreadExceptionEventArgs | 该类提供了用于 ThreadException 事件的数据 |
| ThreadInterruptedException | 当处于等待状态的线程被中断时，会抛出这个异常 |
| ThreadPool | 该类提供线程池，可以用来递送工作项、处理异步的输入/输出、代表其他线程等待以及处理计时器 |
| ThreadStateException | 当用于方法调用的线程处于无效状态时，会抛出这个异常 |
| Timeout | 该类包含一个常量，用来指定无限的时间 |
| Timer | 该类提供一种在指定时间间隔内执行方法的机制 |
| WaitHandle | 该类封装了操作系统特定的对象，这些对象等待对共享资源的排他访问 |

这些类不会在本节中全部使用，但是理解 System.Threading 命名空间提供了什么可用的功能是很必要的。其他类在后面的章节中讨论。

## 2.1.1　Thread 类

下面集中介绍一下 Thread 类，因为这个类代表了处理器线程。它可以做任何事情，包括管理线程的优先级，读取线程的状态等。

表 2-2 中列出了 Thread 类的 Public 方法。

表 2-2

| 公共方法的名称 | 说　明 |
| --- | --- |
| Abort() | 这个重载的方法在调用它的线程上产生一个 ThreadAbortException，以并始终止这个线程的进程。调用该方法通常会终止线程 |
| AllocateDataSlot() | 该静态方法在所有的线程上分配一个未命名的数据槽 |
| AllocateNamedDataSlot() | 该静态方法在所有的线程上分配一个已命名的数据槽 |
| FreeNamedDataSlot() | 该静态方法释放先前分配的已命名的数据槽 |
| GetData() | 该静态方法检索当前线程上指定数据槽的数值，检索范围是当前线程的当前域 |
| GetDomain() | 该静态方法返回当前运行的线程所处的当前域 |
| GetDomainID() | 该静态方法返回惟一的应用程序域标识符 |
| GetHashCode() | 该方法用作特定类型的散列函数，适用于散列算法和数据结构，如散列表 |
| GetNamedDataSlot() | 该静态方法查找已命名的数据槽 |

(续表)

| 公共方法的名称 | 说　　明 |
| --- | --- |
| Interrupt() | 该方法中断处于 WaitSleepJoin 线程状态的线程 |
| Join() | 该重载的方法暂停调用线程，直到线程终止 |
| ResetAbort() | 该静态方法取消当前线程请求的 Abort()方法 |
| Resume() | 该方法恢复被挂起的线程 |
| SetData() | 该静态方法在当前线程的当前域中设置指定数据槽中的数据 |
| Sleep() | 该静态重载方法暂停当前线程，暂停的时间为指定的毫秒数 |
| SpinWait() | 该静态方法让线程等待由 iterations 参数定义的一段时间 |
| Start() | 该方法让操作系统将当前线程的状态改为 ThreadState.Running |
| Suspend() | 该方法会挂起线程，如果线程已挂起，就不起任何作用 |

表 2-3 列出了 Thread 类的 Public 属性。

表 2-3

| 公共属性名称 | 说　　明 |
| --- | --- |
| ApartmentState | 设置或获取线程的单元状态 |
| CurrentContext | 该静态属性获得执行线程的当前环境 |
| CurrentCulture | 设置或获取当前线程的文化 |
| CurrentPrincipal | 该静态属性设置或获取线程的当前负责人。它用于基于角色的安全性 |
| CurrentThread | 该静态属性获取当前运行的线程 |
| CurrentUICulture | 在运行期间使用这个属性，它设置或获取由资源管理器使用的当前文化，来查询特定文化的资源 |
| IsAlive | 获取表示当前线程的执行状态的值 |
| IsBackground | 设置或获取表示线程是否为后台线程的值 |
| IsThreadPoolThread | 获取表示线程是否在线程池中的值 |
| Name | 设置或获取线程的名称 |
| Priority | 设置或获取表示线程优先级的值 |
| ThreadState | 获取包含当前线程状态的值 |

本章也不会使用所有这些属性和方法。我们已经见过这些类的成员，但是在创建线程或线程的引用之前，这些成员不会有什么帮助。因此，下面介绍一个简单的 C#线程实例。

## 2.1.2　创建线程

这里介绍一个简单的例子，它并不适合于解释为什么要使用新的线程，但去除了稍后讲到的所有复杂因素。创建一个新的控制台应用程序，把文件命名为 simple_thread.cs，添加以下的代码：

```
using System;
using System.Threading;

public class SimpleThread
{
  public void SimpleMethod()
  {
    int i = 5;
    int x = 10;
    int result = i * x;
    Console.WriteLine("This code calculated the value " +
                      result.ToString() + " from thread ID; " +
                      AppDomain.GetCurrentThreadId().ToString());
  }

  public static void Main()
  {
    // Calling the method from our current thread
    SimpleThread simpleThread = new SimpleThread();
    simpleThread.SimpleMethod() ;

    // Calling the method on a new thread
    ThreadStart ts = new ThreadStart(simpleThread.SimpleMethod);
    Thread t = new Thread(ts);
    t. Start ();
    Console.ReadLine() ;
  }
}
```

保存并编译，然后执行这个文件。输出结果如下：

```
This code calculated the value 50 from thread id: 1400
This code calculated the value 50 from thread id: 1040
```

下面解释一下这个简单的示例，理解其内容。线程的功能封装在 System.Threading 命名空间中。因此，必须先将这个命名空间导入项目。一旦导入了这个命名空间，就可以建立一个能在主线程上和新工作线程上执行的方法。本示例中使用了方法 SimpleMethod()：

```
public void SimpleMethod()
{
   int i = 5;
   int x = 10;
   int result = i * x;
   Console.WriteLine("This code calculated the value " +
                     result.ToString() + " from thread ID: " +
                     AppDomain.GetCurrentThreadId().ToString()) ;
}
```

这里使用了第 1 章介绍的 AppDomain 类，来查找正在运行的线程。这个方法只要执行，就会进行汇总，并输出结果，报告在哪些线程上执行了计算。

程序的入口点是 Main()方法。在这个方法中，首先执行 SimpleMethod()方法，这样，该方法就会在运行 Mian()方法的线程上运行，接下来的部分是很重要的。首先看看如何创建线程，在 C#中创建线程之前，必须先创建一个 ThreadStart 委托实例。委托就是面向对象的、类型安全的函数指针。下面要告诉线程执行什么函数，所以给线程的构造函数传递一个函数指针，在本例中，这个过程如下所示：

```
ThreadStart ts = new ThreadStart(simpleThread.SimpleMethod);
```

注意，方法名并没有放在圆括号中；它只接受方法名称。创建了 ThreadStart 委托后，就可以创建要执行的线程了。线程的惟一构造函数以 ThreadStart 委托的实例作为其参数，其代码如下：

```
Thread t = new Thread(ts);
```

把变量 t 声明为新的线程。Thread 类的构造函数把 ThreadStart 委托作为其惟一的参数。

下一行代码调用 Thread 对象的 Start()方法，开始一个新的执行线程，也就是说，首先调用传递给构造函数的 ThreadStart 委托，该委托再调用 Start()方法。之后使用 Console.ReadLine()方法，这样，在退出主线程之前程序会一直等待用户的键盘输入：

```
t.Start();
Console.ReadLine();
```

在第二次执行该方法时，代码实际上在另一个线程上执行。

现在创建了一个线程，但它不具有任何功能。我们显示不同的线程 ID 也说明不了问题——毕竟还没有一次执行多个线程。为了理解如何在更真实的应用程序中使用这些线程代码，下面创建另一个程序，模拟一个在后台执行的长进程，而另一个进程在前台执行。创建一个新的控制台应用程序，将下面的代码放在新文件 do_something_thread.cs 中：

```
using System;
using System.Threading;

public class DoSomethingThread
{
   static void WorkerMethod()
   {
      for(int i = 1; i < 1000; i++)
      {
         Console.WriteLine("Worker Thread: " + i.ToString());
      }
   }

   static void Main()
   {
      ThreadStart ts = new ThreadStart(WorkerMethod);
      Thread t = new Thread(ts);
      t.Start();
      for(int i = 1; i < 1000; i++)
      {
         Console.WriteLine("Primary Thread: " + i.ToString()) ;
      }

      Console.ReadLine() ;
   }
}
```

输出结果每次都会有些不同。每次执行线程时，都会在循环的不同点上切换。若把结果串联起来，就会如下所示：

```
Primary Thread: 1
Primary Thread: 2
Primary Thread: 3
...
```

```
Worker Thread: 743
Worker Thread: 744
Worker Thread: 745
...
Primary Thread: 1000
```

因为这些代码没有涉及新的编码技术，所以这里不再逐一介绍。不过可以看出，执行时间在两个线程之间共享。在一个线程完成之前，另一个线程并没有完全停止。每个线程都会有一小段时间来执行。在一个线程用完它的执行时间后，下一个线程就开始在它的时间片中执行。这两个线程会一直这样交替执行下去，直到执行完毕。实际上，系统上有多于两个的线程在交替执行，共享时间片。在前面的应用程序中，就不只是在两个线程之间来回切换。事实上，该应用程序的线程在与当前运行在计算机上的许多线程一起共享执行时间。

## 2.1.3　ThreadStart 委托和执行分支

现在回过头来看一下 ThreadStart 委托。使用这些委托可以完成一些有趣的工作。下面介绍一个现实生活中的小例子。假定在用户启动应用程序时需要执行某个后台例程。根据启动应用程序的用户的不同，将执行不同的例程。例如，在管理员登录到应用程序上时，就要运行一个后台进程，来收集报告数据，并将其格式化。当报告完成后，这个后台进程就会通知管理员。而对于普通用户，就不需要实现这个报告功能了。这就是 ThreadStart 的面向对象特性。

下面看看示例代码。这里不对上面描述的情况编写代码，但会说明如何基于 ThreadStart 类中定义的某一标准进行分支。创建一个新的控制台应用程序，在 ThreadStartBranching.cs 文件中添加以下代码：

```
using System;
using System.Threading;

public class ThreadStartBranching
{
   enum UserClass
   {
      ClassAdmin,
      ClassUser
   }

   static void AdminMethod()
```

```
}
   Console.WriteLine("Admin Method") ;
}

static void UserMethod()
{
   Console.writeLine("User Method");
}

static void ExecuteFor(UserClass uc)
{
   ThreadStart ts;
   ThreadStart tsAdmin = new ThreadStart(AdminMethod);
   ThreadStart tsUser = new ThreadStart(UserMethod);

   if(uc = UserClass.ClassAdmin)
      ts = tsAdmin;
   else
      ts= tsUser;

   Thread t = new Thread(ts);
   t.Start() ;
}

static void Main()
{
   // execute in the context of an admin user
   ExecuteFor(UserClass.ClassAdmin);
      // execute in the context of a regular user
      ExecuteFor(UserClass.ClassUser) ;

      Console.ReadLine() ;
   }
}
```

以上代码的输出是非常简单的:

```
Admin Method
User Method
```

下面详细讲解其中的一些重点。首先，这段代码创建了一个可以执行代码的用户类型的枚举：

```
enum UserClass
{
   ClassAdmin,
   ClassUser
}
```

接着创建两个方法：AdminMethod()和 UserMethod()。在理论上，这两个方法会为两种不同的用户类型执行完全不同的长指令序列。在本示例中，只标识这两个方法已运行过，并将它们写到控制台上：

```
static void AdminMethod()
{
   Console.WriteLine("Admin Method") ;
}

static void UserMethod()
{
   Console.WriteLine("User Method") ;
}
```

之后，在 ExecuteFor()方法中，把一个变量 ts 声明为 ThreadStart 类，但并没有用 New 关键字创建实例。然后创建两个新的 ThreadStart 对象，指向上面创建的不同方法：

```
ThreadStart ts;
ThreadStart tsAdmin= new ThreadStart(AdminMethod);
ThreadStart tsUser = new ThreadStart(UserMethod);
```

现在有了两个新的 ThreadStart 对象和一个可以保存 ThreadStart 实例的变量。然后，用 If 语句分支代码，并将空的 ts 变量设置为符合商务规则的 ThreadStart 实例：

```
if (uc == UserClass.ClassAdmin)
   ts = tsAdmin;
else
   ts = tsUser;
```

最后，给线程构造函数传送动态分配内存的 ThreadStart 委托，创建一个线程，并开始执行该线程：

```
Thread t = new Thread(ts) ;
t.Start() ;
```

## 2.1.4　线程的属性和方法

如本章开头所述，Thread 类有很多属性和方法。使用 System.Threading 命名空间可以使控制线程的执行简单得多。到目前为止，我们只是创建线程并启动它。

下面再介绍两个 Thread 类的成员：Sleep()方法和 IsAlive 属性。第 1 章说过，线程可以睡眠一段时间，直到发生时钟中断为止。要使线程进入睡眠，只需调用静态方法 Sleep()即可。线程的状态也是可以确定的。在下面的例子中，将使用 IsAlive 属性来判断线程是否执行完毕，并使用 Sleep()方法以暂停线程的执行。观察以下代码，这些代码在 thread_sleep.vb 文件中，该文件使用了这两个成员：

```
using System;
using System.Threading;

public class ThreadState
{
   static void WorkerFunction()
   {
      string ThreadState;

      for(int i = 1; i < 50000; i++)
      {
         if(i % 5000 ==0)
         {
            ThreadState = Thread.CurrentThread.ThreadState.ToString();
            Console.WriteLine("Worker; " + ThreadState) ;
         }
      }
      Console.WriteLine("Worker Function Complete");
}

static void Main()
{
   string ThreadState;
   Thread t = new Thread(new ThreadStart(WorkerFunction));
   t.Start ();
   while(t.IsAlive)
   {
      Console.WriteLine("Still waiting. I'm going back to sleep.");
```

```
            Thread.Sleep(200) ;
        }

        ThreadState = t.ThreadState.ToString();
        Console.WriteLine("He's finally done! Thread state is: "
                          + ThreadState) ;
        Console.ReadLine() ;
    }
}
```

输出结果如下所示(在 For 循环中试验不同的值，并传递到 Sleep()方法，就可以看到不同的结果)：

```
Still waiting. I'm going back to sleep.
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker: Running
Worker Function Complete
He's finally done! Thread state is: Stopped
```

Main()方法使用了前面介绍的新概念。首先，创建一个线程，然后把它传递给要执行为委托的方法：

```
Thread t = new Thread(new ThreadStart(Worker Function)) ;
t. Start ();
```

注意，这里并没有通过创建变量来保存 ThreadStart 类，而只是随时创建了一个变量，并把它作为参数传递给线程的构造函数。照例，Main()方法继续与新线程一起执行，因为处理器在这两者之间切换。然后，使用新建线程的 IsAlive 属性，确定线程是否还在执行。继续测试这个变量，当工作线程处于激活状态时，主线程将继续睡眠 200 毫秒，然后唤醒主线程，并测试工作线程是否仍处于激活状态：

```
while(t.IsAlive)
{
```

```
        Console.WriteLine("Still waiting. I'm going back to sleep.");
        Thread.CurrentThread.Sleep(200) ;
    }
```

接下来，看看代码中已使用两次的 ThreadState 属性。ThreadState 属性实际上返回一个枚举类型。这个枚举会说明线程处于什么状态。使用 If 语句可以测试这个属性，就像上一个例子那样，也可以在 ThreadState 属性上使用 ToString()方法，并以文本格式写出其状态：

```
ThreadState = t.ThreadState.ToString();
Console.WriteLine("He's finally done! Thread state is: "
                  + ThreadState) ;
```

代码的其余部分是标准的，不需要回顾。但有一些重要的地方需要注意。首先，让一个线程睡眠指定的一段时间，给其他线程让出执行时间。为此，使用 Thread 对象的 Sleep()方法——传入线程睡眠的时间(毫秒)。另外，可以使用 IsAlive 属性测试线程，看看这些线程是否已执行完毕。最后，使用线程实例的 ThreadState 属性确定线程的状态。

## 2.1.5　线程的优先级

线程的优先级决定了各个线程之间相对的优先级。ThreadPriority 枚举定义了可用于设置线程优先级的值，可用的值是：

- Highest(最高值)
- AboveNormal(高于正常值)
- Normal(正常值)
- BelowNormal(低于正常值)
- Lowest(最低值)

当运行库创建一个线程，但还没有分配优先级时，线程的初始优先级为 Normal。不过，可以使用 ThreadPriority 枚举改变优先级。在介绍线程优先级的例子之前，先讨论一下线程的优先级。创建一个简单的线程实例，显示当前线程 thread_priority.cs 的名称、状态和优先级信息：

```
using System;
using System.Threading;

public class ThreadPriority
{
  public static Thread worker;
```

```
static void Main()
{
   Console.WriteLine("Entering void Main()");
   worker = new Thread(new ThreadSCart(FindPriority));
   // Let's give a name to the thread
   worker.Name = "FindPriority() Thread";
   worker.Start() ;
   Console. WriteLine ("Exiting void Main()");
}

public static void FindPriority()
{
   Console.WriteLine("Name; " + worker.Name) ;
   Console.WriteLine("State: " + worker.ThreadState.ToString())
   Console.WriceLine("Priority; " + worker.priority.ToString())
}
}
```

这段代码有一个简单的方法 FindPriority()，该方法显示了当前线程的名称、状态和优先级信息，产生的输出如下所示：

```
Entering the void Main()
Exiting the void Main()
Name: FindPriority() Thread
State: Running
Priority: Normal
```

工作线程是以 Normal 优先级运行的。现在，添加一个新的线程，并用一个不同的优先级调用方法。下面是 thread_priority2.cs 文件的代码：

```
using System;
using System.Threading;

public class ThreadPriority2
{
   public static Thread worker;
   public static Thread worker2;
   static void Main()
   {
      Console.WriteLine ("Entering void Main()");
      worker = new Thread(new ThreadStart(FindPriority));
```

```
worker2= new Thread(new ThreadStart(FindPriority));

// Let's give a name to the thread
worker.Name = "FindPriority() Thread" ;
worker2.Name = "FindPriority() Thread 2";

// Give the new thread object the highest priority
worker2.Priority = System.Threading.ThreadPriority.Highest;

worker.Start() ;
worker2.Start() ;
    Console.WriteLine("Exiting void Main()");
    Console.ReadLine() ;
  }

  static public void FindPriority()
  {
    Console.WriteLine("Name; " + worker.Name);
    Console.WriteLine("State: " + worker.ThreadState.ToString());
    Console.WriteLine("Priority: " + worker.Priority.ToString());
  }

}
```

thread_priority2.cs 的输出结果如下所示：

```
Entering void Main()
Name: FindPriority() Thread2
State: Running
Priority; Highest
Exiting void Main()
Name: FindPriority() Thread
State: Running
Priority: Normal
```

线程是基于由 Priority 属性设置的优先级来安排执行的，每个操作系统都会根据不同的优先级来执行线程，操作系统还可以改变线程的优先级。

**注意：**

应用程序无法限制操作系统修改由开发人员为线程指定的优先级，因为操作系统

控制着所有的线程，它知道何时以及如何安排这些线程。例如，操作系统可以由于一些原因动态地修改线程的优先级，比如具有更高优先级的用户输入这样的系统事件，或者因缺乏内存而触发垃圾收集进程。

## 2.1.6　计时器和回调

前面介绍了一些简单的线程示例，但还没有触及同步问题，下一章将详细地介绍它。由于线程不像应用程序的其余代码那样按次序运行，所以不能确定发生在线程中影响特定共享资源的动作，是否会在另一个线程中的代码访问同一共享资源之前完成。处理这些问题有各种方法，这里介绍一种简单的方法：使用计时器。使用计时器，可以按特定的时间间隔执行方法，检查在继续操作之前所要求的动作是否已完成。这是一个非常简单的模型，但可以应用于很多种情况。

计时器由两个对象组成：TimerCallback 和 Timer。TimerCallback 委托定义了以指定间隔执行的方法，而 Timer 对象本身就是计时器。TimerCallback 将一个特定的方法与计时器联系起来。Timer 的构造函数(由重载得到)需要 4 个参数。第一个是前面指定的 TimerCallback 对象，第二个是可用于将状态传输给指定方法的一个对象。后两个参数分别是开始调用方法之后的时间，以及以后调用 TimerCallback 方法的时间间隔。这些参数可以是整型或者长整型，表示毫秒数。而在下面的内容中，使用的是 System.TimeSpan 对象，该对象可以以时钟滴答、毫秒、秒、分、小时或天为单位指定间隔。

演示 System.TimeSpan 对象的最简单方法是通过示例证明下面的应用程序执行了两个线程。第一个线程完成其操作后，第二个线程才会执行：thread_timer.cs 文件如下所示：

```
using System;
using System.Threading;
using System.Text;

public class TimerExample
{
  private string message;
  private static Timer tmr;
  private static boot complete;
```

上面的内容很简单。将 tmr 声明为静态变量，并适用于整个类，因为它在 Main() 方法中定义：

```
public static void Main()
{
   TimerExample obj = new TimerExample();
   Thread t = new Thread(new ThreadStart(obj.GenerateText));
   t.Start();

   TimerCallback tmrCallBack = new TimerCallback(obj.GetText);
   tmr = new Timer(tmrCallBack, null, TimeSpan.Zero,
                   TimeSpan.FromSeconds(2)) ;
```

这里触发了一个将在 GenerateText()方法上执行的新线程，该方法在一个 For 循环中迭代，以产生一个字符串，并存储到类范围内的 message 字段中：

```
   Do
   {
      if( complete )
       break;
   } while(true) ;
   Console.WriteLine("Exiting Main...");
   Console.ReadLine();
}
```

以上的循环冻结了 Main()循环，直到 complete 字段为 True 为止。在 GUI 中，可以使用不同的方法，因为 Application.Run()方法使应用程序始终处于无限循环中：

```
public void GenerateText()
{

    StringBuilder sb = new StringBuilder();
    For(int i = 1; i < 200; i++)
    {
        sb.Append(sb.Length, "This is Line ") ;
        sb.Append (sb. Length, i . ToString () ) ;
        sb.Append(sb.Length, System.Environment.NewLine) ;
    }

  message = sb.ToString() ;
}
```

以上使用了第一个方法，该方法使用一个 StringBuilder 对象产生了 200 行文本，然

后将这些文本行存储到 message 字段中。

```
    public void GetText(object state)
    {
      if(message == null)
        return;
      Console.WriteLine("Message is :") ;
      Console.WriteLine(message) ;
      tmr.Dispose() ;
      complete = true;
    }
} // class
```

这个类中使用的最后一个方法由计时器每两秒触发一次。如果还没有设置 message 字段，这个方法就会退出；否则就输出一个消息，然后删除计时器。这将使计时器停止继续计时。只要不再需要计时器，就应该执行这个操作。

thread_timer.cs 的输出结果如下：

```
Message is :
This is Line 1
This is Line 2

...

This is Line 199
This is Line 200

Exiting Main...
```

## 2.1.7　使用线程调节线程

前面介绍了在代码中如何从 Sub Main()方法产生线程。用类似的方法，还可以在一个线程中产生多个线程。例如，假定有一个 Car 类，它有一个公共方法 StartTheEngine()。该方法调用另外 3 个私有方法，分别为 CheckTheBattery()、CheckForFuel()和 CheckTheEngine()方法。由于检查电池、燃料和引擎可能同时发生，可以在不同的线程中运行每个方法。下面是 Car 类在 thread_spinning.vb 中的实现代码：

```
using System;
using System.Threading;

class Car
{
   public void StartTheEngine()
   {
      Console.WriteLine("Starting the engine!");
      //Declare three new threads
      Thread batt = new Thread(new ThreadStart(CheckTheBattery));
      Thread fuel = new Thread(new ThreadStart(ChcckForFuel));
      Thread eng = new Thread(new ThreadStart(CheckTheEngine));

      batt.Start();
      fuel.Start();
      eng.Start ();

      for(int i = l; i < 100000000; i++)
      {
           // some real executing code here
      }
      Console.WriteLine("Engine is ready!");
}

private void CheckTheBattery()
{
   Console.WriteLine("Checking the Battery!");
   for(int i = 1; i < 100000000; i++)
   {
      // some real executing code here
   }
   Console.WriteLine("Finished checking the Battery!");
}

private void CheckForFuel()
{
   Console.WriteLine("Checking for Fuel ! ") ;
```

```
for(int i = 1; i < 100000000; i++)
{
   // some real executing code here
}

   Console.WriteLine("Fuel is available!");
}

private void CheckTheEngine()
{
   Console.WriteLine("Checking the engine!");
      for(int i = 1; i < 100000000; i++)
      {
          // some real executing code here
      }
      Console.WriteLine("Finished checking the engine!");
   }
}
```

在 StartTheEngine()方法中，创建 3 个线程，然后依次启动每一个线程。在类中添加一个入口点，看看代码的运行结果：

```
public static void Main()
{

    Console.WriteLine("Entering void Main!");
    int j ;
    Car myCar = new Car() ;
    Thread worker = new Thread(new ThreadStart(myCar.StartTheEngine));
    worker.Start();
    for(int i = 1; i < 100000000; i++)
    {
        //
    }
    Console.WriteLine("Exiting void Main!");
    Console.ReadLine() ;
}
```

在 Void Main()方法中，又创建了一个线程，在这个线程中执行 StartTheEngine()方法，如图 2-1 所示。

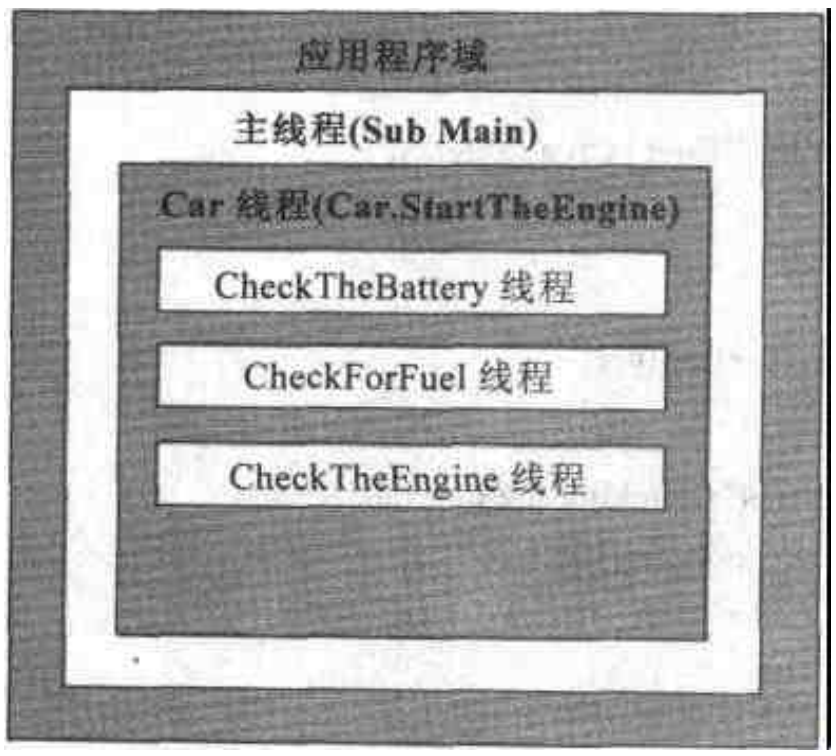


图　2-1

输出结果如下：

```
Entering Void Main!
Exiting Void Main!
Starting the engine!
Checking the Battery!
Checking for Fuel!
Checking the engine!
Finished checking the Battery!
Fuel is available!
Finished checking the engine!
Engine is ready!
```

这些方法都在自己的线程中执行，在自己的时间片中执行。

### 1. 用线程调节线程

可以将 Car 类拆分为几个单独的类，在一个新的 Engine 类中再构建两个方法，分别为 check1()和 check2()方法。然后，这个 Engine 类会在自己的线程中执行 check1()和 check2()方法，如图 2-2 所示。


图　2-2

从 Car 类中删除 CheckTheEngine() 方法，再创建一个 Engine 类；现在 thread_spinning2.cs 文件如下：

```
using System;
using System.Threading;

class Engine
{
   public void CheckTheEngine()
   {
      Thread chck1 = new Thread(new ThreadStart(Check1));
      Thread chck2 = new Thread(new ThreadStart(Check2));
      chck1.Start() ;
      chck2.Start() ;
      Console.WriteLine("Checking the engine!");
      for(int i = 1; i < 100000000; i++)
      {
         // some real executing code here
      }
      Console.WriteLine("Finished checking the engine!");
}

private void Check1()
```

```
{
    Console.WriteLine("Starting the engine check1!");
    for(int i = 1; i < 100000000; i++)
    {
        // some real executing code here
    }
    Console.WriteLine("Finished engine check1!");
}

private void Check2()
{
    Console.WriteLine("Starting the engine check2!");
    for(int i = 1; i < 100000000; i++)
    {
        // some real executing code here
    }
    Console.WriteLine("Finished engine check2!");
  }
}
```

Engine类含有CheckTheEngine()公共方法，该方法又创建了两个线程，并调用check1()和check2()方法。其结果如下所示：

```
Entering Void Main!
Exiting Void Main!
Starting the engine!
Checking the Battery!
Checking for Fuel!
Checking the engine!
Starting the engine check1!
Starting the engine check2!
Finished checking the Battery!
Fuel is available!
Engine is ready!
Finished engine check1!
Finished checking the engine!
Finished engine check2!
```

在线程中创建线程是非常容易的。但这项功能的缺点是：活动线程的数量增加了，性能降低了。

### 2. 性能

创建的线程越多，系统需要承担的维护线程环境和 CPU 指令的工作就越多。Windows Task Manager 的 Processes 选项卡将说明，当前有多少个进程和线程在运行。但是这些都是操作系统进程，它们并不等于 AppDomain。在使用线程窗口调试给定的.NET 应用程序时，还可以看到正在运行着的线程。

如果要知道有多少个线程在公共语言运行库(CLR)中运行，就必须使用 Windows Performance Monitor 工具，然后添加几个公共语言运行库专用的性能类别。公共语言运行库提供了一种性能计数器类别.NET CLR LocksAndThreads，使用这个类别可以获得 CLR 托管线程的更多信息。现在，运行 Performance Monitor，然后从.NET CLR LocksAndThreads 类别中添加表 2-4 中的计数器。

表 2-4

| 性能计数器 | 说　明 |
| --- | --- |
| # of current logical Threads | 该计数器显示应用程序中当前托管线程的数目，包括正在运行的和已经停止运行的线程 |
| # of current physical Threads | 该计数器显示由公共语言运行库创建并拥有的操作系统线程的数目。这个计数器可能和托管线程不能一一映射 |
| # of total recognized threads | 该计数器显示由公共语言运行库识别的当前线程的数目 |
| Current Queue Length | 该计数器显示托管应用程序中正在等待获得锁的线程的数目 |
| Total # of Contentions | 该计数器显示托管应用程序试图获得锁而失败的次数 |

图 2-3 显示了利用各种值查找 thread_spinning2 应用程序的过程。

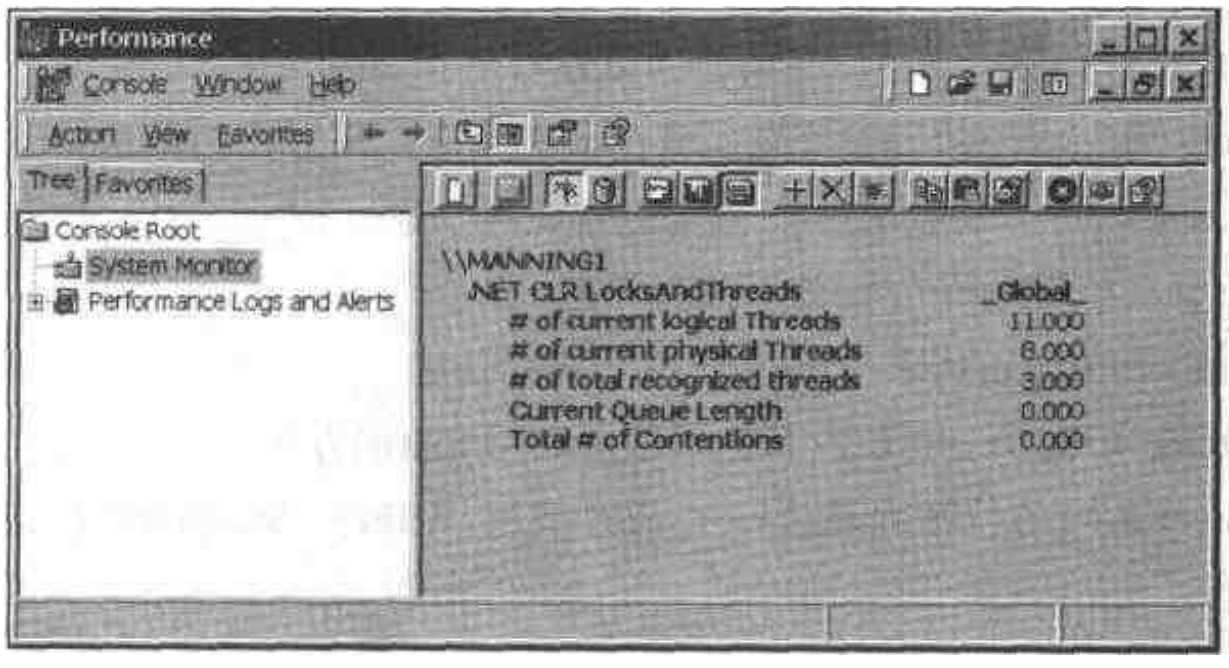


图　2-3

下面总结.NET CLR LocksAndThreads 性能计数器的信息：

- # of current local Threads 计数器指定公共语言运行库创建并拥有 11 个托管线程。

**提示：**

由于添加了_Global_计数器实例，所以所有的线程都由公共语言运行库创建。

- # of current physical Threads 计数器指定公共语言运行库创建并拥有 8 个 OS 线程。
- # of total recognized Threads 计数器指定公共语言运行库识别 3 个 OS 线程，这 3 个线程是由 Thread 对象创建的。
- Total # of Contentions 计数器指定运行库试图获得托管锁时并没有失败。托管锁的失败对代码的执行是不利的。

## 2.2　线程的生存期

当安排执行一个线程后，这个线程会经历几个状态，包括未启动、激活、睡眠状态，等。Thread 类包含一些方法，允许启动、停止、恢复、中止、挂起以及联接(等待)线程。使用线程的 ThreadState 属性可以确定线程当前的状态，这个状态将是 ThreadState 枚举中指定的一个值：

- Aborted(中止)——线程处于停止状态，但不一定移执行完毕
- AbortRequested(请求中止)—— Abort()方法已调用，但是线程还没有收到试图终止自己的 System.Threading.ThreadAbortexception，也就是说，线程还没有停止，但不久就会停止
- Background(后台)——线程在后台执行
- Running(运行)——线程已启动，而且没有停止
- Stopped(停止)——线程已完成了其所有的指令，而且已经停止
- StopRequested(请求停止)——请求停止线程
- Suspended(挂起)——线程处于挂起状态
- SuspendRequested(请求挂起)——请求线程挂起
- Unstarted(未启动)——还没有在线程上调用 Start()方法
- WaitSleepJoin(等待睡眠连接)——通过调用 Wait()、Sleep()或 Join()方法，来暂停线程

图 2-4 显示了线程的生存期。

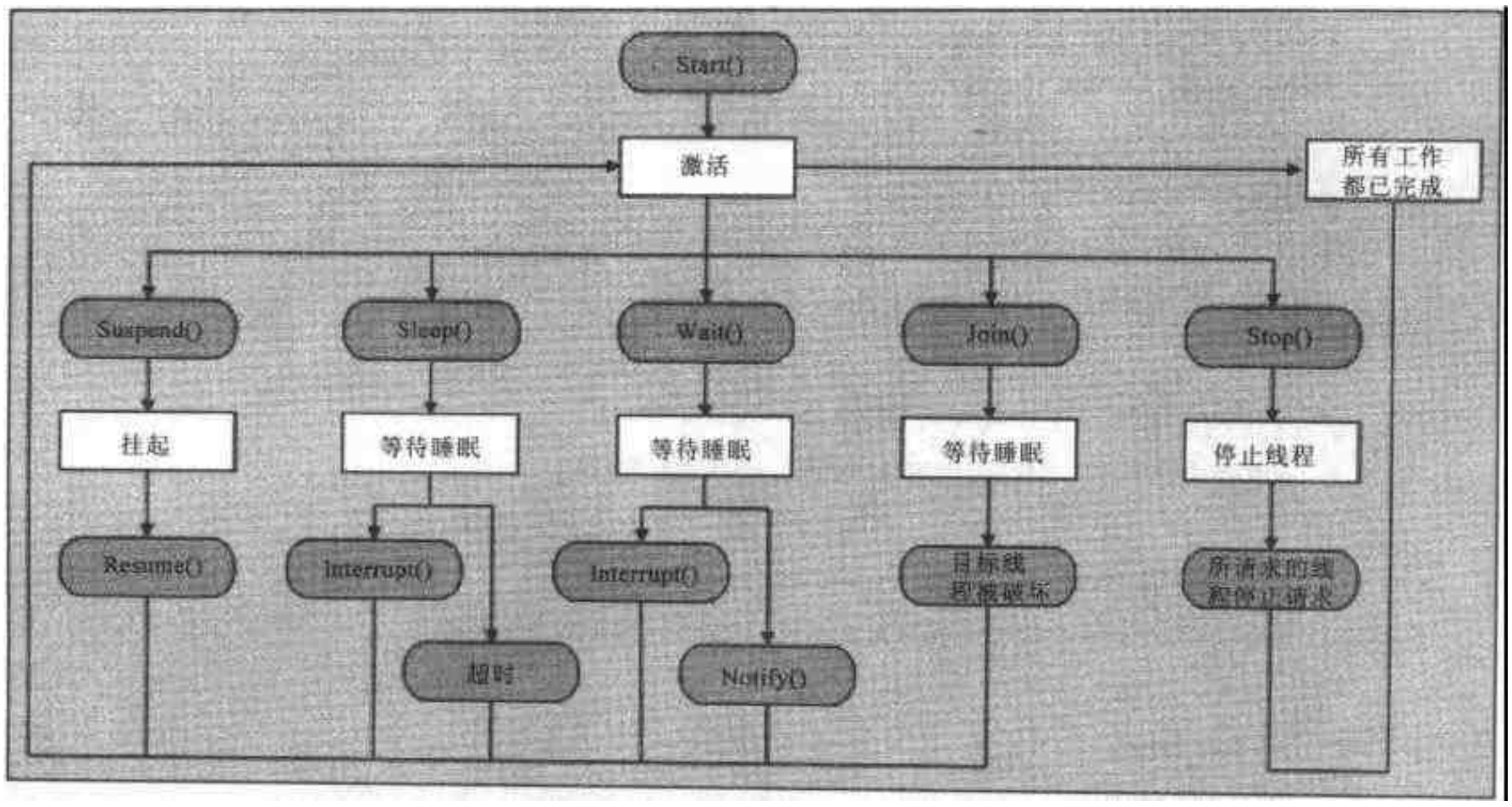


图　2-4

本节将讨论线程的生存期。

## 2.2.1　使线程睡眠

当创建新线程时，必须调用 Thread 对象的 Start()方法以安排线程的进度。此时，公共语言运行库会将一个时间片分配给传递到 Thread 对象的构造函数的方法地址。一旦线程处于运行(Running)状态，当操作系统处理其他线程时，它就可以进入睡眠(Sleep )或中止(Abort)状态。用 Thread 类的 Sleep()方法可以使线程进入睡眠。如果在等待一项资源，且要重复使用这项资源，就可以使用 Sleep()方法。例如，假定由于要访问的资源不可使用，应用程序就不能继续进行。应用程序希望在几毫秒之后，再次试着访问该资源，此时就可以使用 Sleep()方法，使线程在应用程序再次访问资源之前睡眠一段指定的时间。

重载的 Sleep()方法可以以两种方式使用。第一种重载将一个整数作为参数，指定线程挂起指定的毫秒数。例如，如果给参数输入 100，线程就会挂起 100 毫秒，使线程处于 WaitSleepJoin 状态。下面是一个这样的例子，thread_sleep2.cs：

```
using System;
using System.Threading;

public class ThreadSleep
{
```

```
public static Thread worker;
public static Thread worker2;

public static void Main()
{
   Console.WriteLine("Entering the void Main!");

   worker = new Thread(new ThreadStart(Counter));
   worker2 = new Thread(new ThreadStart(Counter2));

   // Make the worker2 object as highest priority
   worker2.Priority = System.Threading.ThreadPriority.Highest;

   worker.Start() ;
   worker2.Start();

   Console.WriteLine("Exiting the void Main!");
}

public static void Counter()
{
   Console.WriteLine("Entering Counter");
   for(int i= 1; i < 50; i+ +)
   {
      Console.write(i + " ");
      if(i ==10)
         Thread.Sleep(1000) ;
   }
   Console.WriteLine();
   Console.WriteLine("Exiting Counter") ;
}

public static void Counter2 ()
{
   Console.WriteLine("Entering Counter2"),
   for(int i = 51; i < 100; i++)
   {
         Console.Write(i + " ") ;
         if( i == 70 )
            Thread.Sleep(5000) ;
```

```
        }
        Console.WriteLine();
        Console.WriteLine("Exiting Counter2") ;
    }
}
```

Counter()方法从 1 计数到 50，当计数到 10 时，线程就会睡眠 1000 毫秒。Counter2()方法从 51 计数到 100，当计数到 70 时，线程会睡眠 5000 毫秒。输出结果如下所示：

```
Entering the Void Main!
Entering Counter2
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 Exiting the Void Main!
Entering Counter
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Exiting Counter
71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Exiting Counter2
```

第二种重载将 TimeSpan 对象作为参数，并且基于 TimeSpan 值挂起当前的线程。TimeSpan 是一种在 System 命名空间中定义的结构体。TimeSpan 结构体有几个有用的属性，这些属性基于时钟的滴答返回时间间隔。使用 Public 方法，如 FromSeconds()和 FromMinutes()可以指定睡眠的持续时间，下面是一个例子 thread_sleep3.cs 的代码如下：

```
public static void Counter()
{
   ...
   for(i = 1; i < 50; i++)
   {
      Console.Write(i + " ") ;
      if(i == 10)
         Thread.Sleep(System.TimeSpan.FromSeconds(1))
   }
   ...
}

public static void Counter2()
{
      ...
   for(int i = 51; i < 100; i++)
```

```
        {
            Console.Write(i + " ");
            if( i == 70 )
                Thread.Sleep(5000) ;
        }
        ...
    }
```

它的输出结果与 thread_sleep2 的输出类似。

## 2.2.2　中断线程

当线程进入睡眠时，它就进入 WaitSleepJoin 状态。如果线程处于睡眠状态，在到达指定的睡眠时间之前唤醒线程的惟一方法是使用 Interrupt()方法。Interrupt()方法会将线程放回到调度队列中。下面是示例 thread_interrupt.cs 的代码：

```
using System;
using System.Threading;

public class Interrupt
{
    public static Thread sleeper;
    public static Thread worker;

    public static void Main()
    {
        Console.WriteLine("Entering the void Main!");

        sleeper = new Thread(new ThreadStart(SleepingThread));
        worker = new Thread(new ThreadStart(AwakeTheThread));

        sleeper.Start () ;
        worker.Start() ;

        Console.WriteLine("Exiting the void Main!");
    }
public static void SleepingThread()
{
    for(int i = 1; i < 50; i++)
```

```
        {
            Console.write(i + " ");
            if(i == 10 || i == 20 || i == 30)
            {
                Console.WriteLine("Going to sleep at: " + i) ;
                Thread.Sleep(20);
            }
        }
    }

    public static void AwakeTheThread()
    {
        for(int i = 51; i < 100; i++)
        {
            Console.Write(i + " ") ;
            if (sleeper.ThreadState ==
                System.Threading .ThreadState.WaitSleepJoin)
            {
                Console.WriteLine("Interrupting the sleeping thread"),
                sleeper.Interrupt();
            }
        }
    }
}
```

在上面的例子中，当计数器达到 10、20 和 30 时，第一个线程(sleeper 线程)进入睡眠状态。第二个线程(worker 线程)检查第一个线程是否处于睡眠状态。如果是，就中断第一个线程，将其放回到调度队列中。Interrupt()方法是将处于睡眠状态的线程重新激活的最好方法，如果等待资源的过程结束，且希望线程进入激活状态，就可以使用这项功能。输出结果如下所示：

```
Entering the Void Main!
Exiting the Void Main!
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 1 2 3 4 5 6 7 8 9 10 Going to sleep
at: 10
11 12 13 14 15 16 17 18 19 20 Going to sleep at: 20
21 22 23 24 25 26 27 28 29 30 Going to sleep at: 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

## 2.2.3　暂停及恢复线程

Thread 类的 Suspend()和 Resume()方法可以用来挂起和恢复线程。Suspend()方法将无限期地挂起当前线程，直到另一个线程将其唤醒。调用 Suspend()方法时，线程会处于 SuspendRequested 或 Suspended 状态。

下面是一个暂停和恢复线程的示例。创建一个新的 C#应用程序，该程序在一个新的线程中产生质数，并可以中止和恢复这个产生质数的线程。要检验这些功能，可以创建一个新的 C# WinForms 项目 PrimeNumbers，然后在 Form1 中建立一个用户界面，如图 2-5 所示。

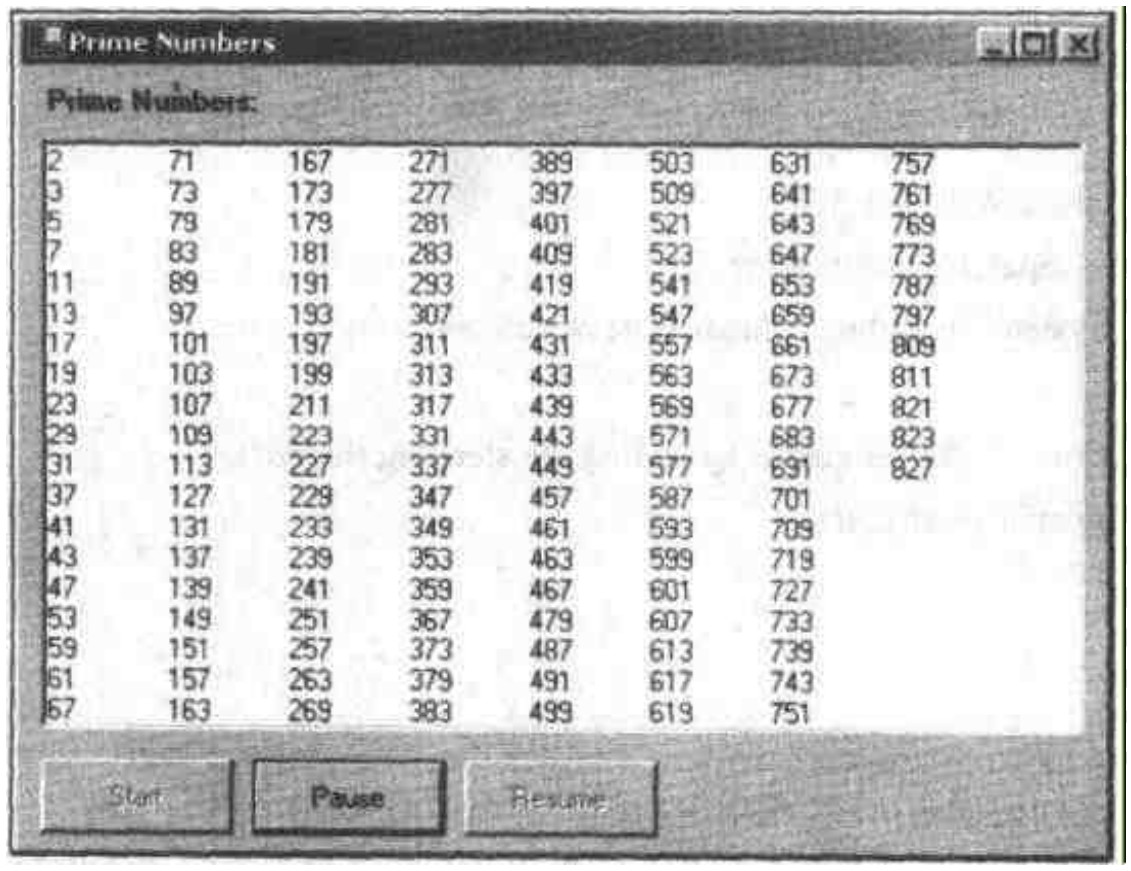


图　2-5

在用户界面中有一个列表框和 3 个命令按钮。列表框用来显示质数，3 个命令按钮用来启动、暂停及停止线程。最初，暂停和恢复按钮是禁用的，因为只有启动了线程才能启用这两项命令。下面来看一下代码。该代码首先声明了一个类级别的 Thread 对象，用于产生质数。

```
using System,
using System. Drawing;
using System. Collections;
using System. Component Model,
using System. Windows.Forms ;
using System .Threading;
```

```
namespace Chapter_02
{
  public class Form1 : System.Windows.Forms.Form
  {
    // private thread variable
    private Thread primeNumberThread;
```

双击 Start 命令按钮，添加以下代码。

```
private void cmdStart_Click(object sender, System.EventArgs e)
{
  // Let's create a new thread
  primeNumberThread = new Thread(
    new ThreadStart(GeneratePrimeNumbers));

  // Let's give a name for the thread
  primeNumberThread.Name = "Prime Numbers Example";

  primeNumberThread.Priority = ThreadPriority.BelowNormal;

  // Enable the Pause Button
  cmdPause.Enabled = true;
  // Disable the Start button
  cmdStart.Enabled = false;

  // Let's start the thread
  primeNumberThread.Start () ;
}
```

单击 Start 按钮，可以用 GeneratePrimeNumbers()方法的 ThreadStart 委托创建一个新的 Thread 对象，并给线程指定名称 Prime Number Example。然后启用 Pause 按钮，禁用 Start 按钮。接下来，使用 Thread 类的 Start 方法启动产生质数的线程。

双击 Pause 按钮，添加如下代码：

```
private void cmdPause_Click(object sender, System.EventArgs e)
{
  try
  {
  try
  {
    // If current state of thread is Running,
```

```
// then pause the Thread
if (primeNumberThread.ThreadState ==
   System.Threading.ThreadState.Running)
{

   //Pause the Thread
   primeNumberThread.Suspend();

   //Disable the Pause button
   cmdPause.Enabled = false;

      //Enable the resume button
      cmdResume.Enabled = true;
   }
}
catch(ThreadStateException Ex)
{
   MessageBox.Show(Ex.ToString(), "Exception",
      MessageBoxButtons.OK, MessageBoxIcon.Error,
      MessageBoxDefaultButton.Button1) ;
   }
}
```

Pause 按钮检查线程是否处于 Running 状态。如果是，它就调用 Thread 对象的 Suspend 方法，暂停线程。然后，启用 Resume 按钮，禁用 Pause 按钮。Suspend 方法可以引发 ThreadStateException 异常，所以将代码包装在 try...catch 块中。

双击 Resume 按钮，添加如下代码：

```
private void cmdResume_Click (object sender. System. EventArgs e)
{
   if(primeNumberThread.ThreadState==
      System.Threading.ThreadState.Suspended ||
      primeNumberThread.ThreadState ==
      System.Threading.ThreadState.SuspendRequested)
   {
      try
      {
         // Resume the thread
         primeNumberThread.Resume();
```

```
            // Disable the resume button
            cmdResume.Enabled = false;

            // Enable the Pause button
            cmdPause.Enabled = true;
        }
        catch(ThreadStateException Ex)
    {
        MessageBox.Show(Ex.ToString(), "Exception",
            MessageBoxButtons.OK, MessageBoxIcon.Error,
            MessageBoxDefaultButton.Button1);
        }
    }
}
```

Resume 按钮在恢复线程前检查线程状态是 Suspended 还是 SuspendRequested。如果线程的状态是 Suspended 或 SuspendRequested，就恢复线程，禁用 Resume 按钮，启用 Pause 按钮。

到目前为止，商务逻辑已经准备好，下面看看生成质数的代码。本例的主要目的是使用多线程而不是生成质数，所以不深入讲解这些代码。GeneratePrimeNumbers()方法生成从 3 开始的前 255 个质数。该方法找到一个质数后，就将这个新的质数分别添加到数组和列表框中。第一个质数 2 会自动添加到列表框中。最后，方法 GeneratePrimeNumbers()会启用 Start 按钮，禁用 Pause 按钮。

```
public void GeneratePrimeNumbers()
{
    long lngCounter;
    long lngNumber;
    long lngDivideByCounter;
    bool blnIsPrime;
    long[ ] PrimeArray = new long[256];

    // initialize variables
    lngNumber = 3;
    lngCounter = 2;

    // We know that the first prime is 2. Therefore,
    // let's add it to the list and start from 3
    PrimeArray[1] = 2;
```

```
lstPrime.Iterms.Add(2) ;

while(lngCounter < 256)
{
   blnIsPrime a true;

   // Try dividing this number by any already found prime
   // which is smaller then the root of this number.
   for(lngDivideByCounter = 1; PrimeArray[lngDivideByCounter]
         * PrimeArray[lngDivideByCounter] <= lngNumber;
      lngDivideByCounter++)
   {
      if(lngNumber % PrimeArray[lngDivideByCounter] == 0)
      {
      // This is not a prime number
      blnIsPrime = false;
      // Exit the loop
      break ;
   }
}

// If this is a prime number then display it
if (blnIsPrime)
{
   // Guess we found a new prime.
   PrimeArray[lngCounter] = lngNumber ;
   // Increase prime found count.
   lngCounter++ ;
      lstPrime.Items.Add(lngNumber) ;
      // Let's put the thread to sleep for 100 milliseconds.
      // This will simulate the time lag and we'll get time
      // to pause and resume the thread
      Thread.Sleep(100);
   }

   // Increment number by two
   lngNumber += 2;
}
   // Once the thread is finished execution enable the start
   // and disable the pause button
```

```
        cmdStart.Enabled = true;
        cmdPause.Enabled = false;
    }
```

现在就可以运行代码，查看应用程序的结果了，如图 2-6 所示。

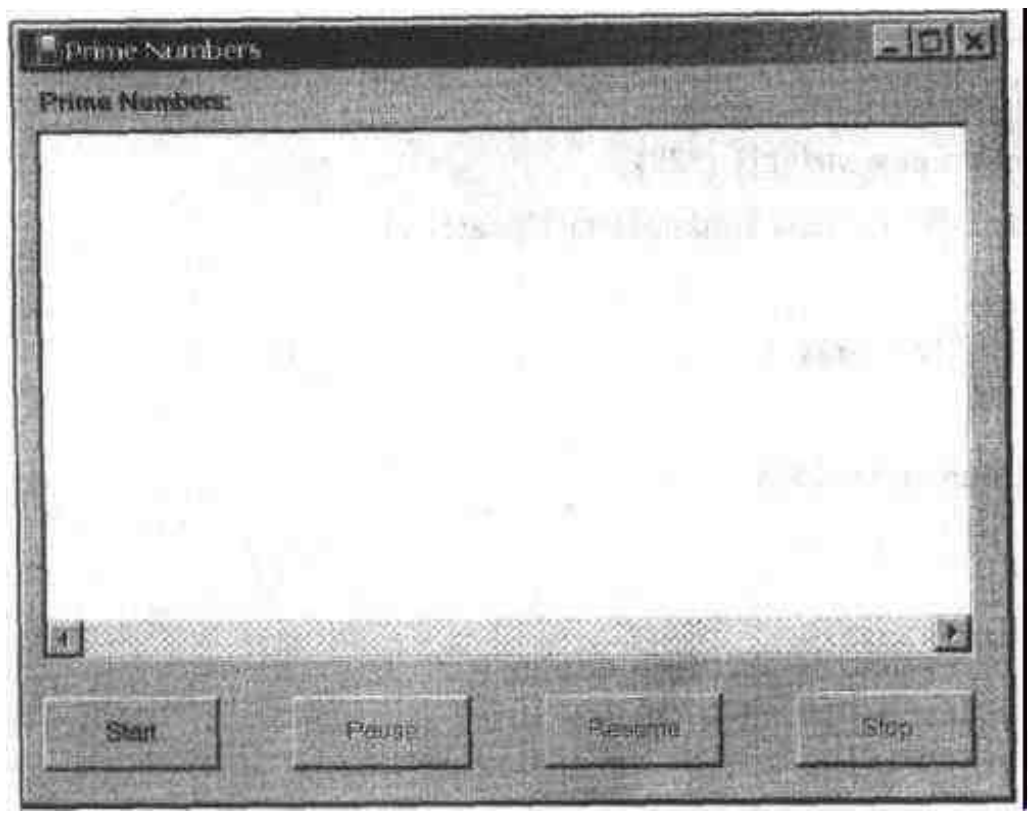


图　2-6

程序看起来一切正常，代码也运行良好。但是，代码中存在一个很大的缺陷。当 GeneratePrimeNumbers()方法查找到一个质数时，就将这个质数添加到列表框控件中。如果这些代码以同步方式运行，即质数生成代码和用户界面在同一个线程上运行，这就不成问题。但是在这个例子中，用户界面与 GeneratePrimeNumbers()方法在不同的线程中运行。在两个线程中运行以写入数据时，应用程序会发生一些意想不到的行为。

解决这个问题的最好方法是使用委托。可以声明一个委托，通知 UI 线程更新列表框控件。这样，就不会超出线程的边界，应用程序的稳定性也不会降低。

下面看看如何实现这一操作。添加一个公共委托 UpdateData：

```
public delegate void UpdateData (string returnVal) ;
```

稍微修改一下 GeneratePrimeNumbers()方法，以调用委托。添加一个新的字符串数组，将其初始值设置为 2，因为第一个质数是 2。然后，把一个 UpdateData 类型的新对象声明为委托，并传入 UpdateUI 方法的地址。接下来使用 Me.Invoke 方法、委托对象和字符串数组，通知用户界面更新其自身。当发现一个质数时，也进行同样的操作。

**提示：**

在这段代码中，Form1 用 this 代表。

```
public void GeneratePrimeNumbers()
{
   long lngCounter;
   long lngNumber;
   long lngDivideByCounter;
   bool blnIsPrime;
   long[ ] PrimeArray = new long [255];
   string[ ] args = new string[] {"2"};
   UpdateData UIDel = new UpdateData(UpdateUI);
   ...
   this.Invoke(UIDel, args) ;

while(lngCounter <= 255)
{
   ...
   // If this is a prime number then display it
   if(blnIsPrime)
   {
      // Guess we found a new prime.
      PrimeArray[lngCounter] = lngNumber;
      // Increase prime found count.
      lngCounter++;
      args[0] = lngNumber.ToString();
      this.Invoke(UIDel, args);
    ...
   }
   ...
}
```

UpdateUI()方法只接受参数中需要添加到列表框中的值，然后将值添加到列表框中。由于 UpdateUI()方法在 UI 线程中运行，不存在跨线程边界的更新，所以应用程序的稳定性也不会受损：

```
void UpdateUI(string result )
{
   lstPrime.Items.Add(result) ;
}
```

## 2.2.4　销毁线程

Abort()方法可以用来销毁当前的线程。如果因某种原因(比如线程执行了太长时间或用户选择了取消)要终止线程，就可以使用 Abort()方法。例如，如果搜索进程执行了很长时间，就可以终止该进程。搜索可以继续执行，但用户已经得到了需要的结果，就不再需要继续执行搜索例程上的线程了。在线程上调用 Abort()方法时，会引发 ThreadAbortException 异常。如果没有在线程的代码中捕获该异常，线程就会终止。在多线程环境下访问的方法中编写一般异常处理代码之前，应慎重考虑一下。因为 catch(Exception e)也会捕获 ThreadAbortException 异常(可能不希望从中恢复)。由此可见，ThreadAbortException 异常并不容易停止，程序流也不会像期望的那样继续执行。

下面看一个这样的例子。创建一个新项目 Destroying，然后将先前的质数生成代码复制到新的 Form1.cs 类中，再将 Stop 按钮添加到用户界面中，如图 2-7 所示。

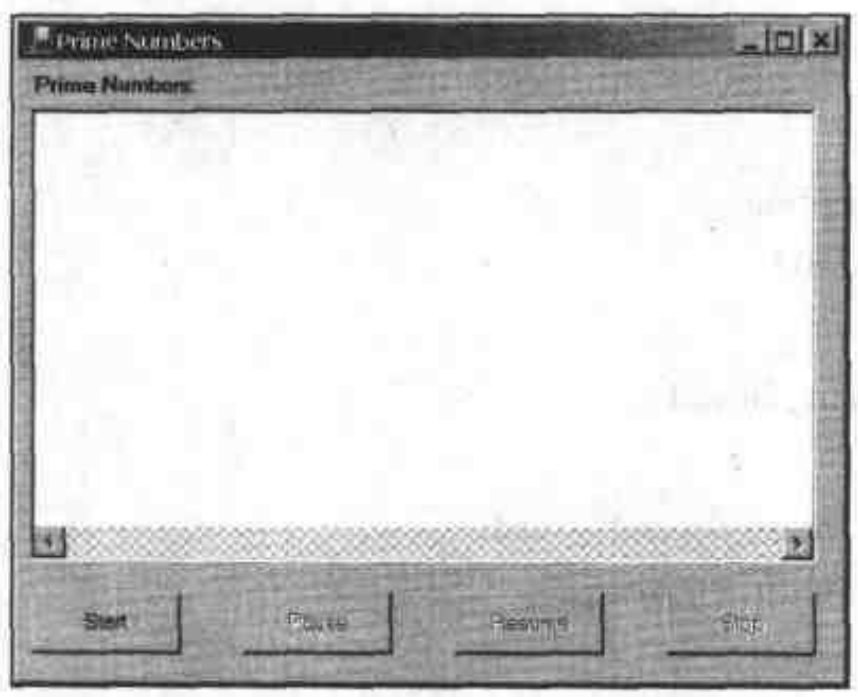


图　2-7

将如下代码添加到 Stop 按钮中。

```
private void cmdStop_Click(object sender. System.EventArgs e)
{
   // Enable the Start button and disable all others
   cmdStop.Enabled     = false;
   cmdPause.Enabled    = false;
   cmdResume.Enabled = false;
   cmdStart.Enabled    = true;
   // Destroy the thread
   primeNumberThread.Abort() ;

}
```

这个例子与前一个例子非常类似。惟一的不同是，当用户单击 Stop 按钮时，该程序将用 Abort()方法销毁线程。然后启用 Start 按钮，禁用其他按钮。还要注意，ThreadAbortException 是一个特殊的异常，与其他异常一样，这个异常也是可以捕获的。但是，一旦 catch 代码块执行完毕，就会再次自动引发该异常。此时，运行库就会执行 finally 代码块，之后销毁线程。

## 2.2.5 连接线程

Join()方法会暂停给定的线程，直到当前的线程终止。在给定的线程实例上调用 Join()方法时，线程将置于 WaitSleepJoin 状态。如果一个线程依赖于另一个线程，就可以使用这个方法。连接两个线程的意思是当调用 Join()方法时，运行着的线程将进入 WaitSleepJoin 状态，而直到调用 Join()方法的线程完成了任务，该线程才会返回到 Running 状态。这听起来有点混乱，下面用一个例子 thread_joining.cs 来说明，其代码如下所示：

```
using System;
using System. Threading;
namespace Chapter_02
{
   public class JoiningThread
   {
      public static Thread SecondThread;
      public static Thread FirstThread;
      static void First()
      {
         for (int i = 1; i <= 250; i++)
            Console.Write(i + " ");
   }

   static void Second()
   {
      FirstThread.Join();

      for(int i = 251; i <= 500; i++)
         Console.Write(i + " ");
   }

   public static void Main()
```

```
        {
            FirstThread = new Thread(new ThreadStart(First));
            SecondThread = new Thread(new ThreadStart(Second));

            FirstThread.Start() ;
            SecondThread.Start() ;
        }
    }
}
```

这个简单的示例的目的就是顺序地将数字输出到控制台，从 1 开始到 500 为止。First()方法将输出前 250 个数字，Second()方法则输出从 251 到 500 的数字。如果 Second()方法中没有 FirstThread.Join()行，执行流就会在两个方法之间来回切换回，输出结果就会很混乱(试着注释掉该行，再次运行这个例子)。通过在 Second()方法中调用 FirstThread.Join()方法，将暂停 Second()方法的执行，直到 FirstThread(First()方法)中的代码执行完毕。

Join()方法是重载的；它惟一的参数可以是一个整数，也可以是一个 TimeSpan，该方法将返回一个布尔值。调用这个方法的一个重载版本后，线程会暂停，直到另一个线程结束或者超时(以先发生的为准)为止。如果线程已经结束，返回值就是 True，否则将为 False。

## 2.3　为什么线程不是万能的

前面介绍了线程的几个非常有用的优点：一次可以运行多个进程，在这些进程中也可以运行多个线程。既然线程有这些优点，为什么不为新的方法使用新的线程呢？这样不就使每一项操作都快速运行吗？实际上并非这样。本节将说明，如果过度使用线程，就会适得其反。

多线程应用程序需要很多资源。线程需要内存来存储线程本地的存储器，所使用的线程数受到可用的内存总数限制。目前，内存是相当便宜的，因此很多计算机有很大的内存。但是，并不是所有的计算机都是这样。如果在未知的硬件配置上运行应用程序，就不能假定应用程序有足够的内存，也不能假定只有一个进程会产生线程，消耗系统资源。一台计算机有很多内存空间，并不意味着所有的内存都由一个应用程序使用。

每一个线程还会导致额外的处理器开销。如果在应用程序中创建太多的线程，就会限制线程的总执行时间。因此，与执行线程包含的指令相比，处理器在线程之间切换所花的时间会更多。如果应用程序创建了更多的线程，应用程序获得的执行时间将比其他包含较少线程的进程更多。

为了使这个概念更容易理解，下面用本地食品杂货店的例子来说明。两个收银员在为顾客扫描其购买的商品。但是，只有一个装袋员，他在为两个收银员装袋。装袋员在两个收银机之间的来回走动以及装袋的效率是较高的，因为杂货堆积的速度并不比装袋员装袋的速度快。不过，如果又有两个收银员开通收银通道，很明显装袋员就要花更多的时间在各个收银机之间来回走动，比他们花在装袋上的时间要多。最终，商店还需要一个装袋员。对于线程，可以将收银员看作应用程序，或者是线程，将装袋员看作处理器。处理器必须在各个线程之间切换。随着“线程”的增加，食品杂货店必须增加新的“处理器”，以保证顾客得到他们需要的服务。

“太多的线程”这个词语太泛泛了。一个系统上的“太多”在另一个系统上可能是正常的。由于硬件配置在很大程度上决定了系统上可用的线程数目，如果没有具体的配置信息和许多测试，“太多”只是一个无法量化的变数。

因此，Microsoft 建议在应用程序中使用尽可能少的线程，以限制操作系统需要的资源总数。

## 2.4　使用线程的时机

既然线程可能会对应用程序产生负面影响，为什么还要使用线程？重要的是要知道在什么时候，在什么地方使用线程。了解在什么环境下适于产生新线程是设计决策的关键因素。考虑建立新线程有两个完全不同的时机。本节将讨论这些时机。

### 2.4.1　后台进程

当应用程序需要在后台运行一个很大的进程，同时保持其户界面处于激活和可用状态时，就应产生新线程。我们都遇到过这样的情形：应用程序看起来没有反应，这是因为应用程序在查询数据或处理一个很大的信息。例如，专业的图形程序包需要将图形以一种特定的文件格式显示。在一些产品的早期版本中，让应用程序显示大图形，该程序会在显示的过程完成后才响应。用户常常需要结束该应用程序，再把它设置为用整个晚上来显示图形。等用户第二天早上上班时，再查看结果是否与期望的相符——因为坐在计算机前等待一个小时实在让人难以忍受。要解决这个问题，应设定一个理想时间，把后台线程设置为进行计算机密集的处理工作，而使用户界面在应用程序主线程上运行。

下面是一个需要产生新线程的后台进程例子。这个例子演示了文件的搜索。当搜索程序找到一个与指定模式相匹配的文件时，就会将一个新的数据项添加到列表框中，如图 2-8 所示。

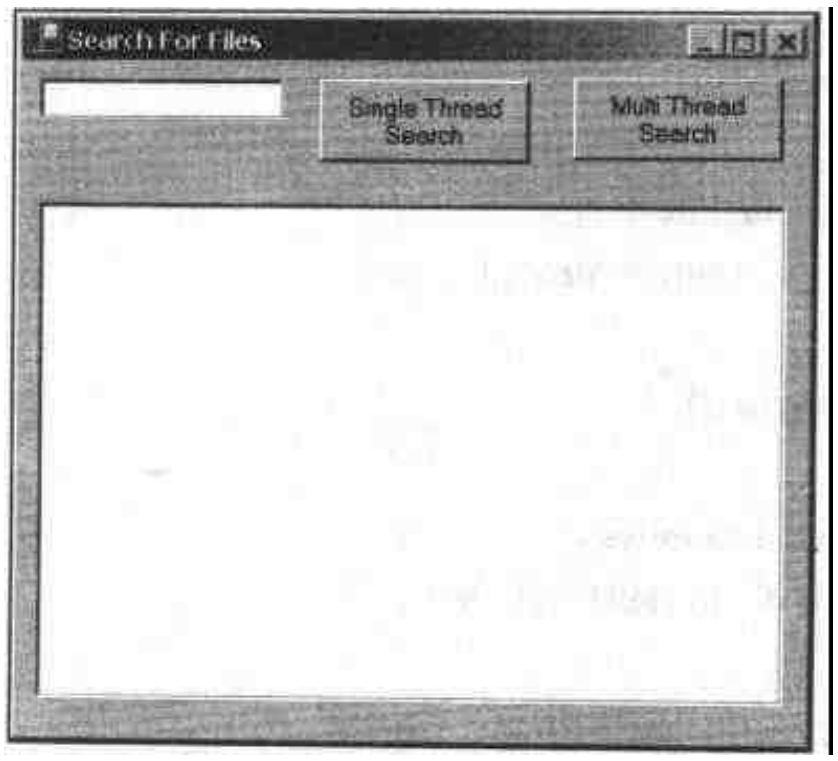


图　2-8

以下代码将说明这个方法确实需要自己的线程:

```
using System.Threading;
using System.IO;
public class Threaded_Search : System.Windows.Forms.Form
{
  string search;
  int fileCount;
  private void cmdSingle_Click(object sender, System.EventArgs e)
  {
    Search() ;
  }

  public void Search()
  {
    search = TextBox1.Text ;
    ListBox1.Items.Clear() ;
    fileCount = 0;
    SearchDirectory(@"C:\winnt") ;
  }

public void SearchDirectory(string Path)
{
  // Search the directory
  DirectoryInfo di = new DirectoryInfo(Path) ;
  FileInfo[ ] f = di.GetPiles(search);
```

```
ListBoxl.BeginUpdate() ;

foreach(FileInfo myFile in f)
   ListBoxl.Items.Add(myFile.FullMame) ;

ListBoxl.EndUpdate();

// Search its sub directories
DirectoryInfo[] d = di.GetDirectories();
      foreach(DirectoryInfo myDir in d)
        SearchDirectory(myDir.FullName) ;
    }
  }
```

编译这个例子并运行它。在搜索文本框中输入一个搜索词语，比如*.*，然后单击 Single Thread Search 按钮，观察所出现的问题。这个例子将搜索文件，每次使用搜索条件找到一个文件时，都要更新用户界面。但是，因为用户界面和搜索代码在同一个线程上运行，在搜索代码完成其处理过程之前，是看不到用户界面的更新的。另外，在进行搜索时，不能调整窗体的大小。

这部分相当长的代码实际上是一个非常简单的演示。下面看看是否可以简单地修改一下，更正这个问题。在 Button_Click 例程中，添加如下代码，使用新的线程调用 Search() 方法：

```
private void cmdMulti_Click(object sender, System.EventArgs e)
{
  Thread t = new Thread(new ThreadStart(Search));
  t.Start();
}
```

现在，重新编译并再次运行该程序。这次输入同样的搜索词，然后单击 Multi Thread Search 按钮。这次将发生很大的变化，其结果会立刻显示出来。这是因为 Windows 一直在用户界面和进行搜索线程之间来回切换着执行流。处理器现在有一个时间片来更新用户界面，以反映列表框中的变化。现在也可以在搜索的过程中调整窗体的大小了。

其他一些后台进程也会导致用户界面没有响应。人们有时需要进行一些密集的处理，比如搜索、排序、格式化、分析以及筛选数据库中的大量记录。这是适合使用新线程的另一种情形。如果要运行一个经常记录信息的例程，也可以创建新线程。在这种情况下，用户界面不一定会没有响应，但是如果这种例程不运行在自己的线程上，

用户界面的响应可能会比较慢。

## 2.4.2　访问外部资源

适合创建新线程的第二种情况是访问非本地系统上的资源。这可能是一个数据库进程或者是网络上的一个网络文件共享。在这些情况下，网络性能会对应用程序性能产生不利的影响。

下面用一个例子检验一下。这个例子要与一个数据库建立连接。假定网络性能很差，可能导致这个应用程序低速运行。再假定公司政策规定应用程序不能安装在数据库服务器上：

```
using System.Threading;
using System.IO;
using System.Data;
using System.Data.SqlClient;

namespace Chapter_02
{
   public class Threaded_Resource : System.Windows.Forms.Form
   {
      public void Button1_Click(object sender , System.EventArgs e)
      {
         QueryData() ;
      }

      public void QueryData()
      {
   SqlDataReader objReader;
   SqlConnection objConn;
   SqlCommand objCommand;
   int intEmployeeID;
   string strFirstName;
   string strTitle;
   int intReportsTo;
   objConn = new SqlConnection("server=RemoteServer;" +
         "UID=RemoteUser;PWD=Password;database=northwind") ;
   objCommand = new SqlCommand("SELECT EmployeeID, FirstName, " +
         "Title, ReportsTo FROM Employees", objConn);
```

```
obj Conn.Open() ;
objReader = objCommand.ExecuteReader(
      CommandBehavior.CloseConnection ) ;

while (objReader.Read())
{
   intEmployeeID = objReader.GetInt32(0);
   strFirstName = objReader.GetString(1);
   strTitle = objReader.GetString(2);
   if(objReader.IsDBNull(3))
      intReportsTo = 0;
   else
      intReportsTo = objReader.GetInt32(3);

   listBox1.Items.Add(intEmployeeID.ToString()+ " " +
      strFirstName + " " + strTitle + " " +
      intReportsTo.ToString()) ;
   }
   objReader.Close();
   objConn.Close () ;
}

public static void Main()
{
   Application.Run(new Threaded_Resource()) ;
}
}
```

这个例子只是查询一个远程数据库。返回的数据不会过多，但要注意，在程序获取数据及更新列表框的过程中，会冻结用户界面。下面创建一个新线程，并在这个线程中执行数据库代码，来修正这种情况。再添加一个按钮，并使用以下代码：

```
private void button2_Click(object sender, System.EventArgs e)
{
   Thread t = new Thread(new ThreadStart(QueryData)) ;
   t.Start() ;
}
```

现在，运行代码，此时将获得与上一个例子类似的结果。可以在运行查询的过程中，

调整窗体的大小。用户界面在整个查询过程中都是可响应的。

当然，这并不一定意味着，每次与数据库建立连接时，都要创建新线程。不过，用户应分析一下，看看是否可以移动数据库或应用程序，使它们处于同一台服务器上。而且，要保证这个组件不会被各种客户应用程序不停地调用。这样做会为每一次调用创建额外的线程，并消耗比预期更多的资源。有一些方法可以复用对象及其线程，而不必在每次调用对象时使用新线程。这些问题将在第 3 章和第 5 章介绍。

## 2.5　线程的陷阱

前面介绍了适合在应用程序中使用线程的两种情况。但是，在一些情况下，创建新线程是不利的。在此不会列出不适合创建新线程的所有情况，只是说明一下在什么情况下不适合创建新线程。本节主要介绍两种情况：第一种情况是执行顺序极其重要，第二种情况是代码中常见的一个错误——在循环中创建新线程。

### 2.5.1　再次访问的执行顺序

在本章前面的 do_something_thread.cs 示例中，创建了一些代码，说明执行流是从一个线程到另一个线程随机进行的。这看起来好像是先执行一个线程，在控制台中显示 10 行，再让下一个线程显示 15 行，然后返回第一个线程，执行 8 行。决定是否使用线程的一个常见错误是，自以为知道在线程的给定时间片中要执行多少代码。

下面用一个例子来说明这个问题。看上去线程 t1 会第一个结束，因为它最先启动，但这是错的。创建一个控制台应用程序 ExecutionOrder，将其启动对象设置为 Main()。构建并运行几次这个实例，会得到不一样的结果：

```
using System;
using System.Threading;

namespace Chapter_02
{
   public class ExecutionOrder
   {
      static Thread t1;
      static Thread t2;

      public static void WriteFinished(string threadName)
```

```
{
  switch(threadName)
  {
    case "T1":
      Console.WriteLine() ;
      Console.WriteLine("T1 Finished") ;
      break;
    case "T2":
      Console.WriteLine() ;
      Console.WriteLine("T2 Finished") ;
      break;
  }
}

public static void Main()
{
  t1 = new Thread(new ThreadStart(Increment));
  t2 = new Thread(new ThreadStart(Increment));
  t1.Name = "T1";
  t2.Name = "T2";

  t1.Start() ;
  t2.Start () ;
  Console.ReadLine() ;
}

public static void Increment()
{
    for(long i = 1; i <=1000000; i++)
    {
      if(i % 100000==0 )
        Console.Write(" {" + Thread.CurrentThread.Name + "}");
    }
    WriteFinished(Thread.CurrentThread.Name) ;
  }
 }
}
```

有时候，线程 t1 会先结束，然后线程 t2 将执行更多的代码，然后才结束。有时候线程 t2 会先结束，然后线程 t1 会执行到结束。不能指望线程按照启动的先后顺序结束。

本书后面将讨论如何同步线程，以指定的顺序执行。但是注意，同步并不是在默认情况下发生的。

这并不是惟一与执行顺序相关的问题。下面是示例代码的下一部分 ExecutionOrder2，它说明了未同步的线程对数据会有负面影响：

```
using System;
using System.Threading;
public class ExecutionOrder2
{
   static Thread tl;
   static Thread t2;
   static int iIncr;

public static void WriteFinished(string threadName)
{
   switch(threadName)
   {
      case "Tl":
         Console.WriteLine();
         Console.WriteLine("Tl Finished: iIncr = " + iIncr.ToString);
         break ;
      case "T2":
         Console.WriteLine() ;
         Console.WriteLine("T2 Finished: iIncr = " + iIncr.ToString) ;
         break;
   }
}

public static void Main()
{
   iIncr = 0;
   tl =newThread(new ThreadStart(Increment));
   t2=new Thread(new ThreadStart(Increment));
   tl.Name = "Tl";
   t2.Name = "T2";
   tl.Start() ;
   t2.Start() ;
   Console.Read() ;
}
```

```
public static void Increment()
{
   for(long i = 1; i <= 1000000; i++)
   {
      if(i% 100000 = 0)
         Console.WriteLine(" {" + Thread.CurrentThread.Name + "} " +
               iIncr.ToString());
   }
   iIncr++;
   WriteFinished(Thread.CurrentThread.Name) ;
   }
}
```

这是一个与 ExecutionOrder 非常相似的类。但是，这一次创建了一个共享递增计数器 iIncr。应用程序在移到 WriteFinished()方法之前要递增变量的值。如果执行几次这个应用程序，递增计数器的数值将改变不同的次数。稍后将说明如何同步这些线程。这两个示例提出了警告：线程不会按希望的默认顺序执行。但是，在这些情况下，可以使用同步技术，如使用前面讨论的 Join()方法。线程的同步将在本书的后面深入探讨。

## 2.5.2　循环中的线程

尝到线程的甜头后，人们常犯的另一个错误是在一个循环中创建并使用线程。下面是演示这个错误的代码示例，这种代码通常是由刚刚接触到线程概念的程序员编写的。在发生一个事件时，开发人员或系统管理员通常使用这个概念来发送通知。这个想法不错，但是在循环中使用线程实现它会导致很多问题。

注意：

运行这些代码可能导致系统不可使用。不要运行这些代码，除非不介意重新启动计算机，回收程序浪费的资源。

```
using System;
using System.Threading;
using System.Web.Mail;
using System.Collections;

public class LoopingThreads
{
```

```
public delegate void SendMail(string oMessageTo);

private class MyMail
{
   public string EmailTo;
   public string EmailFrom;
   public string EmailSubject;
   public string EmailBody;
   public SendMail SendThisEmail;          // Delegate instance

   public void Send()
   {
      System.Web.Mail.MailMessage oMail =
            new System.Web.Mail.MailMessage() ;
      oMail.To   = EmailTo;
      oMail.From = EmailFrom;
      oMail.Body = EmailBody;
      oMail.Subject = EmailSubject;
      oMail.BodyFormat = MailFormat.Text;
      SmtpMail.Send(oMail) ;
      SendThisEmail(EmailTo) ;
   }
}

public static Thread CreateEmail(SendMail oSendEmail,
   string EmailTo , string EmailFrom ,
   string EmailBody , string EmailSubject )
{
   MyMail oMail = new MyMail();
   oMail.EmailFrom = EmailFrom;
   oMail.EmailBody = EmailBody;
   oMail.EmailSubject = EmailSubject;
   oMail.EmailTo = EmailTo;
   oMail.SendThisEmail = oSendEmail;

   Thread t = new Thread(new ThreadStart(oMail.Send));
   return t;
  }
}
```

```
class Mailer
{
   public static void MailMethod(string oString)
   {
      Console.WriteLine("Sending Email: " + oString) ;
   }
}

public class DoMail
{
   static ArrayList al = new ArrayList() ;

public static void Main()
{
   for(int i = 1; i <= 1000; i++)
   {
      al.Add(i.ToString() + "@someplace.com") ;
   }
   SendAllEmail() ;
}

public static void SendAllEmail()
{
   int loopTo = al.Count - 1 ;
   for(int i = 0; i <= loopTo; i++)
   {
      Thread t = LoopingThreads.CreateEmail(
         new LoopingThreads.SendMail(Mailer.MailMethod),
         (string)al[i],
         "johndoe@somewhere.com",
         "Threading in a loop", "Mail Example");
         t.Start ();
         t.Join(Timeout.Infinite) ;
      }
   }
}
```

这部分代码可能比较复杂，因为这里还演示了如何使用委托和许多带有参数的类来调用线程。这是必要的，因为线程只能在不带参数的方法上创建入口。因此，程序员应负责创建代理方法，为另一个方法创建参数，并返回一个线程对象(在后面的章节中

详细介绍)。然后调用方法就可以使用对返回的 Thread 类的引用来启动执行。

现在集中讨论一下 SendAllEmail 方法。该方法在 ArrayList 对象中循环，并将参数发送给代理方法。为每一个要发送的 Email 启动一个新线程:

```
public static void SendAllEmail()
{
  int loopTo = al.Count - 1;
  for(int i = 0; i <= loopTo; i++)
  {
    Thread t = LoopingThreads.CreateEmail(
        new LoopingThreads.SendMail(Mailer.MailMethod),
        (string) al [i] ,
        "johndoe@somewhere.com",
        "Threading in a loop", "Mail Example");
      t.Start() ;
    t.Join(Timeout.Infinite) ;
  }
}
```

乍看上去，这很不错。为什么不在另一个线程上发送 Email 呢？它有时需要很长时间来处理，不是吗？是的，但问题是在线程之间来回切换将占用处理器的执行时间。到完成这个进程为止，分配到每个线程的时间片主要花费在拆包和打包线程本地存储上了。执行线程中的指令只花费了很少的时间。系统可能会完全锁住，无法发送 Email。更合理的解决方案是创建一个线程，并在该线程上执行 SendAllEmail 方法。另外，再使用一个含有固定数目线程的线程池。在这个实例中，当线程池中的一个线程完成后，它将产生下一个线程，发送另一封 Email。

一个通用的编程惯例是将工作放到队列中，由服务处理。例如，一家银行将基于 XML 的文件放到一个网络目录中，让在另一台服务器上运行的服务程序获得这个文件。这个服务将浏览目录，查找新的文件，然后一次处理一个文件。如果一次将多个文件放到目录中，服务也将依次处理文件。在一个典型的环境中，新文件很少放在这个目录中。因此，乍看上去，在找到一个文件时，应启动一个新线程。这可能是对的，但是，如果处理这些文件的服务程序停止了，会发生什么情况呢？如果因为网络问题在很长时间里阻止服务访问这个目录，会发生什么情况呢？目录中的文件会堆积起来。当再次启动服务时，或者允许服务再次访问目录时，每一个文件都会在服务上产生一个新线程，这会使服务器停止运行。

这个文件模型只是个例子，另一个类似的模型是使用 Microsoft BizTalk Server 或者 Microsoft Message Queue，它含有一个服务程序，用于处理队列中的数据项。所有这些实现方式都有相同的基本结构，在这里，实现代码并不是很重要。要记住的是，如果把工

作放到队列中，并且应该使用多线程，就可以考虑使用线程池。

## 2.6　本章小结

本章介绍了 System.Threading 命名空间，并详细分析了 Thread 类。还讨论了一些基本的方法，帮助用户在应用程序中使用多线程时，作出明智的决定。线程是需要资源的，在消耗那些资源前，要分析资源的使用对系统有什么影响，如何使系统开销减到最少。如果访问外部资源，比如网络共享或者远程数据库，就应该考虑创建一个线程。当计划执行一个长进程时，比如打印、I/O 操作或者后台数据处理，还应该考虑产生一个新线程。

不管处于何种情况，都要使线程数目保持最少。这样会减小处理器的开销，增加用来处理线程中指令的时间量，并减少应用程序必需的内存大小。

# 第3章 使用线程

第2章讨论了线程在开发多用户应用程序中扮演的重要角色。使用线程可以解决一些重要的问题，如使多个用户或客户机在同一时刻访问同一资源。然而在学习的过程中，我们忽略了一个需要指出的问题：如果一个用户改变了资源的状态，而在同一时刻另一个用户也想改变同一资源的状态，会出现什么情况？

举一个ATM的例子。X先生和X夫人决定从ATM上支取$1000，销掉他们的支票账号。遗憾的是他们忘记决定由谁来做此事了。更富有戏剧性的是，X先生和X夫人几乎在同一时刻从不同的ATM机上进入他们的支票账户。如果两个用户同时访问同一个账户，而应用程序不是线程安全的，那么两个ATM机就有可能发现支票账户上有足够的存量，并分配给每个用户$1000。两个用户触发两个线程同时访问账户数据库。在理想情况下，当一个用户试图修改其账户时，其他用户就不能在同一时刻访问该账户了。简言之，当用户访问账户，以修改与这个账户相关的信息时，该账户就应被锁定。

**注意：**

.NET Framework提供了特有的机制来处理这种问题。这种在任一时刻只允许一个线程访问资源的现象称作同步。开发人员可以利用同步技术，对重要的资源进行线程安全的访问。

## 3.1 为何要同步

.NET开发人员之所以需要在设计多线程应用程序时保持同步，主要有以下两个原因：

- 避免竞争条件
- 确保线程的安全

由于.NET Framework对线程提供了内置的支持，所以使所开发的类最终能够应用于多线程应用程序。所设计的每个类不必(也不需要)都是线程安全的，因为线程安全不是免费的。不过每次设计.NET类时，至少要考虑到线程安全。本章后面将介绍线程安全的代价以及何时使类具有线程安全性的指导原则。不必担心多线程对局部变量、方法参数和返回值的访问，因为这些变量驻留在堆栈中，本来就具备线程安全性。不过，只要很合适地设计类，实例和类变量都将是线程安全的。

在分析同步之前，先考虑一下在本章开始的那个 ATM 例子，例子中的 X 夫妇二人试图同时从同一账户中支取$1000。图 3-1 清晰地描述了从 ATM 取款的流程。一个线程访问了某资源，使该资源处于无效状态，与此同时，另一个线程使用处于无效状态的对象产生不良后果的情况称作竞争条件。为避免竞争条件，需要使 Withdraw()方法具有线程安全性，以便在任一时刻只有一个线程可以访问该方法。

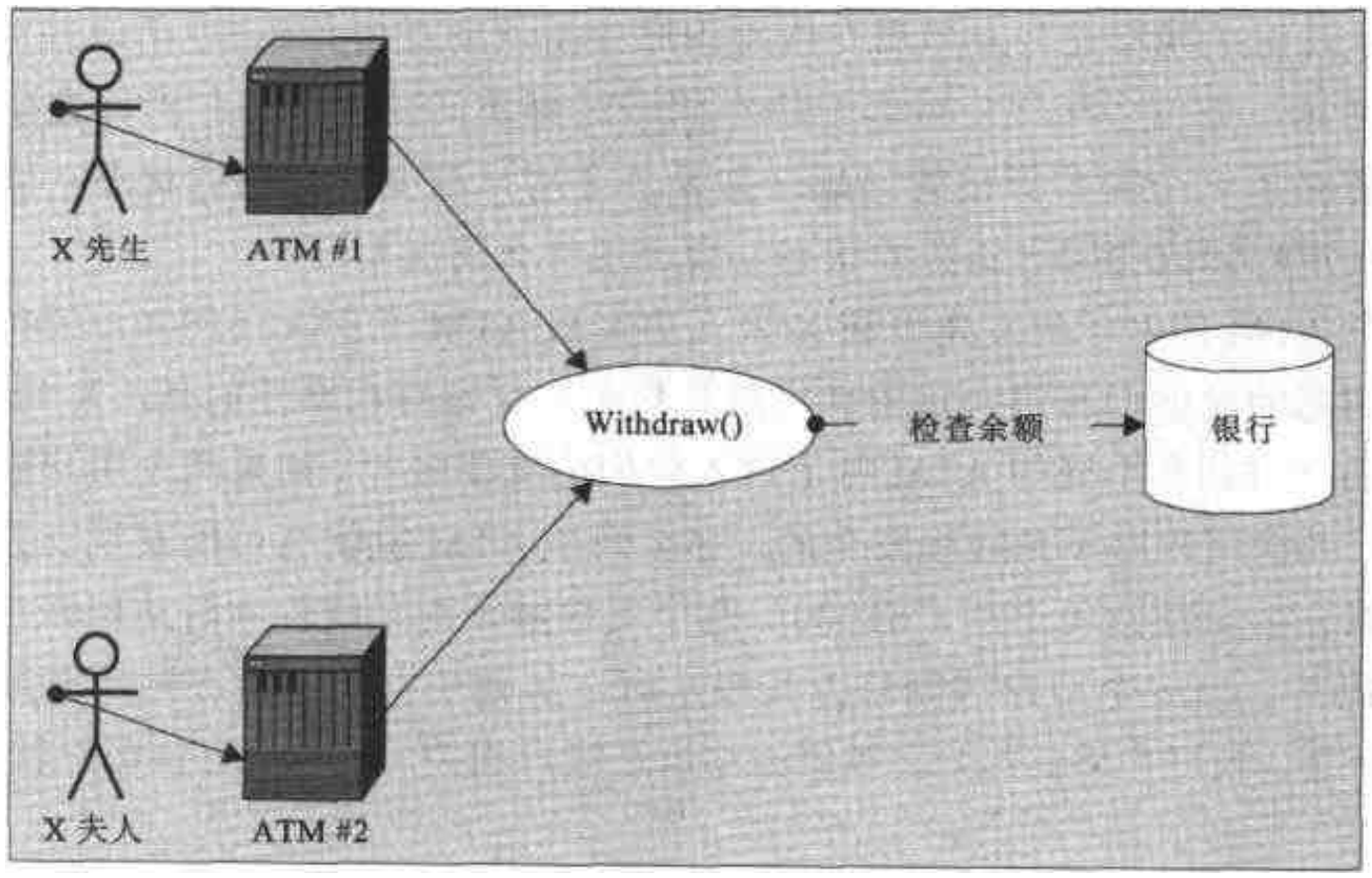


图　3-1

至少有 3 种方式能够使对象具有线程安全性：

- 同步代码中重要的部分
- 使对象不可改变
- 使用线程安全包装器

## 3.1.1　同步重要的代码段

为避免多个线程在同一时刻更新资源而引起的不良结果，需要限制对资源的访问，只允许在同一时刻只有一个线程能够更新资源，换句话说，就是使资源具有线程安全性。使对象或实例变量具有线程安全性的最简单方式是标识和同步重要的代码段。重要的代码段是指程序中多个线程可以在同一时刻访问以更新对象的状态的一段代码。例如，在上面的例子中，X 夫妇二人试图同时访问同一个 Withdraw()方法，Withdraw()方法就成为重要的代码段，需要具有线程安全性。最容易实施的方式是同步化方法 Withdraw()，以便在任一时刻只有一个线程(X 先生或 X 夫人)能够访问资源。在执行期间不能被中断的事务处理叫作原子操作。原子——其传统的含义是不可分的单元，原子操作的处理是作为一个完整单元执行的代码单元——就好像这些代码是一条处理器指

令一样。使 Withdraw()方法具有原子性，就可以确保在第一个线程更改完账户状态(在本例中为销户)之前，另一线程不可能修改同一账户的余额。下面列出的是一个非线程安全的 Account 类的伪代码：

```
public class Account
{
  public ApprovedOrNot Withdraw (Amount)
  {
    1. Make sure that the user has enough cash (Check the Balance)
    2. Update the Account with the new balance
    3. Send approval to the ATM
  }
}
```

下面列出了 Account 类的线程安全的伪代码版本：

```
public class Account
{
  public ApprovedOrNot Withdraw (Amount)
  {
    lock this section (access for only one thread)
    {
      1. Check the Account Balance
      2. Update the Account with the new balance
      3. Send approval to the ATM
    }
  }
}
```

在第一段代码中，两个或两个以上的线程可以同时进入重要的代码段，因此可能有两个线程在同一时刻核对账户余额，都收到账户余额$1000。因此，ATM 机有可能向两个用户各分发$1000，导致账户的透支。

而在第二段代码中，在任一时刻只允许一个线程访问重要的代码段。假定 X 先生的线程先执行 Withdraw()方法，执行完毕后，X 夫人的线程才开始执行。在 X 先生的线程开始执行 Withdraw()方法时，X 夫人的线程是不允许访问重要的代码段的，必须等到 X 先生的线程执行完毕。结果是 X 先生的线程核对账户余额，用新余额$0 更新账户，然后给 ATM 机返回相应的布尔值(此例中是 True)，实现现金支取。在支取现金之前，其他线程都不能访问 X 夫妇的 Account 对象的重要代码段。X 先生收到现金后，X 夫人的线程进入 withdraw()方法的重要代码段。现在，withdraw()方法核对账号余额，其

返回值为$0，于是、withdraw()方法返回 False 布尔值，表明余额不足，ATM 拒绝支取。

## 3.1.2　使账户对象不可改变

使对象具有线程安全性的另一种方式是使对象不可改变。不可变的对象是指该对象一旦创建，其状态就不能修改。只要在创建 Account 对象之后，不允许任何线程修改其状态，就可以实现这项操作。在这种方法中，要将读取实例变量的代码段和写入实例变量的代码段区分开。仅对读取实例变量的代码段不做改动，而对修改对象实例变量的代码段进行改变，不会修改当前对象的状态，而是会创建一个包含新状态的新对象，并返回对新对象的引用。在这种方法中，不需要锁定代码段，因为不可变对象的所有方法(只有构造函数)都不会写入对象的实例变量，所以，不可变对象在本质上就具有线程安全性。

## 3.1.3　使用线程安全包装器

使对象具有线程安全性的第三个方法是在对象上编写一个具有线程安全性的包装器类，而不是使对象本身具有线程安全性。对象将保持不变，而新的包装器类包括线程安全代码的同步部分。下面列出的是基于 Account 对象的包装器类：

```
public class AccountWrapper
{
  private Account _a;

  public AccountWrapper (Account a)
  {
    this._a = a;
  }

  public bool Withdraw(double amount)
  {
    lock(_a)
    {
      return this._a.Withdraw(amount) ;
    }
  }
}
```

AccountWrapper 类是 Account 类的线程安全包装器，Account 实例被声明为 AccountWrapper 类的 Private 实例变量，因此其他对象或线程不能访问 Account 变量。在这种方法中，Account 对象不具有任何线程安全的特性，因为线程安全性是由 AccountWrapper 类提供的。

在处理第三方的库，而库中的类不是为线程安全而设计时，就可以使用这种方法。例如，假设银行已经有一个 Account 类，用于为它的主机系统开发软件，为了保持一致，应使用同一个 Account 类来编写 ATM 软件。在银行提供的 Account 类的文档中，显然 Account 类不是线程安全的。另外，出于安全的原因，也不能访问 Account 的源代码。在这种情况下，必须采用线程安全包装器的方法，将线程安全的 AccountWrapper 类开发为 Account 类的扩展版。包装器用于将同步代码添加到非线程安全的资源中。所有的同步逻辑都在包装器类中，非线程安全的类则保持不变。

## 3.2 .NET 对同步的支持

.NET Framework 在 System.Threading、System.EnterpriseServices 和 System.Runtime.Compiler 命名空间中提供了一些类，程序员可以使用这些类开发线程安全的代码。表 3-1 对.NET Framework 中的一些同步类进行了简单的说明。

表 3-1

| 类 | 说　明 |
| --- | --- |
| Monitor | Monitor 对象用于锁定重要的代码段，这样，在任一时刻就只能有一个线程访问重要的代码段。Monitor 对象可帮助确保重要代码段的原子性 |
| Mutex | Mutex 对象类似于 Monitor 对象，但 Mutex 对象只允许一个线程访问在进程中共享的资源。Mutex 的重载构造函数可以用于指定 Mutex 的所属关系和名称 |
| AutoResetEvent, ManualResetEvent | AutoResetEvent 和 ManualResetEvent 用于通知一个或多个等待线程：事件已经触发。这些类都是不能继承的 |
| Interlocked | Interlocked 类包含如下方法：CompareExchange()、Decrement()、Exchange()和 Increment()，这些方法为同步访问由多个线程共享的变量提供了一种简单的机制 |
| SynchronizationAttribute | SynchronizationAttribute 确保同一时刻只有一个线程可以访问对象。这种同步进程是自动的，不需要显式锁定任何代码段 |
| MethodImplAttribute | 该属性告诉编译器方法该如何实现 |

## MethodImplAttribute 类

System.Runtime.CompilerService 命名空间包含的一些属性将影响 CLR 在运行期间的行为。MethodImplAttribute 就是这样一个属性，它会告诉 CLR 方法是如何实现的。MethodImplAttribute 的一个构造函数把 MethodImplOptions 枚举作为其参数，MethodImplOptions 枚举有一个字段 Synchronized，它指定在任一时刻只允许一个线程访问这个方法。这类似于前面例子中的 lock 关键字。下面列出了 MI.cs 程序，说明了如何使用这个属性来同步方法：

```
using System;
using System.Runtime.CompilerServices ;
using System.Threading;

namespace MethodImpl
{
  class MI
  {
    //This attribute locks the method for use
    //by one and only one thread at a time.
    [MethodImpl(MethodImplOptions.Synchronized)]
    public void doSomeWorkSync()
    {
      Console.WriteLine("doSomeWorkSync()" +
                        "--Lock held by Thread " +
                        Thread.CurrentThread.GetHashCode()) ;

      //When a thread sleeps, it still holds the lock
      Thread.Sleep(5 * 1000);
      Console.WriteLine("doSomeWorkSync()" +
                        "--Lock released by Thread " +
                        Thread.CurrentThread.GetHashCode()) ;
}

//This is a non synchronized method
public void doSomeWorkNoSync()
{
  Console .WriteLine ( " doSomeWorkNoSync () " +
                        "--Entered Thread is " +
```

```
                                Thread.CurrentThread.GetHashCode()) ;
    Thread.Sleep(5 * 1000);
    Console.WriteLine ("doSomeWorkNoSync()" +
                                "--Leaving Thread is " +
                                Thread.CurrentThread.GetHashCode()) ;
}

[STAThread]
static void Main(string[] args)
{
    MI m = new MI () ;

    //Delegate for Non-Synchronous operation
    ThreadStart tsNoSyncDelegate =
      new ThreadStart(m.doSomeWorkNoSync) ;

    //Delegate for Synchronous operation
    ThreadStart tsSyncDelegate =
      new ThreadStart(m.doSomeWorkSync) ;

    Thread t1 = new Thread(tsNoSyncDelegate);
    Thread t2 = new Thread(tsNoSyncDelegate);

    t1.Start() ;
    t2.Start() ;

    Thread t3 = new Thread(tsSyncDelegate);
    Thread t4 - new Thread(tsSyncDelegate);

        t3.Start() ;
        t4.Start() ;
      }
    }
}
```

上述代码的输出如下所示(计算机不同，这些结果就可能不同，因为线程 ID 是不同的)：

```
doSomeWorkNoSync()--Entered Thread is 2
doSomeWorkNoSync0--Entered Thread is 3
```

```
doSomeWorkSync()--Lock held by Thread 4
doSomeWorkNoSync0--Leaving Thread is 2
doSomeWorkNoSync()--Leaving Thread is 3
doSomeWorkSync0--Lock released by Thread 4
doSomeWorkSync0--Lock held by Thread 5
doSomeWorkSync0--Lock released by Thread 5
```

在上面的代码中，MI 类有两个方法 doSomeWorkSync()和 doSomeWorkNoSync()。MethodImpl 属性应用于 doSomeWorkSync()方法，用于同步该方法；而 doSomeWorkNoSync()保持不变，允许多个线程同时访问方法。在 Main()方法中，线程 t1 和 t2 访问未同步的方法，线程 t3 和 t4 访问已同步的方法。在这两个方法中，都添加了 Thread.Sleep()方法，用于为另一个正在等待的线程提供足够的时间进入方法，同时第一个线程仍在方法中。如果程序的情况是这样，线程 t1 和 t2 就可以同步进入 doSomeWorkNoSync()方法，而线程 t3 和 t4 中只有一个可以进入 doSomeWorkSync()方法。如果 t1 和 t2 有相同的优先级，则哪一个线程会优先执行就是随机的；.NET Framework 不保证线程执行的顺序。

仔细研究一下输出，注意线程 2(t1)和线程 3(t2)同时进入 doSomeWorkNoSync()方法，而一旦线程 4(t3)获得了 doSomeWorkSync()的锁定，线程 5(t4)就不能进入该方法了，只有线程 4(t3)释放了该方法的锁定，线程 5(t4)才能再次获得该锁定。

## 3.3　.NET 同步策略

公共语言基础结构(Common Language Infrastructure)提供了 3 种策略去同步访问实例、静态方法和实例字段，也就是：

- 同步上下文
- 同步代码区
- 手控同步

### 3.3.1　同步上下文

上下文是一组属性或使用规则，这组属性或使用规则对执行时相关的对象集合是通用的。能够添加的上下文属性包括有关同步、线程亲缘性和事务处理的策略。简言之，上下文把类似的对象组合在一起。这种策略将使用 SynchronizationAttribute 类对 ContextBoundObject 对象进行简单的自动同步。驻留在上下文中、符合上下文规则的对象称为上下文绑定对象。.NET 把同步锁和对象自动关联起来，在每种方法调用前锁定

对象，方法返回后释放对象(让其他等待着的线程访问)。由于线程同步和并发管理是开发人员遇到的最困难的任务，因此这种方法极大地提高了效率。

SynchronizationAttribute 类对缺少手工处理同步经验的程序员来说是很有用的，因为它包含了实例变量、实例方法和应用这个属性的类的实例字段。然而，它不处理静态字段和方法的同步。如果必须同步特定的代码块，它也不起作用。同步整个对象是对轻松使用必须付出的代价。在使用 System.EnterpriseAttribute 编程时，SynchronizationAttribute 非常方便，因为一个上下文(例如事务处理)中的对象由 COM+ 运行库组合在一起。

回到前面的 Account 示例上来，可以使用 SynchronizationAttribute 将伪代码的 Account 类变成具有线程安全性。下面列出了使用 SynchronizationAttribute 同步 Account 类的示例：

```
[SynchronizationAttribute(SynchronizationOption.Required)]
public class Account : ContextBoundObject
{
  public ApprovedOrNot Withdraw (Amount)
  {
    1. Check the Account Balance
    2. Update the Account with the new balance
    3. Send approval to the ATM
  }
}
```

SynchronizationAttribute 类有两个构造函数，一个构造函数不带参数，另一个构造函数的惟一参数是 SynchronizationOption 枚举。在使用默认的构造函数(不带参数)时，SynchronizationOption 默认为 SynchronizationOption.Required。其他选项是 Disabled、NotSupported、RequiresNew 和 Supported。表 3-2 描述了这些选项。

表 3-2

| 同步选项 | 说明 |
|---|---|
| Disabled | 忽略对象的同步请求，即对象不是线程安全的 |
| NotSupported | 创建组件时没有控制同步，即无论调用者处于什么状态，对象都不能参与任何同步操作 |
| Required | 确保所创建的所有对象都已同步 |
| RequiresNew | 无论调用者是什么，组件总是参与新的同步 |
| Supported | 使用这个选项的对象只有存在(这取决于调用者)时，才能参与同步操作 |

## 3.3.2　同步代码区

第二种同步策略是特定代码区的同步。这些特定的代码区是方法中重要的代码段，它们可以改变对象的状态，或者更新另一个资源(例如数据库，文件)。本节将介绍 Monitor 和 ReaderWriterLock 类。

### 1. Monitor 类

Monitor 用于同步代码区，其方式是使用 Monitor.Enter()方法获得一个锁，然后，使用 Monitor.Exit()方法释放该锁。锁的概念通常用于解释 Monitor 类。一个线程获得锁，其他线程就要等到该锁被释放后才能使用。一旦在代码区上获取了一个锁，就可以在 Monitor.Enter()和 Monitor.Exit()程序块内使用如下方法：

- Wait()—— 此方法用于释放对象上的锁，并暂停当前线程，直到它重新获得锁。
- Pulse()—— 此方法用于通知正在队列中等待的线程，对象的状态已经改变了。
- PulseAll()—— 此方法用于通知所有正在队列中等待的线程，对象的状态已经有了改变。

#### Enter()方法和 Exit()方法

注意，Monitor 方法是静态方法，能被 Monitor 类自身调用，而不是由该类的实例调用。在.NET Framework 中，每个对象都有一个与之相关的锁，可以获取和释放该锁，这样，在任一时刻仅有一个线程可以访问对象的实例变量和方法。与之类似，.NET Framework 中的每个对象也提供一种允许它处于等待状态的机制。正如锁的机制，设计这种机制的主要原因是帮助线程间的通信。如果一个线程进入对象的重要代码段，并需要一定的条件才能存在，而另一线程可以在该代码段中创建该条件，此时就需要这种机制。

诀窍在于，在任一时刻，重要的代码段只允许一个线程访问，当第一个线程进入代码段后，其他的线程就不能再进入了。如果第一个线程已经进入代码段，第二个线程如何在代码段中创建条件呢？例如，如果线程 A 从数据库中获取了一些数据，在线程 A 接收到所有的数据，并处理这些数据的过程中，另一个线程 B 必须调用 Wait()方法，等待线程 A 通知它何时才能接收数据。在接收到数据时，线程 A 就调用 Pulse()方法，通知线程 B，让线程 B 处理数据。这可以通过“等待和发出脉冲”机制来实现。第一个线程进入重要代码段，执行 Wait()方法。Wait()方法先释放锁，再使第一个线程进入等待状态，现在第二个线程进入代码段，修改必需的条件，然后调用 Pulse()方法来通知等待的线程，当条件满足后，就可以继续执行。接着第一个线程先重新获得锁，再从 Monitor.Wait()方法中返回，并从调用 Monitor.Wait()的地方继续执行。

任何两个线程都不能同时进入 Enter()方法。这类似于 ATM，在任一时间点上只允

许一个人操作，其他人只有等第一个人离开后才有机会操作。Enter 和 Exit 这两个名称选择得很合适。图 3-2 展示了 Monitor 的功能。

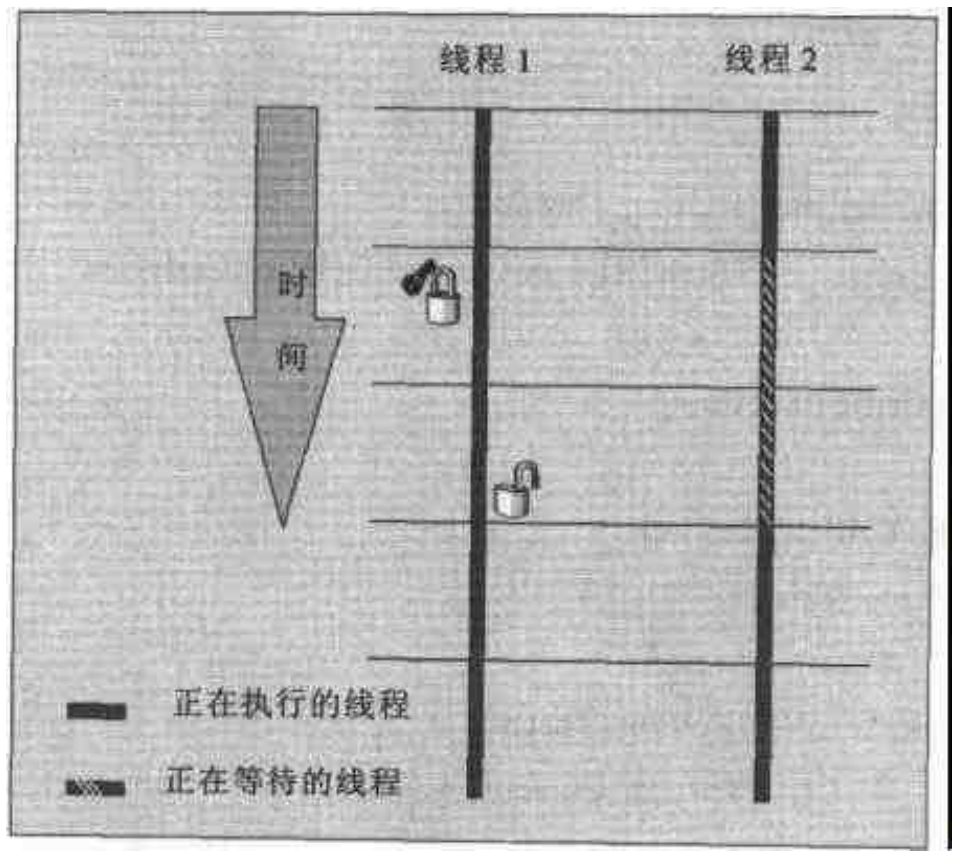


图 3-2

下面是一个使用 Enter() 和 Exit()方法的示例，即 MonitorEnterExit.cs 示例：

```
using System;
using System.Threading;

namespace MonitorEnterExit
{
  public class EnterExit
  {
    private int result = 0;

    public EnterExit()
    {
    }

    public void NonCriticalSection()
    {
      Console.WriteLine("Entered Thread " +
                        Thread.CurrentThread.GetHashCode()) ;

      for(int i=1; i <= 5; i++)
      {
```

```
        Console.WriteLine("Result = " + result++ + "    ThreadID "
                          + Thread. CurrentThread. GetHashCode ()) ;
        Thread.Sleep(1000) ;
    }

    Console.WriteLine("Exiting Thread " +
                      Thread.CurrentThread.GetHashCode()) ;
}
public void CriticalSection()
{
    //Enter the Critical Section
    Monitor.Enter(this) ;

    Console.WriteLine("Entered Thread " +
                      Thread. CurrentThread. GetHashCode ()) ;

    for (int i = 1; i <= 5 ; i++)
    {
        Console.WriteLine("Result = " + result++ + " ThreadID " +
                          Thread.CurrentThread.GetHashCode()) ;
        Thread.Sleep(1000) ;
    }

    Console.WriteLine("Exiting Thread " +
                      Thread.CurrentThread.GetHashCode()) ;

    //Exit the Critical Section
    Monitor.Exit(this) ;
}

public static void Main(String() args)
{
    EnterExit e = new EnterExit( ) ;

    if(args.Length > 0)
    {
        Thread ntl =
            new Thread(new ThreadStart(e.NonCriticalSection)) ;
        ntl.Start( );
```

```
    Thread nt2 =
        new Thread(new ThreadStart(e.NonCriticalSection));
    nt2.Start( ) ;
}
else
{
    Thread ct1 = new Thread(new ThreadStart(e.CriticalSection));
    ct1.Start( ) ;

    Thread ct2 = new Thread(new ThreadStart(e.CriticalSection));
    ct2.Start( ) ;
            }
        }
    }
}
```

在不提供输入参数的情况下运行此应用程序，会从 CriticalSection()方法中得到如下的输出结果：

```
Entered Thread 2
Result = 0    ThreadID 2
Result = 1    ThreadID 2
Result = 2    ThreadID 2
Result = 3    ThreadID 2
Result = 4    ThreadID 2
Exiting Thread 2
Entered Thread 3
Result = 5    ThreadID 3
Result = 6    ThreadID 3
Result = 7    ThreadID 3
Result = 8    ThreadID 3
Result = 9    ThreadID 3
Exiting Thread 3
```

如果提供输入参数，从 NonCriticalSection()方法返回的输出结果如下所示：

```
Entered Thread 2
Result = 0    ThreadID 2
Entered Thread 3
Result = 1    ThreadID 3
Result = 2    ThreadID 2
```

```
Result = 3    ThreadID 3
Result = 4    ThreadID 2
Result = 5    ThreadID 3
Result = 6    ThreadID 2
Result = 7    ThreadID 3
Result = 8    ThreadID 2
Result = 9    ThreadID 3
Exiting Thread 2
Exiting Thread 3
```

在上面的例子中，声明了一个 EnterExit 类，它包含一个全局变量 result 和两个方法：NonCriticalSection()和 CriticalSection()。在 NonCriticalSection()方法，没有指定任何监控器来锁定该代码，而在 CriticalSection()方法中，用监控器锁定了重要代码段。这两个方法都修改了 result 的值。

重要代码段定义为 Monitor.Enter(this)和 Monitor.Exit(this)之间的代码块。参数 this 指定应将锁保存在当前对象上。把哪个对象作为参数传递给 Enter()方法总是不好确定。在需要锁定对象，不让其他线程访问它时，应把 this 指针作为参数传递。例如，在前面讨论过的 AccountWrapper 例子中，就给 Monitor 传递了 Account 对象，而没有传递 AccountWrapper 的 this 指针。这是因为该例子的目的是锁定 Account 对象，而不是锁定 AccountWrapper 对象。我们不希望有多个线程访问 Account 对象，但并不介意多个线程访问 AccountWrapper 对象。

在 Main()方法中，根据所提供的参数运行相应的方法。如果没有提供参数，就使用 CriticalSection()方法，如果提供了参数，就使用 NonCriticalSection()方法。在两种情况下，从 Main()部分开始就有两个线程访问方法，同时修改 result 变量。随后尽管对它们进行了声明，但 For 循环和睡眠时间仍将使线程为争夺资源而竞争。

比较一下重要代码段和非重要代码段的输出结果，就会使重要代码段的概念变得清晰。查看一下 NonCriticalSection()方法的输出结果，线程 nt1 和 nt2 同时修改了 result 变量，产生一个混合型输出结果。这是因为在 NonCriticalSection()方法中没有锁，这个方法不是线程安全的。多个线程可以访问方法，同时全局变量也可以访问方法。相反，查看 CriticalSection()方法的输出结果，很明显，在线程 ct1 退出重要代码段之前，其他的线程(即此例中的 ct2)都不能访问重要代码段。

### Wait()和 Pulse()机制

Wait()和 Pulse()机制用于线程间的交互。当在一个对象上执行 Wait()时，正在访问该对象的线程就会进入等待状态，直到它得到一个唤醒的信号。Pulse()和 PulseAll()用于给等待线程发送信号。下面列出的是一个 Wait()和 Pulse()方法如何工作的例子，即 WaitAndPulse.cs 示例：

**注意：**

只能在 Enter()和 Exit()代码块中调用 Wait()和 Pulse()方法。

```
using System;
using System.Threading;

namespace WaitAndPulse
{
  public class LockMe
  {
  }

public class WaitPulsel
{
  private int result = 0;
  private LockMe _lM;

  public WaitPulsel()
  {
  }

  public WaitPulsel(LockMe l)
  {
    this._lM = 1;
  }

  public void CriticalSection()
  {
    Monitor.Enter(this._1M);
//Enter the Critical Section

Console.WriteLine("WaitPulsel:Entered Thread " +
                  Thread.CurrentThread.GetHashCode()) ;

for(int i = 1; i <= 5; i++)
{
  Monitor.Wait(this._IM) ;
  Console.WriteLine("WaitPulsel:WokeUp") ;
  Console.WriteLine("WaitPulsel:Result = " + result++ +
                    "   ThreadID " +
```

```
                                    Thread.CurrentThread.GetHashCode()) ;
            Monitor.Pulse(this._1M) ;
        }

        Console.WriteLine("WaitPulse1:Exiting Thread " +
                                Thread.CurrentThread.GetHashCode());

        //Exit the Critical Section
        Monitor.Exit(this._1M) ;
    }
}

public class WaitPulse2
{
    private int result = 0;
    internal LockMe _1M;

    public WaitPulse2()
    {
    }

    public WaitPulse2(LockMe 1)
    {
        this._1M = 1;
    }

    public void CriticalSection()
    {
    Monitor.Enter(this._1M) ;
    //Enter the Critical Section

    Console.WriteLine("WaitPulse2:Entered Thread " +
                            Thread.CurrentThread.GetHashCode()) ;
    for(int i = 1; i <= 5; i++)
    {
        Monitor.Pulse(this._1M) ;
        Console.WriteLine("WaitPulse2:Result = " + result++ +
                                " ThreadID " +
                                Thread.CurrentThread.GetHashCode());
        Monitor.Wait(this._1M) ;
```

```
            Console.WriteLine("WaitPulse2:WokeUp") ;
        }

        Console.WriteLine("WaitPulse2:Exiting Thread " +
                          Thread.CurrentThread.GetHashCode()) ;

        //Exit the Critical Section
        Monitor.Exit(this._1M) ;
    }
}

public class ClassForMain
{
    public static void Main(String[] args)
    {
        LockMe 1= new LockMe () ;

        WaitPulsel el= new WaitPulsel(1) ;
        WaitPulse2 e2 = new WaitPulse2(1) ;

        Thread tl = new Thread(new ThreadStart(el.CriticalSection))
        tl.Start( ) ;

        Thread t2 = new Thread(new ThreadStart(e2.CriticalSection))
        t2.Start( ) ;

        //Wait till the user enters something
        Console.ReadLine() ;
    }
}
```

WaitAndPulse 程序的输出结果为：

```
WaitPulse1: Entered Thread 2
WaitPulse2: Entered Thread 3
WaitPulse2: Result = 0    ThreadID 3
WaitPulse1: WokeUp
WaitPulse1: Result = 0    ThreadID 2
WaitPulse2: WokeUp
```

```
WaitPulse2: Result = 1    ThreadID 3
WaitPulse1: WokeUp
WaitPulse1: Result = 1    ThreadID 2
WaitPulse2: WokeUp
WaitPulse2: Result = 2    ThreadID 3
WaitPulse1: WokeUp
WaitPulse1: Result = 2    ThreadID 2
WaitPulse2: WokeUp
WaitPulse2: Result = 3    ThreadID 3
WaitPulse1: WokeUp
WaitPulse1: Result = 3    ThreadID 2
WaitPulse2: WokeUp
WaitPulse2: Result = 4    ThreadID 3
WaitPulse1: WokeUp
WaitPulse1: Result = 4    ThreadID 2
WaitPulse1: Exiting Thread 2
WaitPulse2: WokeUp
WaitPulse2: Exiting Thread 3
```

在 Main()方法中，创建一个名为 l 的 LockMe 对象。接着创建两个类型为 WaitPulse1 和 WaitPulse2 的对象，然后把它们作为委托传递，以便线程能够调用这两个对象的 CriticalSection()方法。注意，WaitPulse1 中的 LockMe 对象实例与 WaitPulse2 中的 LockMe 对象实例相同，因为对象按引用被传递给它们各自的构造函数。初始化对象后，创建两个线程 t1 和 t2，把它们分别传递给两个 CriticalSection()方法。

假设 WaitPulse1.CriticalSection()首先获得调用，则线程 t1 通过 LockMe 对象上的锁定进入该方法的重要代码段，然后在 For 循环中执行 Monitor.Wait()。执行 Monitor.Wait()方法后，线程 t1 等待另一个线程的运行时通知(Monitor.Pulse())，而被唤醒。锁定 LockMe 对象是希望在任一时刻都只允许一个线程访问共享的 LockMe 实例。

注意，当线程执行 Monitor.Wait()方法时，它会临时释放 LockMe 对象上的锁，以便其他线程可以访问它。线程 t1 进入等待状态后，线程 t2 就可自由访问 LockMe 对象。即使 LockMe 对象是个独立的对象((WaitPulse1 和 WaitPulse2)，这两个线程也均指向同一个对象引用。线程 t2 获得 LockMe 对象上的锁后，进入 WaitPulse2.CriticalSection()方法。它一进入 For 循环，就给在 LockMe 对象上等待的线程(此时是 t1)发送一个运行时通知(Monitor.Pulse())，之后进入睡眠状态。结果 t1 醒来，获得 LockMe 对象上的锁。接着线程 t1 访问 result 变量，并给在 LockMe 对象上等待的线程(此时是 t2)发送一个运行时通知。这个循环一直持续到 For 循环结束。

如果把程序的输出结果和上面的描述相对比，概念就非常明确了。注意每个 Enter()方法都应该伴随一个 Exit()方法，否则程序将陷入死循环。

Enter()将一个对象作为其参数。如果该对象参数为 null、方法变量或者值类型对象如整型值，系统就会抛出一个异常。

### TryEnter()方法

Monitor 类的 TryEnter()方法非常类似于 Enter()方法，它试图获得对象的独占锁，不过它不会像 Enter()方法那样暂停。如果线程成功进入，则 TryEnter()方法返回 True。

TryEnter()有 3 个重载方法，其中两个都带超时参数，表示等待锁定的时间。下面的例子 MonitorTryEnter.cs 说明了如何使用 TryEnter()：

```
using System;
using System.Threading;

namespace MonitorTryEnter
{
   public class TryEnter
   {
      public TryEnter()
      {
      }

      public void CriticalSection()
      {
         bool b = Monitor.TryEnter(this, 1000);
         Console.WriteLine("Thread " +
                           Thread.CurrentThread.GetHashCode() +
                           " TryEnter Value " + b) ;

         for (int i =1; i <= 3; i++)
         {
            Thread.Sleep(1000);
            Console.WriteLine(i + " " +
                              Thread.CurrentThread.GetHashCode()+ " ");
         }

      Monitor.Exit(this) ;
   }

   public static void Main()
   {
      TryEnter a = new TryEnter () ;
```

```
            Thread tl = new Thread(new ThreadStart(a.CriticalSection));
            Thread t2 = new Thread(new ThreadStart(a.CriticalSection));
            tl.Start ();
            t2.Start () ;
        }
    }
}
```

MonitorTryEnter 程序的输出结果如下：

```
Thread 2 TryEnter Value True
Thread 3 TryEnter Value False
1 2
1 3
2 2
2 3
3 2
3 3
```

在可能发生竞争，但不希望线程睡眠某个未指定的时间时，就可以使用 TryEnter()。例如对 ISP 的拨号连接。假定有两个应用程序 A 和 B，它们要使用同一个调制解调器拨号连接 ISP。目前只有一个可用的调制解调器连接，而且一旦建立起连接，就无法知道连接上的应用程序会在网上呆多久。假定应用程序 A 先拨号连接上 ISP，过了一段时间后，应用程序 B 也想拨号连接；应用程序 B 可能会无限期地等待下去，因为我们不知道应用程序 A 要在网上呆多久。在这种情况下，应用程序 B 就可以使用 TryEnter() 来确定调制解调器是否已被其他应用程序(本例中是 A)锁定，而不是使用 Enter()方法无限期地等待下去。

### Lock 语句

Lock 关键字可以用作 Monitor 方法的一个替换用法。下面的两部分代码是等价的：

```
Monitor.Enter(x)

    ...
Monitor.Exit(x)
```

```
lock(this)
{
  ...
}
```

下面的例子(Locking.cs)使用 Lock 关键字来代替显式的 Monitor 方法：

```
using System;
using System.Threading;

namespace Lock
{
  public class LockWord
  {
    private int result = 0;
    public void CriticalSection()
    {
      lock(this)
      {
        //Enter the Critical Section
        Console.WriteLine("Entered Thread " +
                          Thread.CurrentThread.GetHashCode()) ;

        for(int i = 1; i <= 5; i++)
        {
          Console.WriteLine("Result = " + result++ + " ThreadID " +
                            Thread. CurrentThread.GetHashCode () ) ;
          Thread.Sleep(1000) ;
        }

        Console.WriteLine("Exiting Thread " +
                          Thread.CurrentThread.GetHashCode()) ;
      }
    }

    public static void Main()
    {
      LockWord e = new LockWord() ;

      Thread t1 = new Thread(new ThreadStart(e.CriticalSection));
      t1.Start()

      Thread t2 = new Thread(New ThreadStart(e.CriticalSection));
      t2.Start() ;
    }
```

```
        }
    }
```

程序 Locking.cs 的输出结果与程序 MonitorEnterExit (当提供一个参数时)的输出一样：

```
Entered Thread 2
Result = 0    ThreadID 2
Result = 1    ThreadID 2
Result = 2    ThreadID 2
Result = 3    ThreadID 2
Result = 4    ThreadID 2
Exiting Thread 2
Entered Thread 3
Result = 5    ThreadID 3
Result = 6    ThreadID 3
Result = 7    ThreadID 3
Result = 8    ThreadID 3
Result = 9    ThreadID 3
Exiting Thread 3
```

### 2. ReaderWriterLock 类

ReaderWriterLock 定义了实现单写程序和多读程序语义的锁。这个类主要用于文件操作，即多个线程可以读取文件，但只能用一个线程来更新文件。ReaderWriterLock 类中 4 个主要的方法是：

- AcquireReaderLock()：该重载方法获得一个读程序锁，超时值使用整数或 TimeSpan。超时是可用于检测死锁的好工具。
- AcquireWriterLock()：该重载方法获得一个写程序锁，超时值使用整数或 TimeSpan。
- ReleaseReaderLock(): 释放读程序锁。
- ReleaseWriterLock(): 释放写程序锁。

**注意：**

使用 ReaderWriterLock 类时，任意数量的线程都可以同时安全地读取数据。只有当线程进行更新时，数据才被锁定。只有在没有占用锁的写程序线程时，读程序线程才能获得锁。只有在没有占用锁的读程序或写程序线程时，写程序线程才能获得锁。

下面给出程序 ReadWriteLock.cs，展示 ReaderWriterLock()锁的用法：

```
using System;
using System.Threading;

namespace ReadWriteLock
{
   public class ReadWrite
   {
      private ReaderWriterLock rwl;
      private int x;
      private int y;

   public ReadWrite()
   {
      rwl= new ReaderWriterLock() ;
   }

   public void ReadInts(ref int a, ref int b)
   {
      rwl.AcguireReaderLock(Timeout.Infinite) ;
      try
      {
         a = this.x;
         b= this.y;
      }
      finally
      {
         rwl.ReleaseReaderLock() ;
      }
   }

   public void WriteInts(int a, int b)
   {
      rwl.AcquireWriterLock(Timeout.Infinite) ;

      try
      {
      this.x = a;
      this.y = b;
```

```
            Console.WriteLine("x = " + this.x + " y = " + this.y +
                              " ThreadID = " +
                                 Thread.CurrentThread.GetHashCode()) ;
            }
            finally
            {
               rwl.ReleaseWriterLock() ;
            }
         }
      }

public class RWApp
{
   private ReadWrite rw = new ReadWrite() ;

   public static void Main(String[] arga)
   {
      RWApp e = new RWApp() ;

      //Writer Threads
      Thread wtl = new Thread(new ThreadStart(e.Write));
      wtl.Start() ;
      Thread wt2 = new Thread(new ThreadStart(e.Write));
      wt2.Start() ;

      //Reader Threads
      Thread rtl = new Thread(new ThreadStart(e.Read));
      rtl.Start() ;
      Thread rt2 = new Thread(new ThreadStart(e.Read));
      rt2.Start() ;
   }
   private void Write()
   {
      int a = 10;
      int b = 11;
      Console.WriteLine("********    Write **********");

      for (int i = 0; i < 5; i++)
```

```
    {
        this.rw.WriteInts(a++, b++) ;
        Thread.Sleep(1000) ;
    }
}

private void Read()
{
    int a = 10;
    int b = 11;

            Console.WriteLine("********   Read **********");
            for (int i = 0; i < 5; i++)
            {
                this.rw.ReadInts(ref a, ref b) ;
                Console. WriteLine ("For i="+i+"a="+a+"b="+b+
                                    " ThreadID = " +
                                    Thread.CurrentThread.GetHashCode() ) ;
                Thread.Sleep(l000) ;
            }
        }
    }
}
```

ReadWriteLock 的输出结果如下：

```
******** Read **********
For i=0 a=0 b=0 ThreadID = 5
******** Read **********
For i=0 a=0 b=0 ThreadID = 4
******** Write **********
x = 10 y = 11 ThreadID = 3

******** Write **********
x = 10 y = 11 ThreadID = 2
For i=l a=10 b=ll ThreadID = 4
x = 11 y = 12 ThreadID = 3
x = 11 y = 12 ThreadID = 2
For i=l a=ll b=12 ThreadID = 5
```

```
For i=2 a=11 b=12 ThreadID = 4
x = 12 y = 13 ThreadID = 3
x = 12 y = 13 ThreadID = 2
For i=2 a=12 b=13 ThreadID = 5
For i=3 a=12 b=13 ThreadID = 4
x = 13 y = 14 ThreadID = 3
x = 13 y = 14 ThreadID = 2
For i=3 a=13 b=14 ThreadID = 5
For i=4 a=13 b=14 ThreadID = 4
x = 14 y = 15 ThreadID = 3
x = 14 y = 15 ThreadID = 2
For i=4 a=14 b=15 ThreadID = 5
```

在上例中，线程 wt1 和 wt2 是在 WriteInts()方法中获取写程序锁的写程序线程，线程 rt1 和 rt2 是在 ReadInts()方法中获取读程序锁的读程序线程。在 WriteInts()方法中，实例变量 X 和 Y 分别变为新值 a 和 b。当线程 wt1 或 wt2 通过调用 AcquireWriterLock()获得写程序锁时，其他线程(包括读程序线程 rt1 和 rt2)就不能访问对象，直到线程 wt1 和 wt2 通过调用 ReleaseWriterLock()释放了写程序锁。在 ReadInts()方法中，线程 rt1 和 rt2 通过调用 AcquireReaderLock()方法获得读程序锁。在 ReadInts()方法中，线程 rt1 和 rt2 可以同时访问实例变量 X 和 Y。写程序线程(wt1 和 wt2)都无法访问对象，直到读程序线程释放读程序锁。在获取读程序锁后，读程序线程才能同时访问对象。

监控器对于仅计划读取数据而不修改数据的线程来说可能“太安全”了。在这方面，监控器的性能有所下降，而且，对于只读类型的访问，这种性能的降低不是必需的。ReaderWriterLock 类允许任意多的线程同时读取数据，提供了一种处理数据读写访问的最佳解决方案。仅当线程更新数据时，它才锁定数据。当且仅当没有写程序线程占用锁时，读程序线程才能获得锁。当且仅当没有读程序或写程序线程占用锁时，写程序线程才能获得锁。因此，ReaderWriterLock 类与重要代码段是一样的。除此之外，该类还支持一个可用于检测死锁的超时值。

## 3.3.3　手控同步

第三个同步策略涉及到了手控技术，而.NET Framework 提供了一套经典的技术，允许程序员使用类似于 WIN32 线程 API 的低级线程 API 创建和管理多线程应用程序。

表 3-3 给出了 System.Threading 命名空间中的一些可以用于手控同步的类。

表 3-3

| 类 | 说 明 |
| --- | --- |
| AutoResetEvent | AutoResetEvent 类可以使线程处于等待状态，直到通过调用 Set()方法某事件将它置于有信号状态为止。有信号状态是指没有线程在等待。在释放一个等待线程后，系统自动将 AutoResetEvent 类重置为无信号状态。如果没有正在等待的线程，事件对象的状态保持为有信号状态。指定 bManualReset 参数为 False 时，AutoResetEvent 类对应于 Win32 的 CreateEvent 调用 |
| ManualResetEvent | ManualResetEvent 类也用来使线程处于等待状态，直到通过调用 Set()方法某事件将它置于有信号状态。ManualResetEvent 对象的状态会一直保持有信号，直到 Reset()方法将它显式设置为无信号状态。指定 bManualReset 参数为 True 时，ManualResetEvent 对应于 Win32 的 CreateEvent 调用 |
| Mutex | Mutex 锁提供了跨进程和跨线程同步。如果没有线程占用 Mutex 锁，Mutex 的状态就置为有信号状态。Mutex 并不具有 Monitor 类的所有等待和发出脉冲功能，但可以创建在进程之间使用的命名互斥体(使用重载构造函数)。使用 Mutex 类优于使用 Monitor 类，因为 Mutex 类可跨进程使用，而 Monitor 类不行 |
| Interlocked | Interlocked 类为在多个线程之间共享的原子的、非暂停整体更新提供了方法。如果变量位于共享的内存中，不同进程的线程就可以使用这种机制 |

### 1. ManualResetEvent 类

ManualResetEvent 对象只能拥有两种状态之一：有信号(True)或无信号(False)。ManualResetEvent 类继承于 WaitHandle 类，其构造函数的参数可确定对象的初始状态。Set()和 Reset()方法返回一个布尔值，表示是否进行了成功的修改。

下面列出的程序 NETThreadEvents.cs 给出了无信号状态的 ManualResetEvent 类的用法。首先创建一个对象 mansig，并赋予 False 值。方法 WaitOne()将一直等待 mansig 变为 True 或超时为止。由于时间在等待中过去，且 mansig 的值没有被设为 True，所以程序就停止暂停并返回 False 值。

```
using System;
using System.Threading;

namespace NETThreadEvents
{
  public class NonSignaledManual
  {
    public static void Main()
```

```
        {
            ManualResetEvent mansig;
            mansig = new ManualResetEvent(false) ;
            Console.WriteLine("ManualResetEvent Before WaitOne " );
            bool b = mansig.WaitOne(1000,false);
            Console.WriteLine("ManualResetEvent After WaitOne " + b) ;
        }
    }
}
```

带 False 值的 NETThreadEvents 的输出结果如下：

```
ManualResetEvent Before WaitOne
ManualResetEvent After WaitOne False
```

在 NETThreadEvents.cs 中，构造了带 False 值的 ManualResetEvent 对象。布尔值 False 把 ManualResetEvent 对象的初始状态设置为无信号。接着调用基类 WaitHandle 的 WaitOne()方法。WaitOne()方法有两个参数。第一个参数是线程在 WaitOne()方法中等待的毫秒数；因此线程在退出前等待一秒。第二个参数是 exitContext，如果已经在上下文的同步域中，想退出同步上下文，或者要重新获得同步上下文，就把该参数设置为 True。

程序块在 WaitOne()方法中暂停一秒，然后因为超时而退出。ManualResetEvent 的状态仍然是 False，因而 WaitOne()返回的布尔值 b 是 False。现在看看如果在创建 ManualResetEvent 时就把它的状态设定为有信号(True)，将会发生什么情况：

```
using System;
using System.Threading;
namespace NETThreadEvents
{
  public class NonSignaledManual
  {
    public static void Main()
    {
      ManualResetEvent mansig;
      mansig = new ManualResetEvent(true) ;
      Console.WriteLine("ManualResetEvent Before WaitOne ");
      bool b = mansig.WaitOne(1000,false);
      Console.WriteLine("ManualResetEvent After WaitOne " + b) ;
      Console.ReadLine() ;
    }
```

```
    }
}
```

带 True 值的 NETThreadEvents 的输出结果如下：

```
ManualResetEvent Before WaitOne
ManualResetEvent After WaitOne True
```

将 ManualResetEvent 的初始状态修改为有信号的，即使指定超时值为 1000 毫秒，线程在 WaitOne()中也不会等待。当 ManualResetEvent 的状态为无信号状态时，线程将等待状态变为有信号状态，但 1000 毫秒后线程超时。状态已经是有信号的，因而线程没理由在 WaitOne()方法中等待。为了把 ManualResetEvent 的状态修改为无信号的，必须调用 ManualResetEvent 的 Reset()方法；为了把状态修改为有信号的，必须调用 Set()方法。

下面列出的程序 ManualReset.cs 展示了 Reset()方法的用法，接下来的程序 ManualSet.cs 展示了 Set()方法的用法：

```
using System;
using System.Threading;

namespace ManualReset
{
   class Reset
   {
      [STAThread]
      static void Main()
      {
         ManualResetEvent manRE;
         manRE = new ManualResetEvent(true);
         bool state = manRE.WaitOne(1000,true);
         Console.WriteLine("ManualResetEvent After first WaitOne " +
                           state) ;
         //Change the state to non-signaled
         manRE.Reset() ;
         state = manRE.WaitOne(5000,true);
         Console.WriteLine("ManualResetEvent After second WaitOne " +
                           state) ;
      }
   }
}
```

程序 ManualReset 的输出结果为：

```
ManualResetEvent After first WaitOne True
ManualResetEvent After second WaitOne False
```

在 ManualReset 中，ManualResetEvent 对象的构造函数将其状态设置为有信号的(True)，结果，线程不在第一个 WaitOne()方法中等待，并返回 True 值。接着将 ManualResetEvent 对象的状态重新设置为无信号的(False)，于是线程在超时之前必须等待 5 秒钟。

在程序 ManualSet.cs 中使用了 Set()方法，代码如下所示：

```
using System;
using System.Threading;

namespace ManualSet
{
   class Set
   {
      [STAThread]
      static void Main(string[ ] args)
      {
         ManualResetEvent manRE;
         manRE = new ManualResetEvent(false) ;
         Console.WriteLine("Before WaitOne") ;
         bool state = manRE.WaitOne(5000,true);
         Console.WriteLine("ManualResetEvent After first WaitOne " +
                           state) ;
         //Change the state to signaled
         manRE.Set() ;
         state = manRE.WaitOne(5000,true);
         Console.WriteLine("ManualResetEvent After second WaitOne " +
                           state) ;
      }
   }
}
```

程序 ManualSet 的输出结果如下：

```
Before WaitOne
ManualResetEvent After first WaitOne False
```

```
ManualResetEvent After second WaitOne True
```

在 ManualSet 中，将 ManualResetEvent 对象的初始状态设置为无信号的 (False)。结果是线程必须等待第一个 WaitOne()方法。接着用 Set()方法将其状态设置为无信号，这样线程就拒绝等待第二个 WaitOne()方法并退出程序。

正如 WaitOne()方法等待一个事件对象变成有信号的一样，WaitAll()方法将等待所有的事件对象都变成 True 或者有信号的，否则它将保持不变直到超时且 WaitAny()方法等待任一事件对象变成 True 或有信号的。

### 2. AutoResetEvent 类

AutoResetEvent 类的工作方式类似于 ManualResetEvent 类。它等待时间超时或者事件变成有信号状态，接着将此事件通知等待线程。ManualResetEvent 和 AutoResetEvent 的一个重要区别是 AutoResetEvent 在 WaitOne()方法中改变状态。下面给出的程序代码展示了 AutoResetEvent 类的用法：

```
using System;
using System.Threading;

namespace AutoReset
{
   class Auto
   {
      [STAThread]
      static void Main()
      {
         AutoResetEvent aRE;
         aRE = new AutoResetEvent(true) ;
         Console.WriteLine("Before First WaitOne ") ;
         bool state = aRE.WaitOne(1000,true);
         Console.WriteLine("After First WaitOne " + stated);
         state = aRE.WaitOne(5000,true);
         Console.WriteLine("After Second WaitOne " + state);
      }
   }
}
```

AutoReset 的输出结果与前面示例 ManualReset 的结果一样，如下所示：

```
Before first WaitOne
After first WaitOne True
```

```
After second WaitOne False
```

在 AutoReset 中，AutoResetEvent 和 ManualResetEvent 的差别很明显。事件对象的状态在第一个 WaitOne()方法中从有信号变为无信号，然后它在第二个 WaitOne()方法中又从无信号变为有信号。其结果是线程在第一个 WaitOne()方法中没有等待，而在第二个 WaitOne()方法中等待，直到超时。

### 3. Mutex 类

正如 ManualResetEvent 和 AutoResetEvent 类，Mutex 类也是从 WaitHandle 类派生出来的。除了可以用于进程间同步之外，Mutex 类非常类似于 Monitor 类。下面看一个示例 WroxMutex.cs:

```
using System;
using System.Threading;

namespace WroxMutex
{
   class NETMutex
   {
      static Mutex myMutex;

      public static void Main()
      {
         myMutex = new Mutex(true, "WROX");
         NETMutex nm = new NETMutex() ;
         Thread t = new Thread(new ThreadStart(nm.Run)) ;
         t.Start() ;
         Console.WriteLine("Thread Sleep for 5 sec");
         Thread.Sleep(5000) ;
         Console.WriteLine("Thread Woke Up");
         myMutex.ReleaseMutex();
         Console.WriteLine("Before WaitOne") ;
         myMutex.WaitOne() ;
         Console.WriteLine("Lock owned by Main Thread");
   }

   public void Run()
   {
      Console.WriteLine("In Run");
      myMutex.WaitOne();
```

```
            Console.WriteLine("Thread sleeping for 10 secs");
            Thread.Sleep(10000) ;
            Console.WriteLine("End of Run() method");
        }
    }
}
```

WroxMutex 的输出结果如下：

```
Thread will sleep for 5 seconds
In Run method
Thread Woke Up
Thread will sleep for 10 seconds
Before WaitOne
End of Run method
Lock owned by Main Thread
```

在 WroxMutex 中，用布尔值和 Mutex 名称字符串构造了一个 Mutex，其中布尔值表明调用线程在最初应该拥有 Mutex。然后创建一个调用 Run()方法的线程。Mutex 仍由主线程拥有。在 Run()函数中，线程 t 必须等到主线程释放 Mutex 锁。因而，在 Run()方法中，线程 t 等待调用 WaitOne()方法。睡眠 5 秒钟后，主线程释放 Mutex 锁。接着线程 t 得到 Mutex 锁，脱离睡眠状态。现在，在线程 t 释放 Mutex 锁或终止之前，Main()方法将无法获得 Mutex 锁。在此例中，线程 t 会超时，于是 Mutex 的所有权传递回主线程。

### 4. Interlocked 类

Interlocked 同步访问一个由许多线程共享的变量。这个操作以原子操作的方式实现。下面来看一个示例 WroxInterlocked.cs：

```
using System;
using System.Threading;

namespace Wroxinterlocked
{
  class WinterLocked
  {
    public ManualResetEvent a = new ManualResetEvent(false);
    private int i = 5;

    public void Run(object s)
```

```
    {
      Interlocked.Increment(ref i) ;
      Console.WriteLine("{0} {1}",
                        Thread.CurrentThread.GetHashCode(), i) ;
    }
}

public class MainApp
{
  public static void Main()
  {
    ManualResetEvent mR = new ManualResetEvent(false);
    WinterLocked wL = new WinterLocked() ;
    for(int i = 1; i <= 10; i++)
    {
      ThreadPool.QueueUserWorkItem(new WaitCallback(wL.Run), 1) ;
    }
    mR.WaitOne(10000, true);
  }
 }
}
```

WroxInterlocked 的输出结果如下：

```
Thread ID = 2 Count = 1
Thread ID = 2 Count = 2
Thread ID = 2 Count = 3
Thread ID = 2 Count = 4
Thread ID = 2 Count = 5
Thread ID = 2 Count = 6
Thread ID = 2 Count = 7
Thread ID = 2 Count = 8
Thread ID = 2 Count = 9
Thread ID = 2 Count = 10
```

WroxInterlocked 说明了 Interlocked 类的用法。我们以原子操作的方式递增全局变量 i 的值。除了 Increment()方法之外，该程序还有一个 Decrement()方法，该方法递减变量的值。Exchange()方法用同样的方式修改按引用传递给它的两个变量值。

### 5. 静态变量、方法和同步

静态变量和方法与同步锁中的实例变量和方法有不同的作用。静态变量是类变量，而属于一个对象的变量是对象或者实例变量。换句话说，静态变量只能有一个实例，静态方法由同一类的多个对象共享，同一类的每个对象都有自己的一组实例变量和方法。如果同步一个静态变量或者静态方法，锁就要应用于整个类上。其结果是，其他对象不能使用此类的静态变量。

#### ThreadStaticAttribute 类

ThreadStaticAttribute 用于在静态变量上为每个执行它的线程创建一个单独的变量，而不是在线程之间共享(默认行为)静态变量。这意味着，带有 ThreadStaticAttribute 的静态变量不会在访问该变量的不同线程之间共享。每个访问变量的线程都有该变量的一个副本。如果一个线程修改了变量，另一个访问变量的线程就不能看到这些变化。这种行为是有背于静态变量的默认行为的。总之，ThreadStaticAttribute 提供了最好的方法(静态和实例)。

下列程序(WroxStatic.cs)说明了 ThreadStaticAttribute 的用法：

```
using System;
using System.Threading;

namespace WroxStatic
{
  class ThreadStatic
  {
    [System.ThreadStaticAttribute ()]
    public static int x = 1;
    public static int y = 1;

public void Run()
{
  for (int i = 1; i <= 10; i++)
  {
    Thread t2 = Thread.CurrentThread ;
    x++;
    y++;
    Console. WriteLine("i = " + i +
                  " ThreadID = " + t2.GetHashCode() +
                  " x(static attribute)= " + x +
                  " y = " + y) ;
```

```
            Thread.Sleep(1000) ;
        }
    }
}

public class MainApp
{
    public static void Main()
    {
        ThreadStatic tS = new ThreadStatic();
        Thread t1 = new Thread(new ThreadStart(tS.Run));
        Thread t2 = new Thread(new ThreadStart(tS.Run));
        t1.Start() ;
        t2.Start() ;
    }
  }
}
```

WroxStatic 的输出结果如下：

```
i = 1 ThreadID = 2 x(static attribute)=1 y = 2
i = 1 ThreadID = 3 x(static attribute)=1 y = 3
i = 2 ThreadID = 2 x(static attribute)=2 y = 4
i = 2 ThreadID = 3 x(static attribute)=2 y = 5
i = 3 ThreadID = 2 xfstatic attribute)=3 y = 6
i = 3 ThreadID = 3 x(static attribute)=3 y = 7
i = 4 ThreadID = 2 x(static attribute)=4 y = 8
i = 4 ThreadID = 3 x(static attribute)=4 y = 9
i = 5 ThreadID = 2 x(static attribute)=5 y = 10
i = 5 ThreadID = 3 x(static attribute)=5 y = 11
i = 6 ThreadID = 2 x(static attribute)=6 y = 12
i = 6 ThreadID = 3 x(static attribute)=6 y = 13
i = 7 ThreadID = 2 x(static attribute)=7 y = 14
i =7 ThreadID = 3 x(static attribute)= 7 y = 15
i = 8 ThreadID = 2 x(static attribute)= 8 y = 16
i = 8 ThreadID = 3 x(static attribute)= 8 y = 17
i = 9 ThreadID = 2 x(static attribute)= 9 y = 18
i = 9 ThreadID = 3 x(static attribute)= 9 y = 19
i = 10 ThreadID = 2 x(static attribute)= 10 y = 20
i = 10 ThreadID = 3 x(static attribute)= 10 y = 21
```

静态变量是类变量，它的值在类的多个对象中保持不变。ThreadStaticAttribute 允许每个访问静态变量的线程拥有自己的副本。在 WroxStatic 中，变量 x 使 ThreadStaticAttribute 应用于它。结果，线程 t1 和 t2 将分别拥有静态变量 x 的一个副本，并且线程 t1 对变量 x 的修改对线程 t2 来说是不可见的。另一方面，线程 t1 对变量 y 的修改对线程 t2 来说是可见的。观察程序的输出结果，变量 x 对线程 t1 和 t2 来说递增的情况是不同的。

**注意：**

带有 ThreadStaticAttribute 的静态变量和实例变量之间的区别是，静态变量不要求对象访问它，但如果没有创建对象的实例就试图访问实例变量，则会抛出异常。

### 3.3.4 同步和性能

要获得同步锁，就需要一定的系统时间。其结果是，性能总是弱于非线程安全版本。当许多线程试图同时访问对象以获得同步锁时，整个应用系统的性能就会受到影响。当设计大型应用程序时，开发人员必须权衡这一点。重要的是，在执行一个彻底的测试之前，这些线程之间的竞争是不可见的。在设计大规模的多线程应用程序时，测试是非常重要的。开发人员必须权衡这些因素：

- 为安全起见，尽可能实现同步。这将使程序更慢，更糟糕的是，还没有单线程版本好。
- 为了性能，尽可能减少同步。

**注意：**

多线程设计是这两个因素的折衷方案。

## 3.4 小心死锁

尽管线程的安全性很重要，但是如果不适当地使用同步，就可能引起死锁。同样，理解什么是死锁并如何避免死锁非常重要。当两个或更多线程等待释放两个或更多的锁时，就会发生死锁，例如程序逻辑中的循环会使锁永远无法释放。图 3-3 展示了典型的死锁情形。

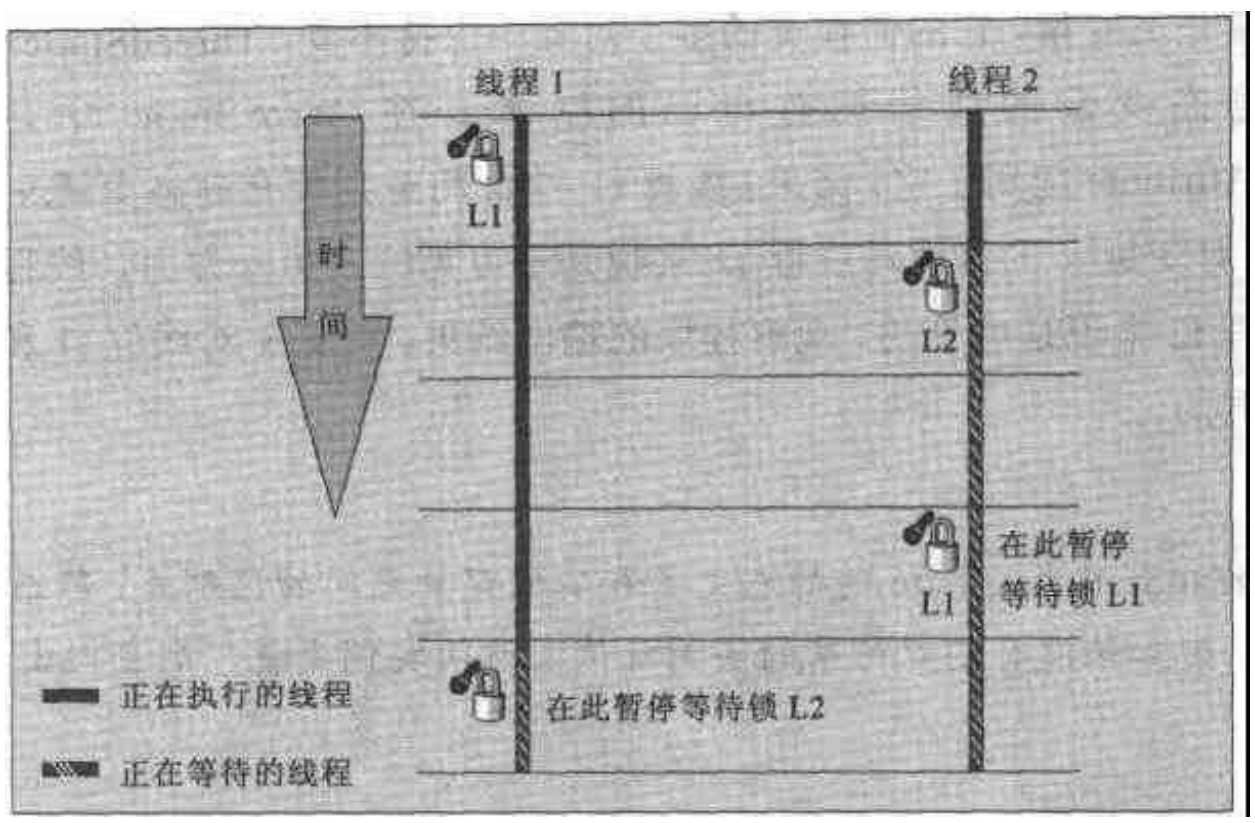


图　3-3

在图 3-3 中，线程 1 通过进入重要代码段获得对象的锁 L1。在该代码段中，假定线程 1 想获得锁 L2。线程 2 想获得锁 L2，并假定它想获得锁 L1。现在线程 1 不能取得锁 L2，因为线程 2 拥有锁 L2，线程 2 也不能取得锁 L1，因为线程 1 拥有锁 L1。结果，两个线程陷入无穷等待或死锁。

阻止潜在死锁的最好方法之一是，避免一次取得多个锁，这通常是可行的。然而，如果不能编码一次取得多个锁，就需要一个策略，来确保以一致的、定义好的次序获得多个锁。依据每个程序设计，避免死锁的同步策略可能有所不同。没有标准的策略可以用来避免所有的死锁。在大多数时候，在应用程序被部署在一个完全可伸缩的基础上之前，死锁不会被检测出来。如果在测试阶段检测出死锁，就很幸运了。

锁定策略中非常关键、但常常不被注意的元素就是文档。即使设计出一个好的同步策略来避免死锁，也不会花太多的精力来说明。最低限度是，每种方法都应有与之关联的文档，指定在方法中获得的锁，并描述该方法中的重要代码段。

下面是一个示例(Deadlock.cs)：

```
using System;
using System.Threading;

namespace DeadLock
{
  class DL
  {
    int field_1 = 0;
    private object lock_1 = new int [1] ;
```

```
int field_2 = 0;
private object lock_2 = new int [1] ;

public void First(int val)
{
   lock(lock_l)
      {
         Console.WriteLine("First:Acquired lock_l:" +
                           Thread.CurrentThread.GetHashCode() +
                           " Now Sleeping");

         //Try commenting Thread.Sleep()
         Thread.Sleep(1000) ;
         Console.WriteLine("First:Acquired lock_l:" +
                           Thread.CurrentThread.GetHashCode() +
                           " Now wants lock_2") ;

         lock(lock_2)
         {
            Console.WriteLine("First:Acquired lock_2:" +
                              Thread.CurrentThread.GetHashCode()) ;
            field_l = val;
            field_2 = val;
      }
   }
}

public void Second (int val)
{
   lock(lock_2)
   {
      Console .WriteLine ("Second:Acquired lock_2 : " +
                          Thread.CurrentThread.GetHashCode()) ;

      lock(lock 1)
      {
         Console.WriteLine("Second:Acquired lock_l:" +
                           Thread.CurrentThread.GetHashCode()) ;
         field_l = val;
         field_2 = val;
```

```
        }
      }
    }
}

public class MainApp
{
   DL d = new DL() ;

   public static void Main()
   {
   MainApp m = new MainApp() ;
   Thread t1 = new Thread(new ThreadStart(m.Run1));
   t1.Start() ;
   Thread t2 = new Thread(new ThreadStart(m.Run2));
   t2.Start() ;
}

public void Run1()
{
   this.d.First(10) ;
}

public void Run2()
{
   this.d.Second(10) ;
}
    }
}
```

Deadlock 程序的输出结果如下：

```
First:Acquired lock_1:2 Now Sleeping
Second:Acquired lock_2:3

First:Acquired lock_1:2 Now wants lock_2
```

在程序 DeadLock 中，线程 t1 调用了 first()方法，获得 lock_1 并睡眠一秒钟。在这其间，线程 t2 调用 second()方法，获得 lock_2。然后 t2 试图用同一方法获得 lock_1。但是 lock_1 由线程 t1 拥有，于是线程 t2 必须等到线程 t1 释放 lock_1。当线程 t1 醒来时，就试图获得 lock_2。现在 lock_2 由线程 t2 拥有，线程 t1 无法取得，直到线程

t2 释放 lock_2。这导致了死锁，并挂起程序。从方法 first()中注释掉 Thread.Sleep()行不会导致死锁，至少暂时不会。因为线程 t1 先于线程 t2 获得 lock_2。不过，在真实的环境中，并不使用 Thread.Sleep()方法，而是连接到一个数据库上，让线程 t2 先于 t1 获得 lock_2，这会导致死锁。这个例子表明了在多线程应用程序中设计一个好的锁定方案是多么重要。好的锁定方案会以一种定义完好的方式整合所有线程对锁的获得。在上面的示例中，线程 t1 不会获得 lock_2，直到线程 t2 释放它，或者，线程 t2 不会获得 lock_1 直到线程 t1 释放它。这些决定有赖于特定的应用程序，不能一概论之。锁定方案的测试同样很重要，因为，由于缺少应力和功能测试，通常会在部署的系统中发生死锁。

## 3.5 端到端的示例

本章的这一节将介绍两个大一些的示例。首先介绍如何创建线程安全包装器，然后介绍数据库连接池。

### 3.5.1 编写自己的线程安全包装器

希望编写自己的包装器，主要是因为可能不想使库中的每个类都具有线程安全性，因为同步伴随着性能损失。我们想给应用程序开发人员提供一个是否使用同步类的选择。应用程序开发人员既不想冒死锁的风险，也不想在单线程环境中使用线程安全的类，而以性能损失为代价。他们更乐于为库中的类使用内置的同步包装器，而不是编写一个包装器。像 System.Collections 命名空间里的 ArrayList 和 Hashtable 集合类就有这种功能。我们可以决定在 Hashtable 的初始化过程中是否使用线程安全的 Hashtable。调用 Hashtable 类的共享 Synchronized()方法，就可以初始化线程安全的 Hashtable，如下代码所示：

```
Hashtable h;
h = Hashtable.Synchronized(new Hashtable());
```

给应用程序开发人员这样一个选择是有益的。在这个示例中，我们将尝试开发一个类和类的同步包装器。我们将开发一个 BookCollection 库，图 3-4 给出了 BookCollection 库的 UML 示意图。

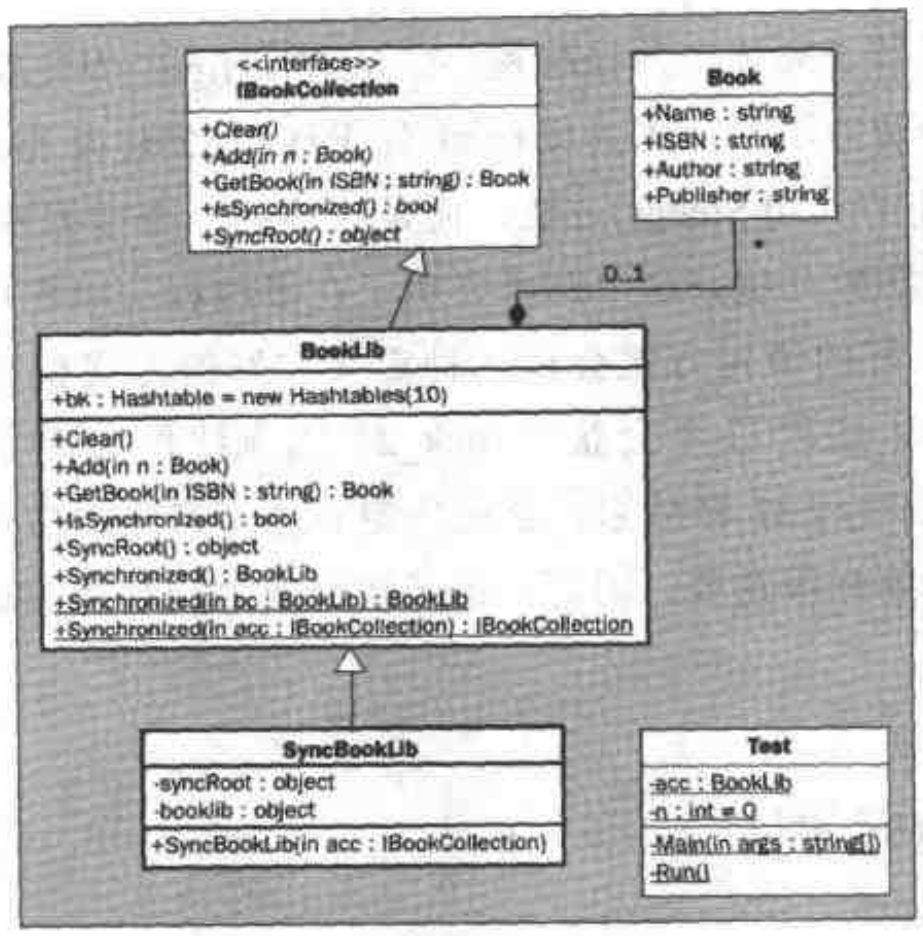


图　3-4

这个程序非常简单，但内部支持同步的概念非常重要。为库添加内在的同步支持，开发人员将能够为类选择同步和非同步环境。例如，不需要同步的程序员可以实例化一个对象，如下代码所示：

```
BookLib b = new BookLib()
```

在需要线程的环境里，使用该类的程序员可以使用线程安全包装器，如下所示：

```
BookLib b = new BookLib()
b = b.Synchronized ()
```

下面是带有同步包装器的完整 BookLib.cs 源程序：

```
using System;
using System.Threading;
using System.Collections;

interface IBookCollection
{
   void Clear() ;
   void Add(Book n) ;
   Book GetBook(string ISBN);
   bool IsSynchronized { get; }
   object SyncRoot { get; }
}
```

```
public class Book
{
   public string Name;
   public string ISBN;
   public string Author;
   public string Publisher;
{

class BookLib : IBookCollection
{
   internal Hashtable bk = new Hashtable(10);

public virtual void Clear()
{
   this.bk.Clear() ;
}

public virtual void Add(Book b)
{
   Console.WriteLine("Adding Book for ThreadID:" +
                        Thread.CurrentThread.GetHashCode()) ;
   Thread.Sleep(2000) ;
   bk.Add(b.ISBM, b) ;
}

public virtual Book GetBook(string ISBN)
{
   Console.WriteLine("Getting Book for ThreadID:" +
                        Thread.CurrentThread.GetHashCode ()) ;
   return (Book)bk[ISBN] ;
}

public virtual bool IsSynchronized
{
   get { return(false) ; }
}
public virtual object SyncRoot
{
   get { return(this); }
}
```

```
public BookLib Synchronized()
{
   return Synchronized(this) ;
}

public static BookLib Synchronized(BookLib bc)
{
   if (bc == null)
   {
      throw new ArgumentNullException("bc") ;
   }

   if (bc.GetType() == typeof(SyncBookLib))
   {
      throw new InvalidOperationException(
            "BookLib reference is already synchronized.");
   }

   return new SyncBookLib(bc);
}

public static IBookCollection Synchronized(IBookCollection acc)
{
   if (acc == null)
   {
      throw new ArgumentNullException("acc") ;
   }

      if (acc.GetType() == typeof(SyncBookLib))
      {
            throw new InvalidOperationException(
               "BookLib reference is already synchronized.");
      }

      return new SyncBookLib(acc);
   }
}

sealed class SyncBookLib : BookLib
```

```
{
   private object syncRoot;
   private object booklib;

   internal SyncBookLib(IBookCollection acc)
   {
      booklib = acc;
      syncRoot = acc.SyncRoot ;
   }
   public override void Clear()
   {
      lock(syncRoot)
      {
         base.Clear() ;
      }
   }

   public override void Add(Book b)
   {
      lock(syncRoot)
      {
         base.Add(b);
      }
   }

   public override Book GetBook(string ISBN)
   {
      lock(syncRoot)
      {
         return (Book)bk[ISBN];
      }
   }

   public override bool IsSynchronized
   {
      get{ return(true); }
   }

   public override object SyncRoot
   {
```

```
        get { return(syncRoot); }
    }
}

class Test
{
    private static BookLib acc;
    private static int n = 0;

    static void Main(string[ ] args)
    {
        acc = new BookLib() ;

if (args.Length > 0)
{
    acc = acc.Synchronized ();
    //OR BookLib.Synchronized(acc) ;
}

Thread[ ] threads = {new Thread(new ThreadStart(Run)),
                     new Thread(new ThreadStart(Run)),
                     new Thread(new ThreadStart(Run))};
foreach (Thread t in threads)
{
    t.Start() ;
}

foreach (Thread t in threads)
{
    t.Join() ;
}

for (int i = 0; i < n; i++)
{
    Book bk =acc.GetBook(i.ToString());

    if (bk !=null)
    {
        Console.WriteLine("Book : " + bk.Name) ;
        Console.WriteLine("ISBN : " + bk.ISBN);
```

```
            Console.WriteLine("Publisher : " + bk.Publisher);
            Console.WriteLine("Author : " + bk.Author);
        }
    }
    Console.WriteLine("Total Number of books added " + n) ;
}

static void Run()
{
    for (int i = 0; i < 2; i++)
    {
        Book bk = new Book() ;
        bk.Author = "Tejaswi Redkar";
        bk.Name = "A" + i;
        bk.Publisher = "Wrox";
        bk.ISBN = (n++).ToString();
        ace. Add (bk) ;
      }
    }
}
```

在上面的示例中，首先声明一个接口 IBookCollection，它具有下述处理书籍集合的方法和属性：

- Clear()——清除书籍集的方法
- Add()——为书籍集添加书的方法
- GetBook()——从书籍集中取得书的方法
- IsSynchronized()——用于检查书籍集是否同步的只读属性
- SyncRoot()——用于取得书籍集的同步根的只读属性

接下来声明 Book 类，表示书籍集中的一本书。例如，书籍集可以是一个库或一个书店，但 Book 类的表示在两者中是一样的。

BookLib 类实现了 IBookCollection 接口。因此，BookLib 类必须实现 IBookCollection 接口的所有方法和属性。声明一个名为 bk 的 Hashtable 作为包含所有书籍的集合。Book 对象的 Key 是书的 ISBN 号码。在 Add()方法中，为 Hashtable 添加一个 Book 对象。在 GetBook()方法中，如果提供了 ISBN 号码，就检索 Book 对象。

现在必须解决同步问题。在 Synchronized()方法中，创建一个 SyncBookLib 类型的对象，并返回对它的引用。SyncBookLib 是 BookLib 类的同步版本。SyncBookLib 继承自 BookLib 类，因而继承了 BookLib 类已经实现的所有属性和方法。SyncBookLib 类和 BookLib 类之间的区别是在 SyncBookLib 类中，使用监控器锁定了所有的重要代码段

(使用 lock 关键字)。例如，Clear()、GetBook()和 Add()方法在它们的实现代码中都有锁，因而具有线程安全性，而在 BookLib 类中任何的方法都没有锁。

在 Test 类中，如果传递任一命令行参数，就创建一个同步的 BookLib。如果没有传递命令行参数，就创建一个非线程安全的 BookLib 对象。接着，创建 3 个线程，为书籍库添加一些书籍。当运行应用程序时，同步的 BookLib 的执行和非同步的 BookLib 的执行有非常明显的区别。在同步版本中，任一时刻只有一个线程可以访问该库。另两个线程必须等待，直到第一个线程完成向 BookLib 中添加书籍的操作。如果使用非同步版本，情况就不是这样了，所有线程都可以并发访问 BookLib 对象实例。

带有命令行参数的(线程安全的)BookLib 的输出结果如下：

```
Adding Book for ThreadID:2
Adding Book for ThreadID:3
Adding Book for ThreadID:4
Adding Book for ThreadID:2
Adding Book for ThreadID:3
Adding Book for ThreadID:4
Book : A0
ISBN : 0
ISBN : Wrox
Author : Tejaswi Redkar
Book : A0
ISBN : 1
ISBN : Wrox
Author : Tejaswi Redkar
Book : A0
ISBN : 2
ISBN : Wrox
Author : Tejaswi Redkar
Book : A1
ISBN : 3
ISBN : Wrox
Author : Tejaswi Redkar
Book : A1
ISBN : 4
ISBN : Wrox
Author : Tejaswi Redkar
Book : A1
```

```
ISBN : 5
ISBN : Wrox
Author : Tejaswi Redkar
Total Number of books added 6
```

没有命令行参数的(非线程安全的)BookLib 的输出结果如下：

```
Adding Book for ThreadID:3
Adding Book for ThreadID:4
Adding Book for ThreadID:2
Adding Book for ThreadID:3
Adding Book for ThreadID:4
Adding Book for ThreadID:2
Getting Book for ThreadID:7
Book : A0
ISBN : 0
ISBN : Wrox
Author : Tejaswi Redkar
Getting Book for ThreadID:7
Book : A0
ISBN : 1
ISBN : Wrox
Author : Tejaswi Redkar
Getting Book for ThreadID:7
Book : A0
ISBN : 2
ISBN : Wrox
Author : Tejaswi Redkar
Getting Book for ThreadID:7
Book : A1
ISBN : 3
ISBN : Wrox
Author : Tejaswi Redkar
Getting Book for ThreadID:7
Book : A1
ISBN : 4
ISBN : Wrox
Author : Tejaswi Redkar
```

```
Getting Book for ThreadID:7
Book : A1
ISBN : 5
ISBN : Wrox
Author : Tejaswi Redkar
Total Number of books added 6
```

## 3.5.2 数据库连接池

对象池在企业软件开发中非常常见，为了提高应用程序的性能，必须控制对象实例的创建。例如，创建数据库连接对象，其代价非常昂贵，因为每次都需要连接数据库。于是，人们不是把资源浪费在为每次数据库调用实例化数据库连接对象上，而是创建数据库连接池，重复使用已经创建的连接对象，从而节省为每次数据库调用而创建新数据库连接所必需的时间和资源，赢得性能优势。

对象池类似于书库。书库保存了类似书籍的库。当对某类书的需求增加时，书库就会购买更多的书籍，否则读者就只能重复使用原来的书。在对象池中，首先要检查该池，看看对象是否已创建，并保存在池中，如果是，就取得池里的对象；否则就创建一个新的对象，并把它放在池里以备将来使用。在像企业级的 Java Beans(EJB) Servers、MTS/COM+，甚至.NET Framework 这些大型应用程序服务器中，都广泛使用了对象池。

本节将为数据库连接开发一个数据库连接池。创建数据库连接所耗费的资源很大。在一个典型的 Web 应用程序中，会有成千上万的用户试图在同一时刻访问 Web 站点。如果绝大多数的访问都需要数据库访问服务器动态数据，就要为每个用户创建新的数据库连接，这样将对应用程序的性能产生负面的影响。创建新对象需要分配新的内存空间。内存空间的分配会降低应用程序的性能，因此，Web 站点发送动态内容的速度将大大减慢，在到达一个临界点后系统就会崩溃。连接池可以保存已创建的对象集合，需要数据库连接的应用程序就可以从连接池中借用连接，在工作完成后返回给连接池，而不必创建新的数据库连接。一旦数据被提供给一个用户，连接就会返回到连接池，以备将来使用。

### 1. 实现池

首先用一个 UML 图来描述数据库连接池应用程序。图 3-5 展示了 ObjectPool 类和由 ObjectPool 类继承而来的 DBConnectionSingleton 类。

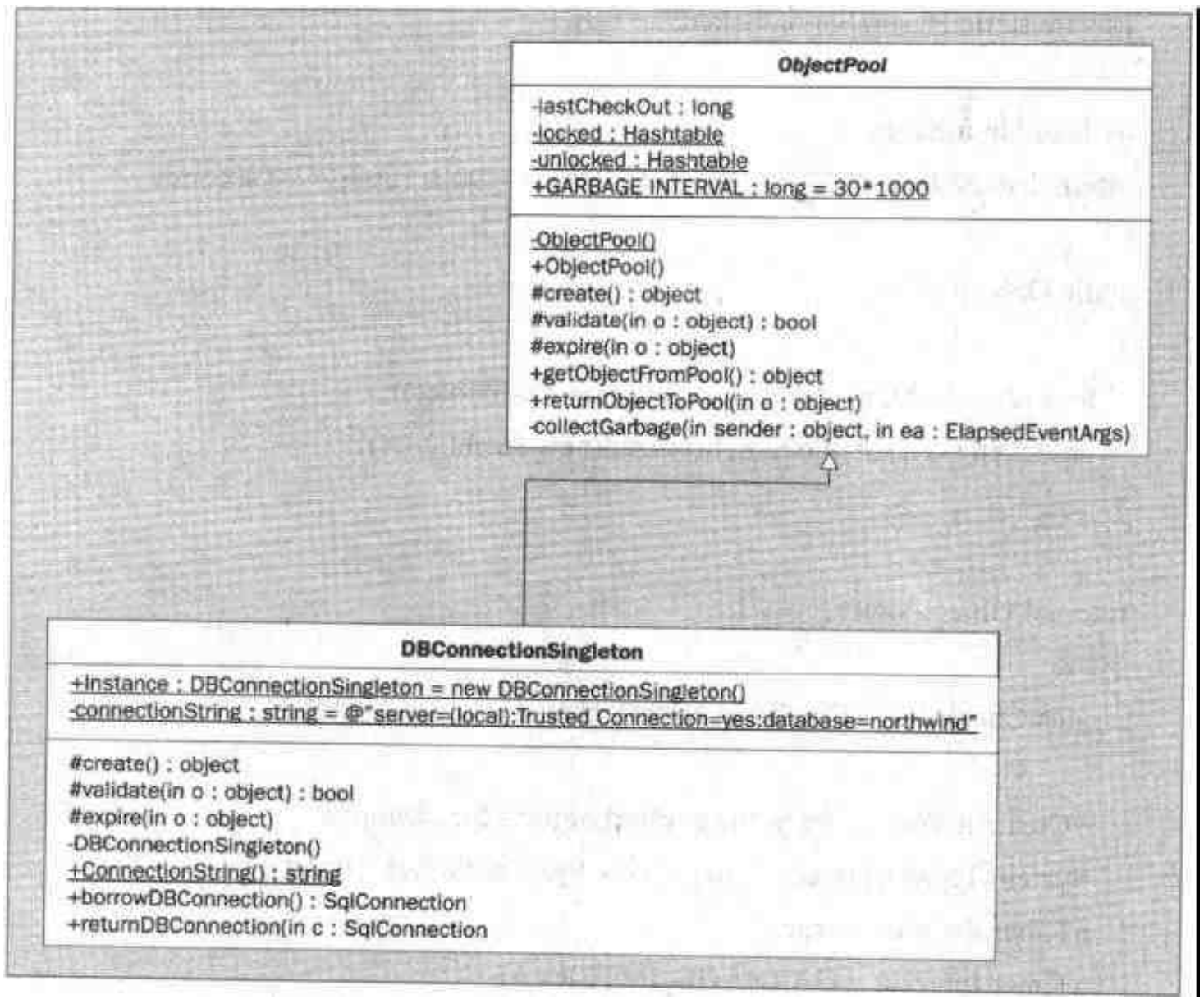


图 3-5

## ObjectPool 类

下面列出 ObjectPool 类的全部代码：

```
using System;
using System.Collections;
using System.Timers;

namespace WroxCS
{

  public abstract class ObjectPool
  {
    //Last Checkout time of any object from the pool.
    private long _lastCheckOut;

    //Hashtable of the checked-out objects
    private static Hashtable locked;
    //Hashtable of available objects
```

```
private static Hashtable unlocked;

//Clean-Up interval
internal static long GARBAGE_INTERVAL = 90 * 1000; // 90 seconds

static ObjectPool()
{
   locked = Hashtable.Synchronized(new Hashtable());
   unlocked = Hashtable.Synchronized(new Hashtable());
}

internal ObjectPool()
{
   _lastCheckOut = DateTime.Now.Ticks;

   //Create a Time to track the expired objects for cleanup.
   System.Timers.Timer aTimer = new System.Timers.Timer();
   aTimer.Enabled = true;
   aTimer.Interval = GARBAGE_IMTERVAL;
   aTimer.Elapsed += new
        System.Timers.ElapsedEventHandler(CollectGarbage) ;
}

protected abstract object Create() ;

protected abstract bool Validate(object o) ;

protected abstract void Expire(object o) ;

internal object GetObjectFromPool()
{
   long now = DateTime.Now.Ticks;
   _lastCheckOut = now;
   object o = null;

   lock(this)
   {
      try
      {
           foreach (DictionaryEntry myEntry in unlocked)
```

```
{
   o = myEntry.Key ;
   if (Validate(o))
   {
      unlocked.Remove(o) ;
      locked.Add(o, now);
      return(o) ;
   }
   else
   {
      unlocked.Remove(o) ;
                  Expire(o) ;
                  o = null;
               }
            }
         }
         catch (Exception){}
         o = Create() ;
         locked.Add(o, now);
   }
   return(o) ;
}

internal void ReturnObjectToPool(object o)
{
   if (o != null)
   {
      lock(this)
      {
         locked.Remove(o) ;
         unlocked.Add(o, DateTime.Now.Ticks) ;
      }
   }
}

private void CollectGarbage(object sender,
      System.Timers.ElapsedEventArgs ea)
{
   lock(this)
   {
```

```
object o;
long now = DateTime.Now.Ticks;
IDictionaryEnumerator e = unlocked.GetEnumerator();

try
{
   while(e.MoveNext())
   {
      o = e. Key ;

      if ((now - ((long) unlocked[ o])) >GARBAGE_INTERVAL)
      {
         unlocked.Remove(o) ;
         Expire(o) ;
         o – null;
      }
   }
  }
  catch (Exception){}
 }
}
}
```

ObjectPool 基类包含了两个重要的方法，一个是 getObjectFromPool()方法，该方法用于从池里取得一个对象，另一个是 returnObjectToPool()方法，该方法用于将对象返回到池里。对象池实现为两个散列表 locked 和 unlocked。Locked 散列表包括当前在使用的所有对象，unlocked 散列表包含了全部空闲和可用的对象。ObjectPool 还包含 3 个 MustOverride 方法，即 create()、validate()和 expire()；它们必须由派生的类来实现。

总之，在 ObjectPool 类中有 3 个重要的代码段：

- 从池里获取对象时，使用 getObjectFromPool()方法。向池中添加对象时需要一个锁，因为 locked 和 unlocked 散列表的上下文改变了，这里不希望出现任何竞争。
- 将对象返回到池中时，使用 returnObjectToPool()方法。将对象返回到池中时，也需要一个锁，因为 locked 和 unlocked 散列表的上下文改变了，有了一个可用的新对象。这里也不能有竞争，因为不希望多个线程同时访问同一个散列表。
- 从池里清除期满的对象时，使用 collectGarbage()方法。在这个方法中，将仔细检查 unlocked 散列表，找出并删除池中期满的对象。Unlocked 散列表的上下文可能会改变，所以期满对象的删除必须是原子操作。

在 getObjectFromPool()方法中，遍历 unlocked 散列表，得到第一个可用的对象。Validate()方法用来检查特定对象的有效性。根据池中对象的类型不同，该方法的执行代码也不同。例如，如果对象是一个数据库连接，对象池的派生类就需要实现 validate()方法，来检查对数据库的连接是打开的还是关闭的。如果对池中对象的有效性验证成功，就从 unlocked 散列表中删除该对象，把它放入 locked 散列表中。Locked 散列表包含当前正在使用的对象。如果验证失败，就使用 expire()方法删除对象。expire()方法也需由派生的类来实现，根据池中对象的类型不同，该方法的实现代码也不同。例如，在数据库连接的示例中，已过期的对象将关闭数据库连接。如果没有找到池中的对象，即 unlocked 散列表是空的，就用 create()方法创建一个新对象，把它放入 locked 散列表中。

returnObjectToPool()方法的实现代码非常简单，它就是从 locked 散列表中删除对象，把它放回 unlocked 散列表中，以备再利用。在整个循环过程中，必须考虑应用程序使用的内存。对象池与内存的使用成正比。因此，池中的对象越多，耗费的内存空间就越多。为了控制内存的使用，应定期回收池里的对象。为此，可以给池里的每个对象指定一个超时时间。如果池里的对象在超时时间内没有使用，就回收它。这样，对象池使用的内存量会根据系统的加载情况而变化。collectGarbage()方法用于处理池中对象回收。这个方法由 ObjectPool 构造函数中初始化的 aTimer 委托所调用。在此例中，在 GARBAGE_COLLECT 常量中把垃圾收集的间隔时间设置为 30 秒。

**注意：**

我们没有实现任何数据库连接代码，因此可以假设 ObjectPool 类可用于把所有类型的.NET Framework 对象放到池中。

### DBConnectionSingleton 类

DBConnectionSingleton 类是数据库连接对象池的实现方式。这个类的主要目的是为继承自 ObjectPool 类的 create()、validate()和 expire()方法提供数据库连接的实现方式。这个类还提供了 BorrowDBConnection()和 ReturnDBConnection()方法，来从对象池借用和返回数据库连接对象。

下面列出了 DBConnectionSingleton 类的完整代码：

```
using System;
using System.Data.SqlClient;

namespace WroxCS
{
  public sealed class DBConnectionSingleton : ObjectPool
  {
    private DBConnectionSingleton() {}
```

```
public static readonly DBConnectionSingleton Instance =
   new DBConnectionSingleton() ;

private static string _connectionString =
   @"server= (local);Trusted_Connection=yes;database=northwind" ;

public static string ConnectionString
{
   set
   {
      _connectionString = value;
   }
   get
   {
      return _connectionString;
   }
}

protected override object Create()
{
   SqlConnection temp = new SqlConnection(_connectionString) ;
   temp.Open () ;
   return(temp) ;
}

protected override bool Validate(object o)
{
   try
   {
      SqlConnection temp = (SqlConnection) o;
      return(
            !((temp.State.Equals(System.Data.ConnectionState.Closed)))) ;
      }
      catch (SqlException)
      {
         return false;
      }
}
```

```
protected override void Expire(object o)
{
   try
   {
      ((SqlConnection) o ).Close();
   }
   catch (SqlException)
   {
   }
}

   public SqlConnection BorrowDBConnection()
   {
      try
      {
         return((SqlConnection)base.GetObjectFromPool()) ;
      }
      catch (Exception e)
      {
         throw e;
      }
   }
   public void ReturnDBConnection(SqlConnection c)
   {
      base.ReturnObjectToPool(c) ;
   }
  }
}
```

当处理 SqlConnection 对象时，expire()方法会关闭 SqlConnection，create()方法可创建 SqlConnection，validate()方法可检查 SqlConnection 是否打开。使用 DBConnectionSingleton 对象实例对客户端应用程序隐藏同步问题。

### 使用 Singleton 模式的原因

Singleton 是一个流行的创造性设计模式，用于仅需一个对象实例的情况。Singleton 模式的目的如《Design Patterns》(ISBN:020170265-7)定义的那样，是确保一个类只有一个实例，并提供访问它的一个全局点。为了实现 Singleton，需要一个私有构造函数，使客户应用程序无论什么时候都不能创建新的对象，而使用静态的只读属性实例创建惟一的 Singleton 类实例。在 JIT 进程中，当(且仅当)有方法使用这个静态属性时，.NET

Framework 才初始化静态属性。如果没有使用该属性，就不创建实例。更准确地说，当调用者使用类的任一静态成员时，就构造或加载该类。这个特征称为傻瓜初始化(lazy initialization)，仅在第一次调用实例属性时创建对象。.NET Framework 保证了共享类型初始化的线程安全。因此不必担心 DBConnectionSingleton 对象的线程安全，因为在应用程序的生存期只创建它的一个实例。静态变量实例包含 DBConnectionSingleton 类对象的惟一实例。

### 使用数据库连接池

现在数据库连接池已可以使用了，可以使用本章下载代码中的 ObjectPoolTester 应用程序来测试它。

下面摘录的一些代码片段说明了如何实例化和使用数据库连接池：

```
// Initialize the Pool
DBConnectionSingleton pool;
pool = DBConnectionSingleton.Instance

// Set the ConnectionString of the DatabaseConnectionPool
DBConnectionSingleton.ConnectionString =
      "servers(local);User ID=sa;Password=;database=northwind"

// Borrow the SqlConnection object from the pool
Sql Connection myConnection = pool.BorrowDBConnection()

// Return the Connection to the pool after using it
pool.ReturnDBConnection(myConnection)
```

在上面的示例中，从 DBConnectionSingleton 类的实例属性中初始化了 DBConnectionSingleton 对象。如前所述，要利用 Singleton 设计模式，就只能有一个 DBConnectionSingleton 对象的实例。将数据库连接的 ConnectionString 属性设置为本地 SQL Server 机器上的 Northwind 数据库。现在，就可以用池中对象的 BorrowDBConnection()方法从对象池中借用数据库连接，然后调用池中对象的 ReturnDBConnection()方法返还数据库连接。图 3-6 显示了运行中的 ObjectPoolTester 应用程序，如果要研究连接池应用程序是如何工作的，最好的方法是打开 Visual Studio .NET 中的项目窗口，并以调试模式步进 ObjectPoolTester 应用程序。

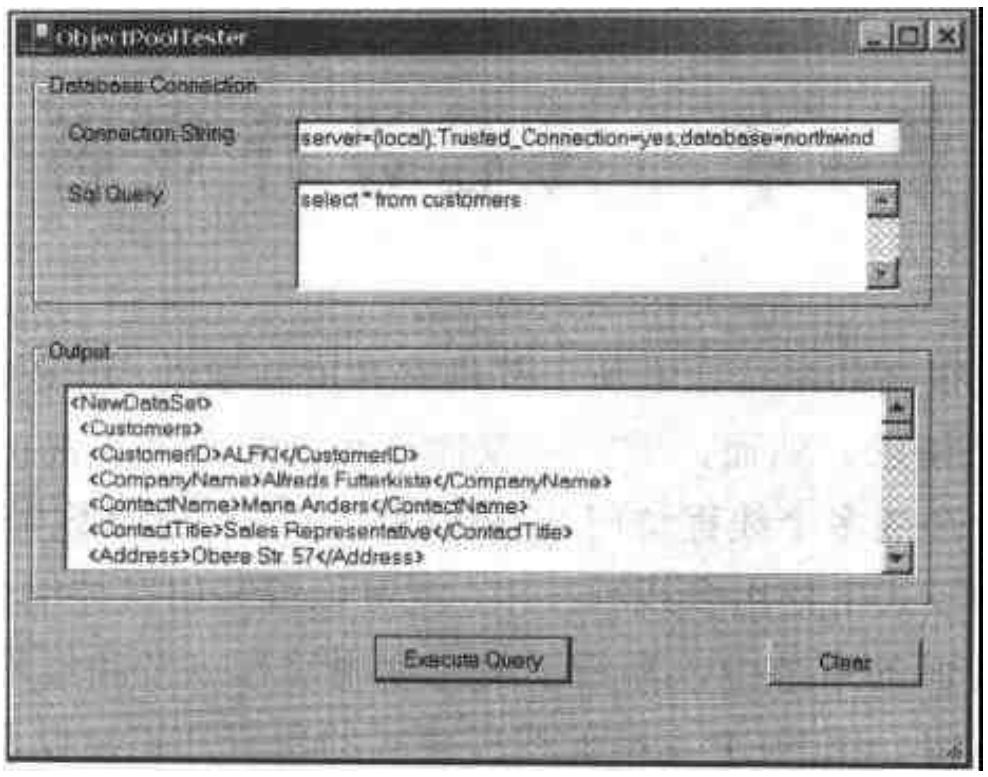


图 3-6

## 3.6 本章小结

同步是企业多线程计算环境中一个极其重要的概念。它广泛应用于诸如数据库、消息队列、甚至 Web 服务程序等流行的应用程序中。开发人员开发多线程应用程序时，需要十分清楚同步的概念。与其让开发人员受制于锁定功能，使每个对象都是线程安全的，不如专注于死锁问题，并尽可能多地在应用程序的设计阶段解决死锁问题。理解与同步相关的性能降低及它如何影响应用程序的整体性能也很重要。在本章中，除了介绍.NET Framework 中的同步支持外，还开发了两个有用的应用程序：

- 一个自定义的线程安全包装器。在此示例中，学习了如何把固有的同步支持添加到库里，允许应用程序开发人员选择是否使用同步。这将有助于开发人员专注于自己的应用程序，而不必担心库的线程安全问题。
- 数据库连接池。在这个示例中，开发了一个通用对象池，可以将任意类型的对象添加到池中。之后，开发了一个继承于对象池的数据库连接池。对象池可以重复用于将任意类型的对象放在池中。

# 第4章　线程设计规则

大多数可伸缩性很高的系统在本质上都具有高度并发性，并发性是指在同一时间存在对一个对象的多个请求。然而，编写并发性和线程安全性很高的代码具有较大的难度。也就是说，在一个或多个线程访问共享数据时，不会出现数据毁坏或混乱的情况。

如果在正常线程模式下使用多线程技术，就可以编写出可伸缩性很高的、以并发方式运行的代码。在前面的章节中，介绍了何时使用线程以及和线程相关的所有信息，包括线程陷阱。本章将介绍.NET 支持的线程模式的所有知识，以及如何利用这些线程模式，同时还将介绍可以在.NET 上层使用的有助于设计代码的一些模式。

在默认情况下，所有的.NET 应用程序都是多线程的，而在 VB6 中却并非如此。在 Windows Forms 应用程序中，有一种特殊的线程，称为 UI(用户界面)线程，它控制着与用户界面有关的所有函数，例如键盘激活和鼠标激活等。在 UI 线程上运行需要长时间连续运行、耗费大量时间的进程时，应用程序将逐渐失去响应能力。如果是在不同于默认 UI 线程的新生线程上执行任务，则应用程序将运行良好，用户界面将响应迅捷。但是，如果认为实现这个目标的惟一途径是创建许多线程对象，那就错了。除了多线程技术之外，还可以使用多种技术，包括实现异步编程和使用基于计时器的函数，这些内容在第 2 章中已经讲述过了。

## 4.1　应用程序中的多线程

如果用户曾使用.NET 之前的版本编过程，就知道 VB 在 COM 容器(例如 MTS 或 COM＋)内部支持多线程。尽管 VB5/6 支持多线程，但是它们所支持的线程模式是 STA(单线程单元，Single Threaded Apartments)。如果用户使用的是 Visual C++，就可以建立 MTA(多线程单元，Multi Threaded Apartments)和 STA 应用程序。然而，.NET Framework 中没有单元的概念，它是在应用程序域中管理所有线程的。在默认情况下，所有的.NET 应用程序都是多线程的，它们可以在任一时间访问所有的对象。所以必须特别关注托管代码中的静态资源。

.NET Framework 支持托管的和非托管的线程，并支持所有的 Win32 线程模式，如 STA 和 MTA。在从托管代码中访问 COM 组件时，传统的 COM 组件将创建非托管线程。.NET Framework 中的线程，无论是托管的还是非托管的，都是使用 Thread 对象创建的。

如果您曾使用 Win32 的 API 编写过多线程程序，就知道 Win32 支持用户界面线程和工作线程。正如第 1 章所述，这两个线程模式名称已经分别变成单元模型线程和自由线程。.NET Framework 支持两种基本的线程模式，一种是单元模型线程或者称为单线程单元(Single Threaded Apartment，STA)，另一种是自由线程或者称为多线程单元(MTA)。默认情况下，在.NET 中创建线程时，其线程模式为 MTA 模式。

**注意：**

*在访问基于 STA 的 COM 组件时，例如 VB6 COM 组件，只能使用 STA 线程模式。否则就不能将当前线程标记为 STA，因为这会大大降低应用程序的性能。*

重申一下前面的内容，对于同一上下文中的共享线程而言，单元是 AppDomain 中的逻辑容器。在激活的进程中创建对象时，对象就会驻留在 AppDomain 中，并创建上下文。

## 4.2　STA 线程模式

STA 线程单元的运行方式称为每客户对象(Object-per-Client)模式，其含义是创建 STA 线程单元的代码拥有它自己的线程。在任何一个单元之内都只能有一个线程，如图 4-1 所示。

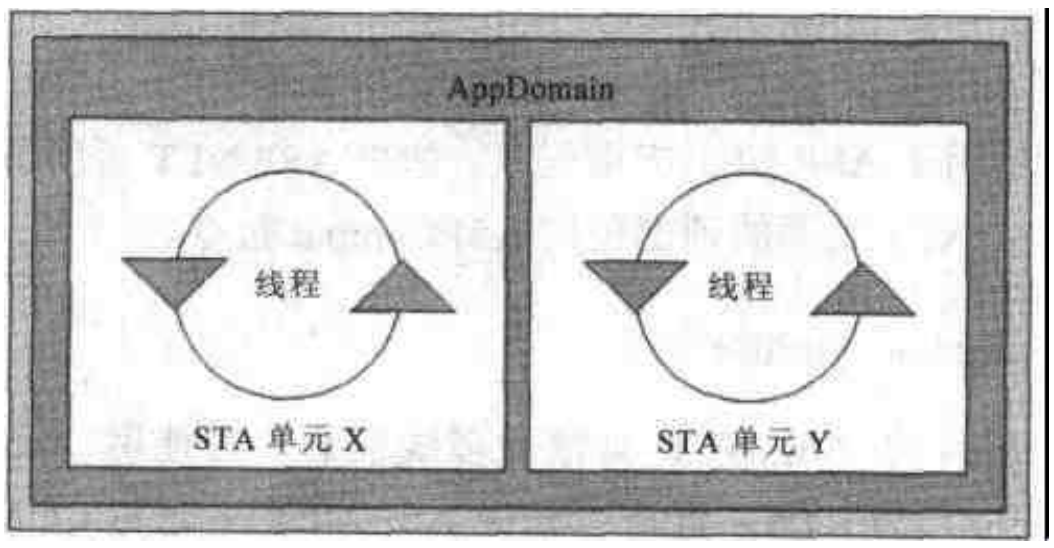


图　4-1

在 STA 线程模式中，对线程的所有调用都放到一个队列中，然后逐个处理这些调用。因此，STA 线程永远不会同时执行多个方法。STA 线程有自己的专有数据，它们不会在线程之间共享数据。这种特征使这一线程模式非常安全，避免了数据的损坏和同步问题。然而，这种特征也束缚了开发人员的手脚，并降低了性能，因为创建每个线程时，都要复制数据。

在图 4-1 中，AppDomain X 有两个 STA 线程 X 和 Y，它们都运行在 AppDomain X 内部，其中每个 STA 单元都只有一个线程。术语“线程亲合性(Thread Affinity)”用于

定义线程和创建线程的代码之间的关系。每当调用 STA 单元线程时，调用者和线程之间的调用将由 AppDomain 中的上下文处理，并由上下文维持线程的亲和性。

如果托管的应用程序打算使用非托管的传统 COM 组件，则在访问这些组件之前，应搞明白 COM 组件的线程模式。如果没有在应用程序中标记正确的线程模式，就可能会出现意想不到的错误和严重的错误。线程模式信息可以在注册表的 HKEY_CLASSES_ROOT\CLSID\{Class ID of the COM component} \InProcServer32 键中找到。

如果想指定使用单元线程模式，就应在Main()方法上应用STAThreadAttribute属性：

```
[STAThreadAttribute]
static void Main()
{
  …
}
```

如果试图在托管代码中访问传统的 STA 组件，就应使用该属性。否则，应将 Main() 标记为 MTAThreadAttribute：

```
[MTAThreadAttribute]
static void Main()
{
    …
}
```

这个规则同样适用于 ASP.NET 应用程序。如果 ASP.NET 页面访问一个 STA COM 组件，就必须在 ASP.NET 页面的顶部使用 ASPCompat 指令。

```
<%@ Page   AspCompat="true" %>
```

默认情况下，所有的 ASP.NET 页都是多线程的，当使用 AspCompat 指令时，ASP.NET 页面会被标记为 STA。这可以确保 ASP.NET 页面与 COM 组件的线程模式兼容。

**注意：**

将 ASP.NET 页面标记为运行在 STA 线程模式下时，会降低应用程序的性能。

**警告：**

如果使用的是 VB.NET，就可以使用 CreateObject 语句来实例化 COM 对象。因为 C#不支持后期绑定，在后期绑定模式下调用 COM 对象的惟一方式是使用反射技术。

## 4.3　MTA 线程模式

STA 和 MTA 线程单元的最大不同在于，在同一个单元中，MTA 单元可以同时运行多个线程，并使用所有可用的共享数据，如图 4-2 所示。

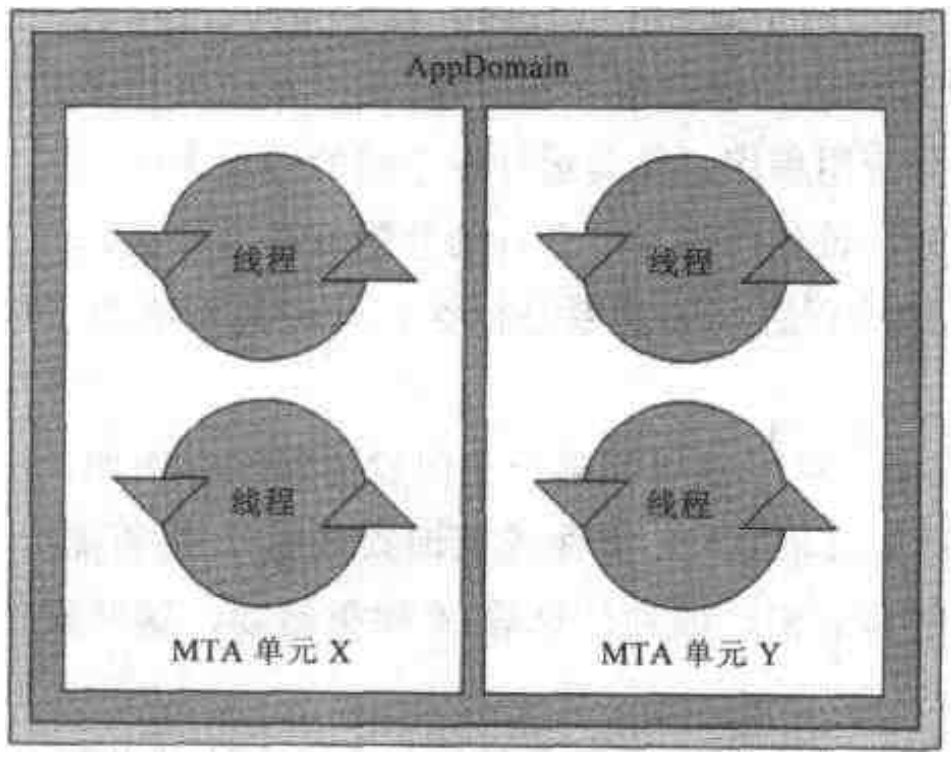


图　4-2

由于 MTA 模式支持同时执行多个线程，所以调用者应负责同步多个线程之间的全局数据。这些问题中的大部分内容在前面的章节中已经讲述过了。

### 4.3.1　指定线程模式

使用 Thread 类的 ApartmentState 属性可以设置线程模式。ApartmentState 枚举定义了.NET 支持的各种线程模式，如表 4-1 所示。

表 4-1

| 枚 举 值 | 含　义 |
| --- | --- |
| MTA | 创建多线程单元 |
| STA | 创建单线程单元 |
| Unknown | 没有设置 Thread 类的 apartment 属性 |

如前所述，如果访问基于 STA 线程的传统 COM 组件，应该将线程标记为 STA 线程。否则，线程模式就是默认的 MTA 线程模式。

## 4.3.2　设计线程应用程序

多线程的程序有两个或多个线程(控制流)。使用并发性，无论是否执行平行线程，都可以获得较高的程序性能。执行并发线程意味着同时执行两个或多个线程。在两个或多个处理器上同时执行两个或多个线程时，就称为并行性。

本节将讨论设计线程应考虑的事项和问题。在开发应用程序之前，应考虑下列问题：

(1) 是否可以细分应用程序，使其运行在不同的线程上？

(2) 如果可以细分，该如何进行细分，细分的标准是什么？

(3) 主线程和工作线程之间的关系是什么？这种关系确定了应用程序的任务相互之间如何关联。

通过检查应用程序，就可以找到第一个问题的答案。例如，应用程序是否需要执行大量的 I/O 操作(例如读取 XML 文件或查询数据库)？是否需要执行大量的 CPU 密集处理(例如数据加密和解密，散列计算等)？如果需要，这些操作就会堵塞应用程序的主线程。

如果要使应用程序的各部分使用单独的线程，还应该考虑下列问题：

(1) 每个标识的任务是否使用单独的全局资源

例如，如果为应用程序标识了两个潜在的线程，且这两个线程使用相同的全局资源，例如全局变量或者 DataSet 对象，那么在两个线程试图同时访问全局资源的情况下，就会使数据产生混乱或损毁，如前面章节所述。防止这一问题出现的惟一途径就是在全局资源上使用锁，让其他线程处于等待状态。如果两个任务使用相同的全局资源，就不应把任务分解为两个任务。对于某些资源，可以使用 Monitor 类来防止线程锁定。这一内容在第 3 章中也已经讲述过了。

(2) 经过多长时间才能阻塞线程

构建使用独立全局资源的应用程序，并非总是可能的。例如，假设应用程序中的两个任务依赖于一个全局 DataSet 对象。如果第一个任务需要大量的时间来填充 DataSet 对象(假设它需要从数据库中得到 50 000 行记录)，通常应锁定 Data Set 对象，以阻止并发问题的发生。下面是第一个任务的伪代码版本。

<1> 打开数据库连接

<2> 锁定全局的 DataSet 对象

<3> 执行查询

<4> 用数据库中的数据填充 DataSet

<5> 解除对 DataSet 对象的锁定

在这种情况下，第二个任务需要等待很长一段时间，直到第一个任务完成，并且解除了对 DataSet 的锁定之后，才能访问 DataSet 对象。这是一个潜在的问题，会使应用程序丧失并发性。解决这个问题有一个更好的方法：

<1> 打开数据库连接

<2> 执行查询

<3> 用数据库中的 50 000 行数据填充本地 DataSet

<4> 锁定全局 DataSet 对象

<5> 将本地数据库设置为全局数据库(DSGlobal = DSLocal)

<6> 解除对全局 DataSet 对象的锁定

通过这种方法，不应锁定全局 DataSet 对象，直到需要更新它为止，从而减少了等待全局对象被解除锁定的时间。

(3) 一个任务的执行是否依赖于另一个任务

例如，已标识的任务可以查询数据库，并在数据 DataGrid 控件中显示数据。可以将这个任务分成两个任务，第一个任务是查询数据库，第二个任务是在 DataGrid 中显示查询结果。第二个任务在第一个任务结束之前不会启动。因此，把查询和在 Data Grid 中显示数据分割成两个独立的、并行运行的任务是不可行的。这个问题的一个解决方案是，让第一个任务在结束时激发一个事件，事件发生时就创建一个新的线程。另外，还可以通过一个公用字段使用计时器来检查第一个任务是否完成，当第一个任务完成后就继续执行线程。

### 4.3.3　线程和关系

从一个多线程应用程序中生成的多个线程之间可能存在关系，也可能不存在关系。例如，在每个应用程序中都有一个主线程，它生成了其他线程。所以在应用程序中，主线程就变成了其他线程的控制器。可以定义多线程应用程序中线程之间的关系的方法很少。

- 主线程和工作线程模式
- 对等线程模式
- 管道线程模式

本节将详细讲述每一种模式，包括一些代码，以说明它们在应用程序中是如何实现的。

(1) 主线程和工作线程模式

这是一种最常见的模式，本书前面的例子使用的就是这种模式。如图 4-3 所示。

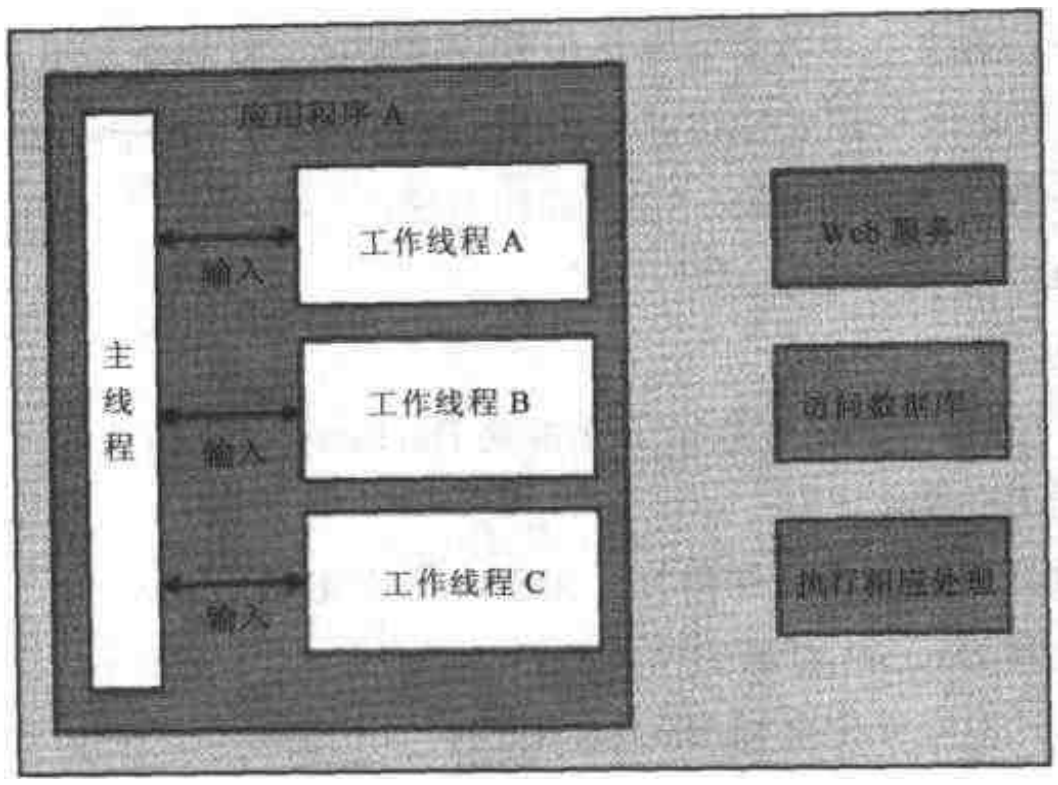


图　4-3

在主线程和工作线程模式中，主线程接收所有的输入，并将这些输入传递给其他线程，以执行某些任务。主线程需要等待工作线程的结束，也可能不需要。在这种模式中，工作线程不和输入源直接交互，它们是从主线程中读取输入数据的。例如，可以在 Windows Forms 应用程序中放置 3 个按钮，以引发 3 个事件。

- 从 Web 服务中获取数据
- 从数据库中获取数据
- 执行其他任务，例如分析 XML 文件

这是最简单的线程模式。主线程包含在 Main()方法中，这在客户 GUI 应用程序中很常见。

下面是演示这种模式的代码。所使用的窗体如图 4-4 所示。

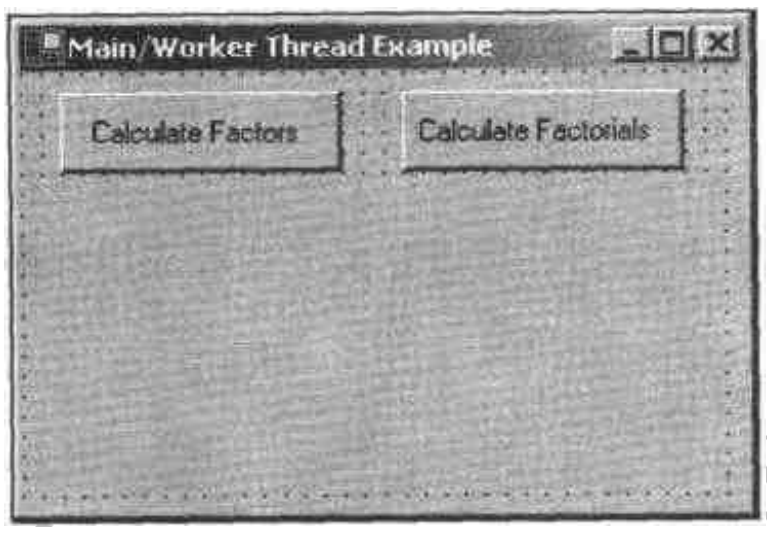


图　4-4

单击窗体中的一个按钮后，程序将激活一个工作线程，执行某些计算，返回结果，并显示在按钮下面的空白处。此处不详细说明 UI 代码；完整的代码可以从 Wrox Web 站点上下载，下面显示的是相关部分的代码：

```
public class MainWorker
{
   public ArrayList CalculateFactors(int number)
   {
      if (number < 3)
         return null;
      else
      {
         ArrayList factors = new ArrayList();
         factors.Add("1") ;
         for (int current = 2; current <= number - 1; current++)
         {
            if ((int)(Math.Floor(number / current) * current) == number)
               factors.Add(current.ToString()) ;
         }
         factors.Add(number.ToString()) ;
         return factors;
      }
   }

public long CalculateFactorial(int number)
{
   if (number < 0)
      return -1;

      if (number == 0)
            return 1;
      else
      {
         long returnValue =1;
         for (int current=1; current <= number; current++)
            returnValue *= current;
         return returnValue;
      }
   }
}
```

上述的方法是相当简单的。出于模块化的原因，它们被打包在一个类中。第一个方法返回一个 ArrayList，其中包含了传入数字的所有乘数，而第二个方法仅返回一个 Long 型的数字。阶乘积的结果瞬间就会变得很大，13 的阶乘积是 6 227 020 800。这个阶乘积

方法并没有占用处理器很长时间，但可以用来说明这种模式。

```
public frmCalculateO
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent() ;

  //Add any initialization after the InitializeComponent() call
  threadMethods = new MainWorker ();
}
```

该构造函数仅包含了新 MainWorker 对象的一个实例，用于这些方法中。下面是按钮的单击事件处理程序代码：

```
private void cmdFactors_Click(object sender, System.EventArgs e)
{
  Thread calculateFactors == new
      Thread(new ThreadStart(FactorsThread)) ;
  calculateFactors.Start() ;
}

void FactorsThread()
{
  ArrayList val = threadMethods.CalculateFactors(200);
  StringBuilder sb = new StringBuilder();

  for (int count = 0; count <= val.Count - 1; count++)
  {
    sb.Append((string)val[count]) ;
    if (count < val.Count - 1)
      sb.Append(", ") ;
  }

  //Create and invoke the delegate with the new value
  UpdateValue updVal = new UpdateValue(DisplayValue);
  String[ ] Arugs = {sb.ToString()} ;
  this.Invoke(updVal,Arugs) ;
}
```

cmdFactors_Click()方法使用 FactorsThread()方法实例化一个新线程，它对包含在MainWorker.CalculateFactors()方法中的返回值进行格式化。

**注意：**

这个方法需要封装起来，因为线程方法是不允许有返回值的。

```
private void cmdFactorial_Click(object sender, System.EventArgs e)
{
   Thread calculateFactorial =
        new Thread(new ThreadStart(PactorialThread));
   calculateFactorial.Start() ;
}

private void FactorialThread()
{
   long val = threadMethods.CalculateFactorial(20);
   //Create and invoke the delegate with the new value
   UpdateValue updVal = new UpdateValue (DisplayValue) ;
   String[ ] Arugs = {val.ToString()} ;
   this.Invoke(updVal,Arugs) ;
}
```

FactorsThread()方法比较简单，只要单击 cmdFactorial 按钮，主线程就会产生一个新线程，当得到返回值时更新 lblResult 文本标签。

这是一个简明的主线程/工作线程的例子，显然，很容易把这个示例改为处理数据库连接，或者其他费时的活动。

但是，使用这种模式时，必须关注和线程相关的几个问题。可以让线程生成线程，让线程访问同样的资源，在无限循环中使用线程。

**注意：**

这是最简单的模式，但也是需要开发人员完成大量工作的模式。

此外，线程是完全相互独立的，所有的线程完全受控于其父线程，在本示例中，则完全受控于主线程。

(2) 对等线程模式

下一个要讨论的线程模式是对等线程模式，在这种线程模式下，每个线程都从适当的来源接收其自己的输入，并对输入作出相应的处理。对等线程模式如图 4-5 所示。

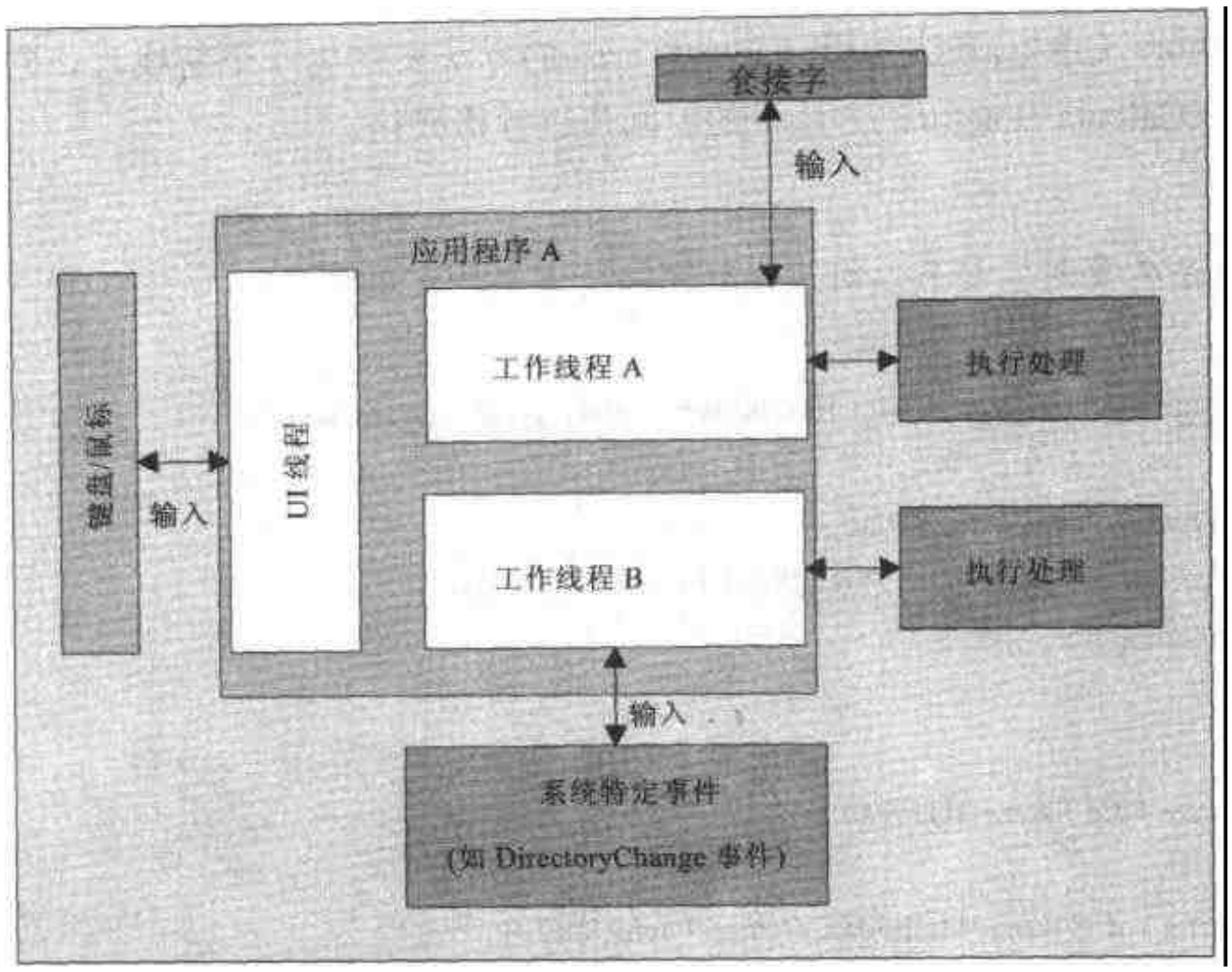


图　4-5

在图 4-5 中，UI 线程从键盘和鼠标处接收输入，并进行相应的处理。工作线程 A 监听特定的套接字，当接收到从套接字传递过来的输入信息时，就作出相应的处理。同样，工作线程 B 将等待系统事件，并作出相应的处理。在这种模式中，所有的线程都将并行执行，不会产生暂停或等待其他线程的情况。

修改前面的示例，使 CalculateFactors()方法注意阶乘线程何时完成，并找出这个数的阶乘积。在这个例子中将使用 8 的阶乘积。然而本例不使用套接字，改为使用一个变量。其原理和套接字是一样的：要么进行持续的监听，要么保存处理器周期，进行间歇性的睡眠。

首先来修改 WorkerThread 类：

```
public class PeerThread
{
   private int factorial;

   public ArrayList CalculateFactors(int number)
   {
      if (number < 3)
         return null;
      else
      {
```

```
        ArrayList factors = new ArrayList();
        factors.Add("1") ;
        for (int current =2; current <= number - 1; current++)
        {
           if ((int)(Math.Floor(number / current) * current) == number)
              factors.Add(current.ToString()) ;
        }
        factors.Add(number.ToString()) ;
        return factors;
     }
}
```

```
public ArrayList CalculateFactors()
{
   for(int count = 1; count<=30; count++)
   {
      Thread.Sleep(TimeSpan.FromSeconds(1));
      if (factorial > 0)
         break;
      else if (count == 30 && factorial == 0)
         return null;
   }
   ArrayList returnValue = CalculateFactors(factorial);
   return returnValue;
}

public long CalculateFactorial(int number)
{
   factorial = 0;
   if (number < 0) return -1;
   if (number == 0) return 1;
   int returnValue = 1 ;

   for (int current = 1; current <= number; current++)
      returnValue *= current;
   factorial = returnValue;
   return returnValue;
}
```

首先说明一些小的变化。线程创建了一个私有字段，用于保存阶乘积的结果。此外，

该类被重新命名为 PeerThread。CalculateFactors()方法添加了一个重载，如果传入参数，它就执行这个模型的事务端。

线程会监控 factorial 字段的状态，就像它是套接字一样。它检查其值是否为 0，如果不是，则调用 CalculateFactors()方法，把 factorial 值作为其参数，并返回所生成的 ArrayList。另一个改动是在 CalculateFactorial()方法的起始部分重新设置了 factorial 字段，以保证总是有工作可做。在方法结束处，设置 factorial 字段等于阶乘积。

还需要修改 frmCalculate 类。请看下面的修改部分：

```
...
  private PeerThread threadMethods;
  ...
  ...
  public frmCalculate()
  {
    //
    // Required for Windows Form Designer support
    //
    InitializeComponent();

    //Add any initialization after the InitializeComponent() call
    threadMethods = new PeerThread() ;
  }

  ...
private void NewFactorsThread()
{
  ArrayList val = threadMethods.CalculateFactors();
  calculateFactorial.Join() ;

  StringBuilder sb = new StringBuilder();
  for (int count = 0; count <= val.Count - 1; count++)
  {
    sb.Append((String)val[count]) ;
    if (count < val.Count - 1)
      sb.Append(", ") ;
  }
  //Create and invoke the delegate with the new value
  UpdateValue updVal = new UpdateValue(DisplayValue);
  String[ ] Arugs = {sb.ToString()};
  this.Invoke(updVal,Arugs) ;
```

```
    }

    ...

private void cmdFactorial_Click(object sender，System.EventArgs e)
{
    Thread calculateFactors = new
        Thread(new ThreadStart(NewFactorsThread)) ;
    Thread calculateFactorial = new
        Thread(new ThreadStart(FactorialThread)) ;
    calculateFactors.Start() ;
    calculateFactorial.Start() ;
}
```

除了将新的 threadMethods 字段定义为新的 PeerThread 类之外，还有两个重要的变化。我们定义了一个新的方法 NewFactorsThread()，它将调用 PeerThread 类的 CalculateFactors()方法(不带参数)。方法的其余部分和原来相同。

在 cmdFactorial_Click()方法中，不是仅在线程中激活 FactorialThread()方法，也激活了 FactorsThread()，并依次执行线程，这样 calculateFactors 就必须等待 calculateFactorial。这可以与网络套接字联系起来，第 7 章将介绍一个监控这种套接字的例子。

这种模式的常见问题是死锁和阻塞。如果让一个线程持续不断地监听套接字，并从中获得参数，这个套接字就需要从其他线程中锁定。这意味着，这是一个用于管理者类的非常好的模式。可以使用一个线程来监控套接字，然后激活新的线程来处理其内容。但是不要用多个线程监控一个套接字。此外，如果线程连续不断地检查套接字或者其他资源，就会占用过多的处理器周期。如上面的示例所示，可以使用 Thread.Sleep()来减少处理器周期的耗费。

这个示例和管道线程模式在概念上非常相似，但使用的是对等线程模式。如果单击 Calculate Factorials 按钮，除非机器的速度非常快，否则就应该有时间单击 Calculate Factorials 按钮，它可以计算并显示 200 的阶乘数，之后用 8!的阶乘数或 40320 覆盖文本框。

(3) 管道线程模式

管道线程模式是基于一系列任务的，每个任务都依赖于前面的任务。管道线程模式如图 4-6 所示。

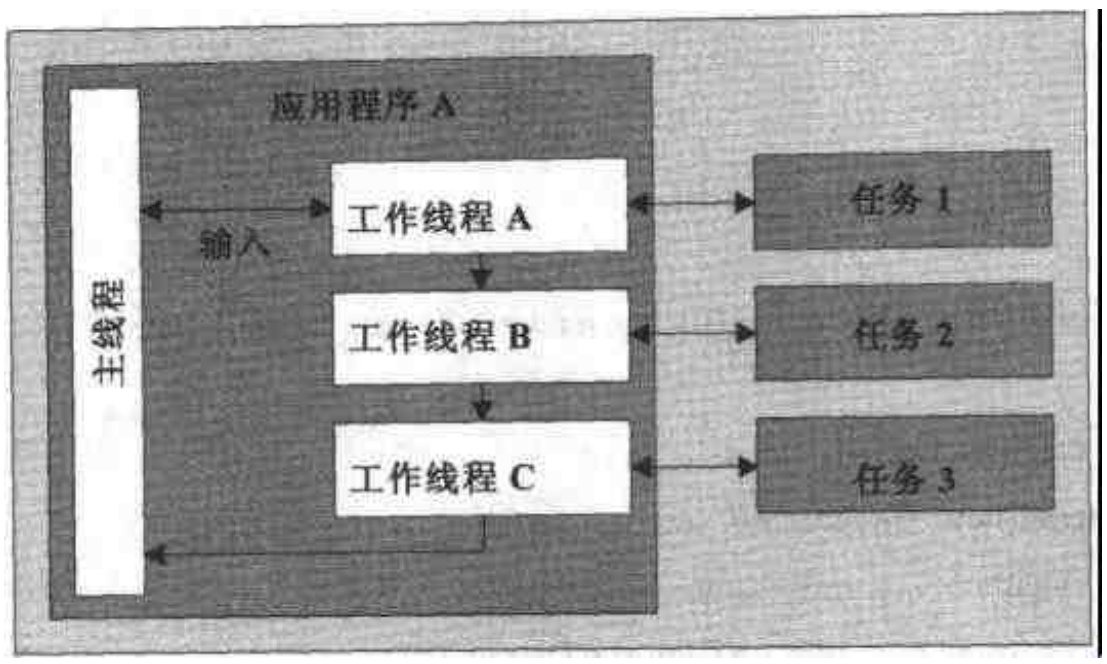


图　4-6

在图 4-6 中，主线程创建了一系列线程，在前面的线程执行完毕之前，其他线程都将等待。如果任务有若干个阶段，每个阶段都依赖于前一阶段，就可以使用这种线程关系。例如，有一个任务是处理输入数据，处理过程包含如下所示的子任务：

- 筛选所有的非法字符，例如<，>和！等。
- 检查数据格式是否符合要求
- 用货币符号和小数点格式化所有的数字
- 处理输入

在这种情况下，下一个任务只能在前面任务结束后才能启动。在前面的模式中就是这样，factorial 字段只能在线程结束后设置。然而，实现管道线程模式的正确方法是测试前面的线程是否已结束。

下面对对等线程实例进行一些修改，使其成为管道线程模式。首先列出 frmCalculate 的代码：

```
...
private PipelineThread threadMethods;
private Thread calculateFactorial;
...
public frmCalculate()
{
   ...
   threadMethods = new PipelineThread() ;
   ...
}
void NewFactorsThread()
{
   ArrayList val = threadMethods.CalculateFactors();
   calculateFactorial.Join();
   ...
```

可以看出，此处的变化是很小的。惟一能引起我们兴趣的是重新界定 calculateFactorial 范围的代码行和 calculateFactors.Join()行。这说明在 calculateFactorial 执行完毕后，calculateFactors 线程才开始执行。Join()调用必须放在要暂停执行的线程之内，并在等待它的线程上调用它。这说明，线程变量必须重新界定为至少是类范围，以便能从其他线程中访问到该线程。

```
public class PipelineThread
  ...
  public ArrayList CalculateFactors()
  {
    ArrayList returnValue = CalculateFactors(factorial);
    return returnValue ;
  }
```

此处的变化也很小。上述方法不再检查是否设置了 factorial，所以它在执行时可以假定阶乘积已经计算出来了。在执行进一步的计算之前要等待填充 DataSet 时，就可以使用这个例子。这样，填充 DataSet 的线程一结束，就创建另一个线程。当然，在实际的应用程序中，必须进行错误检查，因为一个本应正常完成的线程由于出现了一个错误，或者由于从其他线程中发出了 Abort()指令，而可能会意外终止。

此处需要关注的陷阱和其他线程可能的陷阱完全相同。第一个线程或许放在了一个无限循环中，在这种情况下，第二个线程就永远不会被执行。确保在第一个线程中不会发生这种事情，才能保证第二个线程的执行和完成。此外，还要关注线程的意外终止，这可能是由于发生了错误或其他原因，这些内容在前面章节中已经讲述过了。

以上内容就是适用于线程的 3 种线程模式。将应用程序设计为其中的一种模式，用户就会逐渐熟悉需要使用的代码结构了。

## 4.4　本章小结

本章讲述了可以应用到应用程序中的各种线程模式，使用这些模式的原因，以及如何实现这些模式等内容。现在您应该掌握了以下内容：

- 不同类型的基本线程模式，例如 STA 和 MTA 单元等
- 如何指定线程模式
- 可以使用的不同线程模式，以及它们之间的关系

本章的内容将有助于设计线程应用程序，任何应用程序都将适用上述 3 种模式中的一种或多种，因此本章中的代码示例将有助于构建需要的代码。.NET Framework 中的线程功能非常强大，但同时也比过去更容易出错。了解应用程序适用的线程模式，就可以避免这些潜在的错误。

# 第5章　线程应用程序的伸缩

编写多线程应用程序的目的是尽可能充分利用计算机的处理器。我们需要把每个独立的任务分配给不同的线程，让处理器确保总是处理其中一个线程上的命令。对于小型系统，这是可行的。但系统越大，线程数就越多，操作系统分配锁所需的时间也就越多，线程之间的竞争也就越多，而处理程序指令的时间就越少。为了使应用程序可以伸缩，必须对线程进行更多的控制。

对于某些线程生存期比较短暂的情况，使用线程池执行任务比每次完成一个任务时都创建一个全新的线程，随后又删掉的做法更有效率。在这种情况下的任务可能是一个方法，也可能是若干个方法。在应用程序实际使用线程之前，预先指定线程集合或线程池的过程称为线程池管理。

本章将详细讲解线程池问题，其中将涉及以下内容：

- 什么是线程池管理
- 需要线程池的原因
- 线程池的概念
- CLR 在线程池中的作用
- 线程池中的缺陷及其解决方法
- 线程池的大小
- 探讨.NET 的 ThreadPool 类
- C#中的线程池编程

.NET Framework 的公共语言运行库(Common Language Runtime，CLR)在线程池管理过程中将扮演重要的角色。

## 5.1　什么是线程池管理

线程池管理是指在多线程应用程序的初始化过程中创建线程的集合，当需要线程时，为新任务重用这些线程，而不是创建新线程的过程。这个过程中的线程数量通常是固定的，这取决于可用的内存量和应用程序的需要。然而，也可以增加可用的线程数量。池中的每个线程都分派了一个任务，当任务完成时，线程就返回线程池中等待下一次分派。

## 5.1.1　需要线程池的原因

线程池在多线程应用程序中是必需的，其原因如下：

- 线程池缩短了应用程序的响应时间，因为线程池中的线程是现成的，一直处于等待任务分派的状态中，系统无需从头创建线程。
- 在线程池方式下，CLR 不必为每个生存期短暂的任务创建一个全新线程，并在其结束时回收其资源。
- 线程池根据系统当前运行的进程，优化线程时间片。
- 线程池允许启动多个线程，而不必为每个线程设置属性。
- 线程池允许将状态信息作为一个对象传递给当前执行的任务的过程参数。
- 线程池可以将处理客户请求的线程数量固定为某一个最大值。

## 5.1.2　线程池的概念

影响多线程应用程序的响应性的一个主要原因是为每个任务创建线程的时间开销。

例如，Web 服务器是一个多线程应用程序，它可以同时响应多个客户请求。假设有 10 个客户同时访问 Web 服务器：

- 如果服务器采用每个客户一个线程的策略，就应创建 10 个新线程来为这些客户服务，此时服务器要先创建这些线程，然后在整个生存期内管理它们。在某个运行时刻，系统可能会耗尽所有的资源。
- 另一种方法是，如果服务器使用线程池来响应客户的请求，那么每当客户发出请求时，系统就无需为创建线程而耗费时间。服务器还可以管理所创建的线程数，如果太忙，还可以拒绝客户的请求。这就是线程池管理的概念。

.NET CLR 为响应客户请求维护一个线程池。如果应用程序请求线程池中一个新的线程时，CLR 就试图从池中取出一个。如果线程池是空的，就创建一个新的线程以供使用。当使用线程的代码终止时，.NET 就会回收该线程，并返回到线程池中。线程池中的线程数由可用的内存量限制。

影响多线程应用程序的线程设计的因素概括如下：

- 应用程序的响应性
- 线程管理资源的分配
- 资源共享
- 线程同步

应用程序的响应性和资源共享将在本章的线程池管理中讲解。其余的因素在本书前面的章节中已经讲述过了。

## 5.2　CLR 和线程

CLR 是专门用于创建托管代码环境，为在.NET 平台上运行的应用程序提供各种服务的，例如编译、垃圾回收、内存管理，还有线程池。

在定义拥有应用程序所用线程的过程方面，Win32 和.NET Framework 有着显著的差别。在传统的多线程 Win32 应用程序中，每个进程都是由线程集合组成的。每个线程又由线程本地存储器(Thread Local Storage，TLS)和调用堆栈组成，用于在单 CPU 系统中提供时间片。单处理器系统根据线程的优先级为每个线程分配时间片。当某个线程的时间片消耗完时，它就会处于挂起状态，其他线程就将开始执行其任务。在.NET Framework 中，每个 Win32 进程在逻辑上都可细分成多个应用程序域，应用程序域包含了线程以及 TLS 和调用堆栈。注意，应用程序域之间的通信是通过.NET Framework 中的远程处理技术来进行的。

在介绍了线程池和.NET 进程的概念之后，下面开始探讨 CLR 是如何为.NET 应用程序提供线程池管理功能的。

### 5.2.1　CLR 在线程池管理中的作用

CLR 是.NET Framework 的灵魂和核心，它为托管应用程序提供了多个服务，线程池管理就是其中之一。对于线程池中的每个任务(任务项)，CLR 都会从线程池中指派一个线程(工作线程)，然后在任务结束时将线程释放回池中。

线程池总是通过 CLR 使用多线程单元模式，借助抢先式多任务方式，利用高性能的队列和调度程序来实现的。在这个过程中，CPU 时间被分成多个时间片。在每个时间片中，都在执行一个线程，而其他线程则处于等待状态。这个时间片用完之后，系统就根据剩余线程的最高优先级决定由哪个线程使用 CPU。客户请求排在任务队列中，队列中的每个任务都会分配给线程池中第一个可用的线程。

一旦线程完成了分配给它的任务，就返回线程池，等待 CLR 的下一次分配。线程池的大小可以是固定不变的，也可以是动态变化的。在前一种情况下，线程的数量在线程池的生存期内不发生变化。通常情况下，如果确切知道应用程序可用资源的数量，就可以使用这种类型的线程池，这样固定数目的线程就可以在线程池初始化过程中创建。例如，为企业内部网开发解决方案，或者应用程序可以严格定义目标平台的系统需求。大小动态可变的线程池适用于不知道可用资源数量的情况，例如在 Web 服务器中，我们不知道将要同时处理多少个客户请求。

## 5.2.2　线程池管理中的问题

毫无疑问，线程池在构建多线程应用程序方面有许多优点，但是在某些情况下要避免使用线程池。下面逐一列出其缺点和应该避免使用线程池的情况：

- CLR 将线程池中的线程分配给任务，并在任务完成时将线程释放回池中。如果任务已添加到队列中，就不能直接取消任务。
- 线程池管理对于生存期较短的任务非常有效，例如 Web 服务器响应客户对某个文件的请求。线程池不适用于又大又长的任务。
- 线程池管理是一种以成本效率比很高的方式使用线程的技术，其成本效率比根据数量和启动开销来确定。决定如何使用池中的线程时要十分小心。线程池的大小应该固定不变。
- 线程池中的所有线程都处于多线程单元中。如果想把线程放到单线程单元中，线程池就没有用了。
- 如果需要标识线程并执行各种操作，例如启动线程、挂起和中止等，那么线程池不能完成这样的工作。
- 同样，不能对使用线程池的任务设置优先级。
- 对任意给定的应用程序域，只能有一个线程池与其相关联。
- 如果分配给线程池中一个线程的任务被锁定，这个线程将不会再释放回池中。这种情况可以使用有效的编程技巧来避免。

## 5.2.3　线程池的大小

为了在应用程序中使用线程池，在.NET Framework 在 System.Threading 命名空间中提供了 ThreadPool 类。在线程池中可以排队等待的任务数量取决于机器的内存量。同样，在进程中可以激活的线程数量取决于机器中的 CPU 个数。这是因为每个处理器一次只能执行一个线程。默认情况下，线程池的每个线程都将使用默认的任务，在多线程单元中以默认的优先级运行。此处使用的“默认”一词似乎不太明确，但不会有任何问题。每个系统都设置了不同的默认优先级。在任意时刻，如果某个线程处于空闲状态，线程池就会引导工作线程，使所有的处理器保持繁忙。如果线程池中的所有线程都处于繁忙状态，并且队列中有未处理的任务，线程池将产生新的线程来完成待处理的工作。但是，产生的线程数量不能超过指定的最大值。默认情况下，每个进程可以产生 25 个线程池线程。然而，这个数量可以通过编辑 mscoree.h 文件中的 CorSetMaxThreads 成员加以改变。如果有额外的线程请求，这个请求将加入到队列中，直到某个线程完成了分配给它的任

务，返回到线程池中为止。.NET Framework 对异步调用、建立套接字连接和注册过的等待操作等使用线程池。

## 5.3 ThreadPool 类

本节将探讨 ThreadPool 类的相关知识，讨论如何在.NET 应用程序中使用 ThreadPool 类创建线程池。ThreadPool 类提供的线程池可以用来解决以下问题：

- 处理任务项
- 处理异步 I/O 调用
- 处理计时器
- 代表其他线程等待

表 5-1 列出了 ThreadPool 类中的方法及其功能。

表 5-1

| 方法名称 | 功　能 |
| --- | --- |
| BindHandle | 该方法将操作系统句柄绑定到线程池中 |
| GetAvailableThreads | 该方法指出可以添加到任务项队列中的任务数 |
| GetMaxThreads | 该方法指出线程池可以同时排队的请求数量 |
| QueueUserWorkItem | 该方法将一个任务项排列到线程池中 |
| RegisterWaitForSingleObject | 该方法注册一个委托，它等待一个 WaitHandle |
| UnsafeQueueUserWorkItem | 这是 QueueUserWorkItem()方法的非安全版本 |
| UnsafeRegisterWaitForSingleObject | 这是 RegisterWaitForSingleObject()方法的非安全版本 |

在上面这些方法中，QueueUserWorkItem()和 RegisterWaitForSingleObject()在线程池管理中是最重要的两个。下面深入研究这两个方法。先来看一下它们的语法以及在 C#中的调用示例。

BindHandle() 方法将操作系统句柄绑定到线程池中。这是调用 BindIoCompletionCallback 的一种方式。BCL 中的几个类，例如 Socket 和 FileStream，在内部使用它把它们的句柄绑定到 CLR 创建的线程池的 I/O 完成端口。客户应用程序常常不需要直接调用这个方法，在调用 BeginRead 或 BeginReceive 等方法时，会间接访问这个功能。

```
public static bool BindHandle(IntPtr osHandle);
```

osHandle 表示保存操作系统句柄的 IntPtr 类型。其返回值是布尔型，如果绑定成功

则为 True，如果调用程序没有必要的许可，该方法就抛出 SecurityException 异常。

GetAvailableThreads()方法指出在达到最大限制值之前，能够添加到线程池中的请求数：

```
public static void GetAvailableThreads(out int workerThreads,
                                        out int completionPortthreads);
```

WorkerThreads 表示线程池中工作线程的数量，而 completionPortThreads 表示异步的 I/O 线程数量。

GetMaxThreads()方法返回线程池可以处理的并发请求的最大数量。超过这个限制的所有请求都将排到队列中，直到线程池中的某些线程空闲为止：

```
public static void GetMaxThreads(out int workerThreads,
                                  out int completionPortThreads);
```

WorkerThreads 指线程池中工作线程的数量，而 CompletionPortThreads 指异步 I/O 线程的数量。

QueueUserWorkItem()是一个重载方法，它将一个任务项排列到线程池中。这个方法的调用形式有两种。第一种情况是该方法将指定的工作项排列到线程池中，并调用与其相关联的委托。这种情况使用如下语法：

```
public static bool QueueUserWorkItem(WaitCallback callBack);
```

此处，callBack 表示线程池中的线程获得任务项时要调用的委托。返回值 true 表明方法调用成功了，False 表示调用失败。

第二种情况是该方法将特定的任务项排队到线程池中，调用特定的委托，指定线程池中的任务项执行时要传递给委托的对象。其语法如下：

```
public static bool QueueUserWorkItem(WaitCallback callback,
                                     object state);
```

callBack 表示线程池中的线程执行任务时要调用的委托，state 表示包含状态的对象，该对象将在线程执行任务时传递给委托。返回值 true 表明方法调用成功了，False 表示调用失败。

RegisterWaitForSingleObject()也是一个重载方法。它注册了一个等待 WaitHandle 的委托。这个类封装了等待对共享资源进行独占访问的所有操作系统对象。

这个方法有下列 4 种调用形式。第一种，方法注册一个委托，并等待 WaitHandle 一定的时间，该时间是由一个 32 位的有符号整数表示的超时值(用毫秒数表示)。该方法的这种重载版本采用如下所示的 C#语法形式：

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    int millisecondsTimeOutInterval,
    bool executeOnlyOnce);
```

在上述语法中，waitObject 表示 WaitHandle，callBack 表示要调用的 WaitOrTimerCallback 委托。state 参数表示要传递给委托的 Object。millisecondsTimeOutInterval 参数表示超时值(用毫秒数表示)；如果其值为 0，则函数测试对象状态并立即返回。如果它的值是 - 1，函数将无限期等待。executeOnlyOnce 参数表示在调用委托之后，线程是否要等待 waitObject 参数。RegisteredWaitHandle 封装了本地句柄。

如果 millisecondsTimeOutInterval 的值小于 - 1，这个方法将抛出 Argument OutOf RangeException 异常。

第二种情况，这个方法所做的工作与第一种情况相同，只是它等待 WaitHandle 的时间由一个 32 位的无符号整数表示的超时值(用毫秒数表示)确定。这种重载形式的语法为：

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    long millisecondsTimeOutInterval,
    bool executeOnlyOnce);
```

第三种情况，方法等待 WaitHandle 的时间由 TimeSpan 值给定的超时值确定。这种重载形式的语法如下：

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    TimeSpan timeout,
    bool executeOnlyOnce) ;
```

第四种情况，超时值由一个无符号整数指定(毫秒)，无符号整数不是通用的类型系统，所以这个方法不符合 CLS 规则：

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(
```

```
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    uint millisecondsTimeOut Interval,
    bool executeOnlyOnce) ;
```

UnsafeQueucUserWorkItem()方法是 QueueUserWorkItem()方法的非安全版本，因为它不会将调用堆栈扩展到工作线程中。这就意味着代码可以失去调用堆栈，获得不应有的安全权限。其语法如下：

```
public static bool UnsafeQueueUserWorkItem(WaitCallback callBack,
                                           object state) ;
```

UnsafeRegisterWaitForSingleObject()方法是 RegisterWaitForSingleObject()方法的非安全版本，它采用如下 4 种形式的语法：

```
public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    int millisecondsTimeOutInterval,
    bool executeOnlyOnce) ;

public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    long millisecondsTimeOutInterval,
    bool executeOnlyOnce) ;

public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
    TimeSpan timeout,
    bool executeOnlyOnce) ;

public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(
    WaitHandle waitObject,
    WaitOrTimerCallback callBack,
    object state,
```

```
uint millisecondsTimeOutInterval,
bool executeOnlyOnce);
```

在自己不信任的代码中给任务项排序时，不应使用这些不安全的方法，在一般情况下都不要使用。

## 5.4　C#中的线程池编程

本章前面的几节讲述了在.NET Framework 中使用线程池的理论方面。现在该讨论如何在.NET 应用程序(以 C#程序为例)中创建和使用线程池等编程方面的问题了。正如前一节所述，System.Threading 命名空间中的类 ThreadPool 可以用来在.NET 应用程序中创建线程池。

在开始编写代码之前，先介绍 ThreadPool 类的两个重要规则，它们是：

- 每个应用程序域只能有一个 ThreadPool 对象。
- 第一次调用 ThreadPool.QueueUserWorkItem()方法，或者是通过计时器或已注册等待的操作调用回调方法(在内部使用应用程序域线程池)时，就会创建 ThreadPool 对象。

首先介绍的例子说明了 ThreadPool 类比启动各个线程更好的原因。在下一个例子(ThreadDemo.cs)中，将使用独立线程启动两个较长的任务。本例(ThreadPoolDemo.cs)启动两个相同的任务，但使用的是线程池：

```
using System;
using System.Threading;


class ThreadDemo
{
  public void LongTask1()
  {
    for (int i = 0; i <= 999; i++)
    {
      Console.WriteLine("Long Task 1 is being executed");
    }
  }

  public void LongTask2()
```

```
{
   for(int i = 0; i <= 999; i++)
   {
      Console.WriteLine("Long Task 2 is being executed") ;
   }
}

static void Main()
{
   ThreadDemo td = new ThreadDemo() ;
   for(int i = 0; i < 50; i++)
   {
      Thread t1 = new Thread(new ThreadStart(td.LongTask1));
      t1.Start() ;
      Thread t2 = new Thread(new ThreadStart(td.LongTask2));
      t2.Start() ;
   }

   Console.Read () ;
  }
}
```

在上面的例子中，使用两个独立的线程 t1 和 t2 启动了两个任务 longtask1 和 longtask2。注意线程在一个循环中重复调用，强调操作系统的处理功能，以便更好地了解使用线程池的优点。下面的 ThreadPoolDemo 类说明了 ThreadPool 类的用法：

```
using System;
using System. Threading;

class ThreadPoolDemo
{
   public void LongTask1(object obj)
   {
      for(int j = 0; i <= 999; i++)
      {
         Console.WriteLine("Long Task 1 is being executed");
      }
   }
```

```
public void LongTask2(object obj)
{
   for(int i = 0; i <= 999; i++)
   {
      Console.WriteLine("Long Task 2 is being executed");
   }
}

static void Main()
{
   ThreadPoolDemo tpd = new ThreadPoolDemo() ;
   for(int i = 0; i < 50; i++)
   {
      ThreadPool.QueueUserWorkItem(new WaitCallback(tpd.LongTask1)) ;
      ThreadPool.QueueUserWorkItem(new WaitCallback(tpd.LongTask2)) ;
   }

   Console.Read() ;
  }
}
```

下面仔细分析这个例子。它包括两个任务 longtask1 和 longtask2，这两个任务相当简单，只是在循环中向控制台输出一条消息。ThreadPool 类可以用来启动这两个任务，而不用为每个任务设置线程属性，其方法是向 WaitCallback()方法传递过程的委托，这一点如下代码所示：

```
ThreadPool.QueueUserWorkItem(new WaitCallback(tpd.LongTask1)) ;
ThreadPool.QueueUserWorkItem(new WaitCallback(tpd.LongTask2)) ;
```

注意在 ThreadPool 类中，QueueUserWorkItem 是一个静态方法，因此可以由 ThreadPool 类直接调用。示例中还有一条 Console.Read()语句，它用于保存控制台的输入直至用户按下 Enter 键(或者其他任意键)。

按次序运行 ThreadDemo 和 ThreadPoolDemo 应用程序，使用 WindowsTask Manager 比较线程的使用情况。每个操作系统上的数据都会因为该系统的性能而有所不同，但结果是相同的。

ThreadDemo 应用程序的线程使用情况如图 5-1 所示。

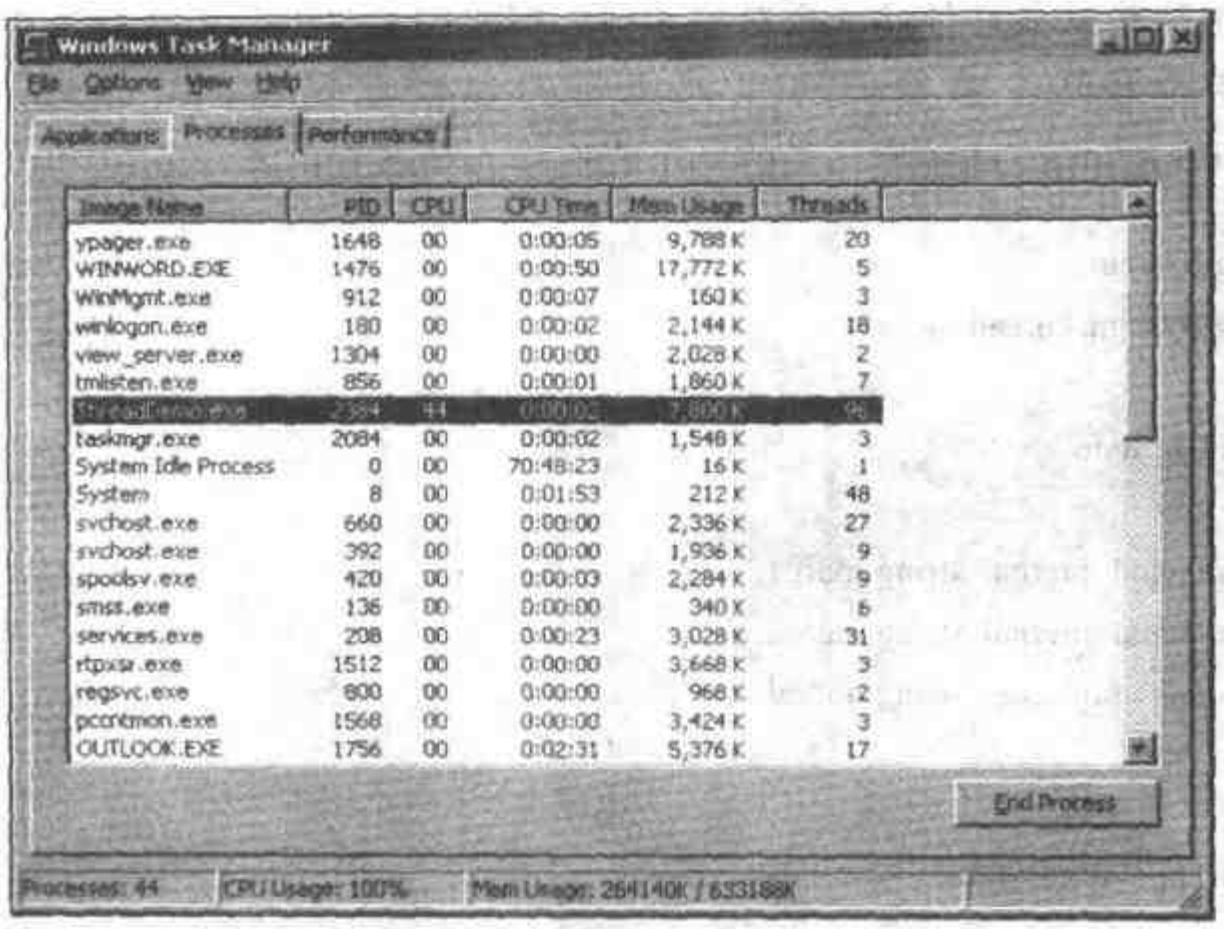


图　5-1

ThreadPoolDemo 应用程序的线程使用情况如图 5-2 所示。

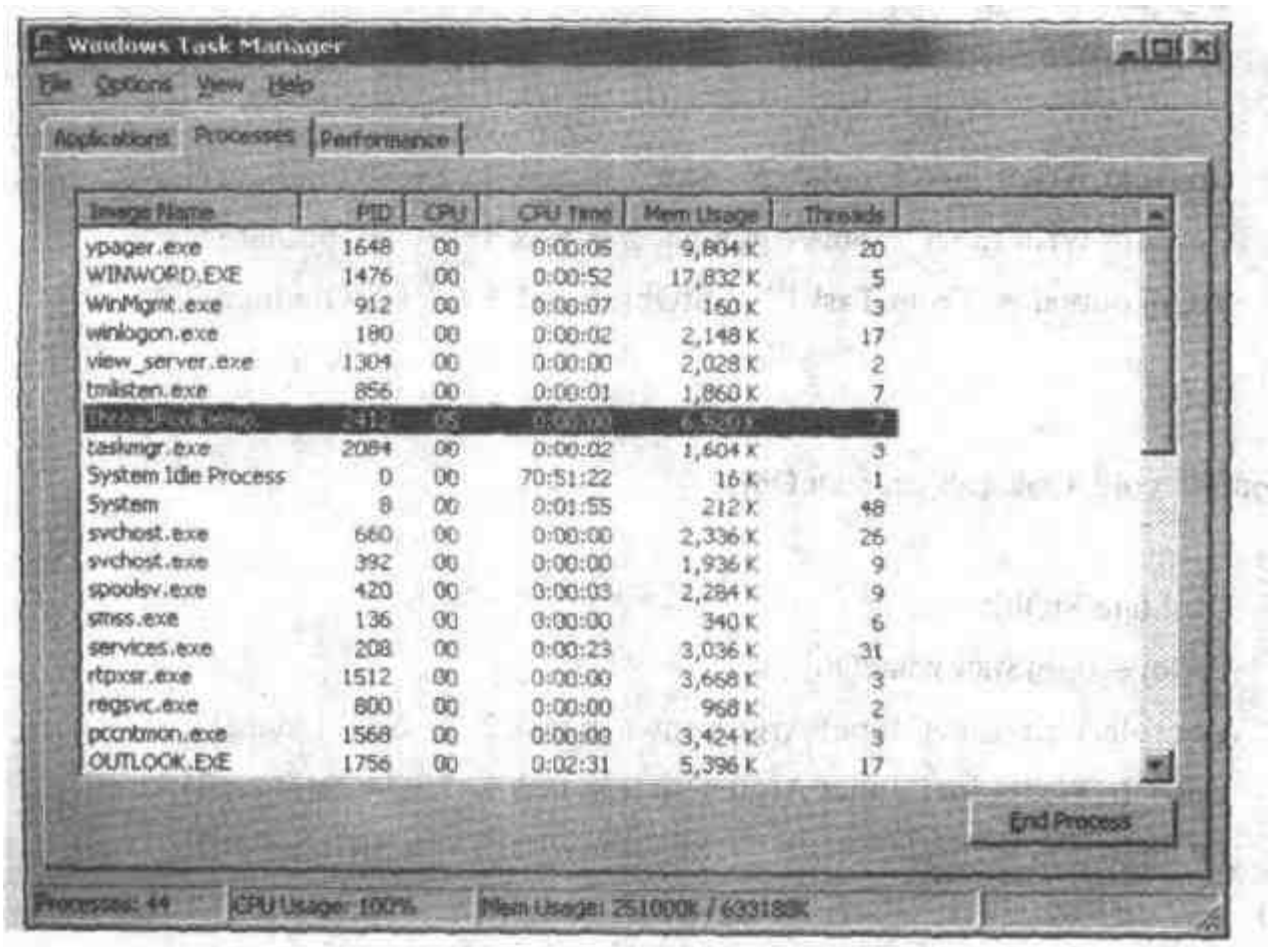


图　5-2

比较这两个屏幕图，显然使用 ThreadPool 不仅有助于减少应用程序使用的线程数，还可以减少 CPU 时间和应用程序使用的内存。

下一个示例(ThreadPoolState.cs)演示如何向线程池中的线程传递数据并从中得到返

回值。线程池架构只允许传递一个对象参数，但我们常常希望给线程池中的线程所执行的方法传递好几个参数。可以把所有的参数都包装到一个类中，然后向QueueUserWorkItem()方法传递该类的一个实例作为其参数：

```
using System;
using System.Threading;

class ObjState
{
   protected internal String inarg1;
   protected internal String inarg2;
   protected internal String outval
}

class ThreadPoolState
{
   public void Task1(object stateObj)
   {
      ObjState StObj;
      StObj = (ObjState)stateObj;

      Console.WriteLine("Input Argument 1 in task 1: " + StObj.inarg1) ;
      Console.WriteLine("Input Argument 2 in task 1: " + StObj.inarg2) ;
      StObj.outval = "From Task1 " + StObj.inarg1 + " " + StObj.inarg2
   }

   public void Tasks(object stateObj)
   {
      ObjState StObj;
      StObj = (ObjState)stateObj ;
      Console.WriteLine("Input Argument 1 in task 2: "+ StObj.inarg1);
      Console.WriteLine("Input Argument 2 in task 2: " + StObj.inarg2);
      StObj.outval = "From Task2 " + StObj.inarg1 + " "+ StObj.inarg2;
}

static void Main()
{
   ObjState StObj1 = new ObjState();
   ObjState StObj2 = new ObjState();
   StObj1.inarg1 = "String Param1 of task 1";
```

```
        StObj1.inarg2 = "String Param2 of task 1";
        StObj2.inarg1 = "String Param1 of task 2";
        StObj2.inarg2 = "String Param2 of task 2";

        ThreadPoolState tps = new ThreadPoolState(),
        // Queue a task
        ThreadPool.QueueUserWorkItem(new WaitCallback(tps.Task1), StObj1) ;
        // Queue another task
        ThreadPool.QueueUserWorkItem(new WaitCallback(tps.Task2), StObj2) ;
        Console.Read () ;
    }
}
```

ThreadPoolState 的输出结果如下所示：

```
Input Argument 1 in task 1: String Param1 of task 1
Input Argument 2 in task 1: String Param2 of task 1
Input Argument 1 in task 2: String Param1 of task 2
Input Argument 2 in task 2: String Param2 of task 2
```

下面剖析上面这个示例。这个示例与前面的示例很相似。但是它传递的是一个对象，我们使用 ObjState 对象向在线程池中排队的任务传递输入和输出参数。

ObjState 对象包含两个输入参数和一个输出参数，所有参数都是字符串型，如下所示：

```
internal class ObjState
{
  protected internal String inarg1;
  protected internal String inarg2 ;
  protected internal String outval;
}
```

下面定义两个方法 task1 和 task2，向每个过程传递一个 ObjState 类的实例作为其参数。方法 task1 和 task2 将所传递的 ObjState 对象中的两个输入参数 inarg1 和 inarg2 的值连接起来，并将结果保存在 outval 类变量中。这两个方法的代码如下：

```
public void Task1(object stateObj)
{
  ObjState StObj;
  StObj = (ObjState)stateObj;

  Console.WriteLine("Input Argument 1 in task 1: " + StObj.inarg1);
```

```
    Console.WriteLine("Input Argument 2 in task 1: " + StObj.inarg2);
    StObj.outval = "From Task1 " + StObj.inarg1 + " " + StObj.inarg2;
}

public void Task2(object stateObj)
{
    ObjState StObj ;
    StObj = (ObjState)stateObj;
    Console.WriteLine("Input Argument 1 in task 2: " + StObj.inarg1);
    Console.WriteLine("Input Argument 2 in task 2: " + StObj.inarg2);
    StObj.outval = "From Task2 " + StObj.inarg1 + " " + StObj.inarg2;
}
```

在 Main()方法中，使用 ThreadPool 类的 QueueUserWorkItem()方法将这两个任务排列在线程池中，其代码如下：

```
static void Main()
{
    ObjState StObj1 = new ObjState();
    ObjState StObj 2 = new ObjState();
    StObj1.inarg1 == "String Param1 of task 1";
    StObj1.inarg2 = "String Param2 of task 1";
    StObj2.inarg1 = "String Param1 of task 2";
    StObj2.inarg2 = "String Param2 of task 2";

    ThreadPoolState tps = new ThreadPoolState() ;

    // Queue a task
    ThreadPool.QueucUserWorkItem(new WaitCallback(tps.Task1), StObj1) ;
    // Queue another task
    ThreadPool.QueueUserWorkItem(new WaitCallback(tps.Task2), StObj2) ;
    Console.Read () ;
}
```

使用 RegisterWaitForSingleObject()方法(给它传递 WaitHandle 参数)也可以将拥有等待操作的任务项排列到线程池中。这个 waitHandle 向包装在 WaitOrTimerCallback 委托中的方法发出信号。在本例中，线程池创建了一个后台线程来调用回调方法。下面的示例(RegWait.cs)就演示了这种使用方法：

```
using System;
using System.Thrcading;
```

```
public class RegWait
{
   private static int i = 0;

   public static void Main()
   {
      AutoResetEvent arev = new AutoResetEvent(false);
      ThreadPool.RegisterWaitForSingleObject
            (arev, new WaitOrTimerCallback(workitem), null, 2000, false);
      arev.Set();
      Console.Read () ;
   }
   public static void workitem(object O ,bool signaled)
   {
      i += 1;
      Console.WriteLine("Thread Pool Work Item Invoked; " +
                        i.ToString() ) ;
   }
}
```

上述示例的输出结果如下：

```
Thread Pool Work Item Invoked: 1
Thread Pool Work Item Invoked: 2
Thread Pool Work Item Invoked: 3
Thread Pool Work Item Invoked: 4
```

程序每两秒钟输出一个新行，i 值递增 1，直到用户按下 Enter 键调用 Console.Read() 语句为止。

首先，创建一个 AutoResetEvent 对象 arev，其初始状态是没有发出执行所排队的任务项的信号：

```
AutoResetEvent arev = new AutoResetEvent(false);
```

调用 RegisterWaitForSingleObject()方法，其状态参数为 null，将超时值设置为 2000 毫秒，executeOnceOnly 参数为 false。RegisterWaitForSingleObject()方法注册了一个委托，并以特定的时间间隔通知任务项。在示例中，时间间隔被设置为 2 秒，如下面的代码所示：

```
ThreadPool.RegisterWaitForSingleObject
```

```
(arev, new WaitOrTimerCallback(workitem), null, 2000, false) ;
```

为了引发事件，需要使用 AutoResetEvent 对象的 Set()方法：

```
arev.Set()
```

这个示例结束了在 C#应用程序中使用线程池的实践练习，下一节将讨论可伸缩性，并构建一个线程池管理器应用程序。

## 多线程的 Microsoft 消息队列(MSMQ)监听程序

本节将利用 ThreadPool 在 MSMQ 中发送和接收消息。MSMQ 是一个分布式的队列，在该队列中，一个应用程序可以给另一个应用程序异步发送消息。

在一般的情形中，MSMQ 服务器会维护一个给其发送消息的队列列表。MSMQ 发送器与 MSMQ 服务器建立连接(连接某个队列)，并给该队列发送消息。MSMQ 接收器会接收由 MSMQ 发送器发送的消息。MSMQ 接收器必须监听一个特定的队列，接收来自该队列的消息。因此，MSMQ 服务器充当了 MSMQ 发送器和 MSMQ 接收器之间的经纪人。MSMQ 发送器并不知道存在 MSMQ 接收器，反之亦然。

在应用程序中，把 MSMQ 发送器开发为一个 Windows Forms 应用程序，把 MSMQ 接收器开发为一个控制台应用程序。在 MSMQ 发送器应用程序(MSMQUI.cs)中，使用 ThreadPool 把消息发送给队列：

```
int count = Convert.ToInt32(countTxt.Text);
while(count > 0)
{
  ThreadPool.QueueUserWorkItem(new WaitCallback(SendMessage),
                               count.ToString()) ;
  count--;
}
```

在上面的代码(MSMQUI.cs)中，创建了一个计数器，记录给 MSMQ 发送的消息数。把这些消息放在 ThreadPool 中，作为 state 对象，再把 SendMessage()方法的引用赋予 WaitCallback 委托。SendMessage()方法由 ThreadPool 调用，以发送消息。换言之，SendMessage()方法完成给 MSMQ 发送消息的实际工作。

```
public void SendMessage(object state)
{
  messageQueue1.Send (state.ToString()) ;
}
```

SendMessage()方法把 state 对象转换为字符串，再把它发送给 MSMQ。

MSMQUI 应用程序执行时，其屏幕图如图 5-3 所示。

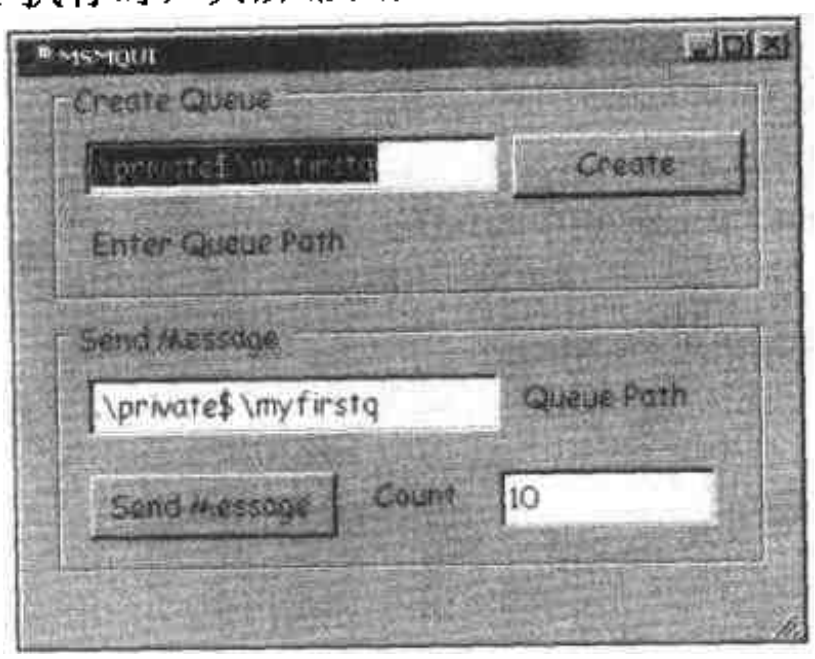


图　5-3

在 MSMQ 接收器(MSMQListener.cs)中，只要消息进入队列，就从 MSMQ 服务器中异步接收消息通知。因此，创建 MSMQ 事件处理程序，并把 MessageReceived()方法名传递给 ReceiveCompltedEventHandler 委托。

```
this._mql.ReceiveCompleted += new
System.Messaging.ReceiveCompletedEventHandler(this.MessageReceive();
```

为了开始接收来自 MSMQ 的消息，需要在消息队列对象上调用 BeginReceived()方法：

```
_mql.BeginReceive(new TimeSpan(0,0,2)) ;
```

BeginReceived()方法带有一个 TimeSpan 对象参数，表示接收器在消息到达队列之前需要等待多长时间。如果没有提供 TimeSpan，应用程序就暂停 BeginReceived()方法，直到消息到达 MSMQ 为止。

在消息到达队列时，MSMQ 事件处理程序就调用 MessageReceived()方法：

```
public void MessageReceived(object source,
    System. Messaging. ReceiveCompletedEventArgs asyncResult)
{
  MessageQueue mq = null;
  try
  {
    //Connect to the queue.
    mq = (MessageQueue)source;
```

```
System.Messaging.Message m =
      mq.EndReceive (asyncResult .AsyncResult) ;
// Process the message here.
//Console.WriteLine(m.Body.ToString()) ;
ThreadPool.QueueUserWorkItem
      (new WaitCallback(InvokeMDAO) , m.Body) ;
}
catch(MessageQueueException)
{

}
catch(Exception ex)
{
   Console.WriteLine(ex.Message) ;
}
finally
{
   this._mqI.BeginReceive(new TimeSpan(0,0,2)) ;
}
}

static void InvokeMDAO(object state)
{
   MDAO md = new MDAO() ;
   md.AccessDatabaseOrDoSomeImportantWork(state) ;
}
```

为了获取已到达的消息，需要在消息队列对象上调用 EndReceived()方法。在得到 System.Messaging.Message 对象之后，就可以对该对象进行需要的处理。在实际的应用程序中，常常是调用 System.EnterpriseServices(COM+)对象，该对象再更新数据库，或把消息发送给其他对象，进行处理。这里可以使用 ThreadPool 来控制对象的调用。把消息体放在 ThreadPool 中，把 InvokeMDAO()方法的名称传送给 WaitCallback 委托。InvokeMDAO()方法在对象上调用一个方法，在控制台上显示消息。如前所述，可以在 InvokeMDAO()方法中调用 System.EnterpriseServices(COM+)对象。这将间接控制所创建的 COM+对象数。该对象数与 ThreadPool 创建的线程数成正比。最后，需要调用 BeginReceived()方法，以再次开始异步等待。如果没有调用这个方法，消息接收器就会停止接收来自 MSMQ 的消息。

## 5.5　.NET 中的可伸缩性

如果有一个多处理器系统，线程就真的派上用场了。Windows 操作系统管理如何将线程分配给处理器，同时，触发任何进程会自动启动一个线程。.NET Framework 并没有对处理器分配提供精细的控制功能，它宁愿让操作系统控制调度，因为与 CLR 相比，它可以提供处理器负载的更多信息。然而，操作系统也对整个进程运行于哪个处理器提供了一些控制措施。但是这适用于进程中的所有线程，所以它的使用并不适合于本书。

如果只有一个线程，即主线程，那么线程中的每个任务都将运行在同一个处理器上。然而，如果创建了一个新的线程，操作系统将安排线程将在哪个处理器上执行。系统在作出这个判断时会消耗一些处理器资源，所以对于小任务而言，这种消耗通常不值得，因为执行这些任务的时间几乎和判断由哪个处理器来执行线程的时间一样长。然而，在 Windows 的后续版本中，这种分配占用的时间越来越少，同时对于除最琐碎的任务之外的事情而言，当使用线程时，创建新线程来执行任务将提高系统性能。这在对称多处理器(symmetric multi-processor，SMP)系统中很明显，因为所有的处理器都可以充分发挥其作用，以分配应用程序的负载。

下一节将讨论如何创建线程池管理器，使用它可以创建并管理线程，并确保线程池中存在最大和最小线程数量，且空线程可以被重用。

### 线程池管理器

纵观本书，已介绍了创建线程的多种不同的方法，本章还讨论了 ThreadPool 类如何为短暂生存期的线程使用操作系统自身的线程池。还可以实现一个介于两者之间的方案。先创建一个类，这个类维护具有指定线程数的线程池，为进行请求的应用程序提供线程。这样就可以在代码中方便地控制这些线程，线程的执行速度也会加快，因为可以使用以前实例化的线程对象。这个类将把前面学过的大部分知识组合在一起，您可以将其应用到自己的多线程应用程序中。下面将解释这个类的内容，最后提供一个应用程序来测试这个程序集是否如预期的那样执行。

现在先分析一下线程池管理器代码，它包含在文件 MyThreadPool.cs 中：

```
using System;
using System.Text;
using System.Threading

namespace GenThreadPool
{
```

上面的声明显示了惟一需要的外部程序集是 System.dll。GenThreadPool 命名空间包含了该项目所需的所有相关类。下面是 IThreadPool 接口，它用于 GenThreadPoolImpl 类：

```
public interface IThreadPool
{
  void AddJob(System.Threading.Thread jobToRun) ;
    Stats GetStats();
  }
}
```

它为线程池定义了两个方法 AddJob() 和 GetStats()，在后面的 GenThreadPoolImpl 类的定义中将详细说明这两个方法，它们生成了要使用的线程池：

```
public class GenThreadPoolImpl : IThreadPool
{
  private int _maxThreads ;
  private int _minThreads;
  private int _maxIdleTime;
  private static bool _debug;
  private ArrayList _pendingJobs;
  private ArrayList __availableThreads;

  public bool Debug
  {
    get
    {
      return _debug;
    }
    set
    {
      _debug = value;
    }
}

public ArrayList PendingJobs
{
  get
  {
    return this._pendingJobs;
  }
```

```
    set
    {
        this._pendingJobs = value;
    }
}

public ArrayList AvailableThreads
{
get
  {
    return this._availableThreads;
    }
}

public int MaxIdleTime
{
    get
    {
        return this ._maxIdleTime;
    }
    set
    {
        this._maxIdleTime = value;
    }
}

public int MaxThreads
{
    get
    {
        return this._maxThreads;
    }
    set
    {
        this._maxThreads = value;
    }
}

public int MinThreads
{
```

```
    get
    {
      return this._minThreads;
    }
    set
    {
      this._minThreads = value;
    }
  }
```

这个类实现了前面定义的 IThreadPool 接口，接着定义了几个 Private 字段。属性是相关 Private 成员的包装器，这样万一需要添加更多的规则时，就可以阻止用户直接修改其值。字段 m_maxThreads、m_minThreads 和 m_maxIdleTime 分别指定了池中的最多和最少线程数，以及在从池中删除一个线程之前，允许其处于空闲状态的最长时间(毫秒)。这个类有 3 个构造函数：

```
public GenThreadPoolImpl()
{
  _maxThreads = 1 ;
  _minThreads = 0 ;
  _maxIdleTime = 300;
  this. pendingJobs =
    ArrayList.Synchronized (new ArrayList() );

  this._availableThreads =
      ArrayList.Synchronized (new ArrayList()) ;

  _debug = false;
}
```

默认的构造函数仅允许池中有一个线程，并在 0.3 秒之后销毁这个线程。它也执行一些基本的傻瓜式初始化工作，创建一个数组列表以容纳等待线程的任务，以及没有分配给某个方法调用的线程。在进行测试时，只有把 m_debug 标记符设置为 True 时，才允许接纳进一步的调试信息。

```
public GenThreadPoolImpl(int maxThreads, int minThreads,
                         int maxIdleTime)
{
  _maxThreads = maxThreads;
  _minThreads = minThreads ;
```

```
    _maxIdleTime = maxIdleTime;

    this._pendingJobs =
        ArrayList.Synchronized (new ArrayList()) ;
    this._availableThreads =
        ArrayList.Synchronized (new ArrayList());

     debug = false;
    InitAvailableThreads ();
}
```

当 GenThreadPoolImpl 类用 3 个整数进行实例化时，指定了线程的最大和最小数，以及线程的空闲时间。该类还触发了 InitAvailableThreads()方法，其代码如下：

```
private void InitAvailableThreads()
{
  if(this._minThreads > 0)
    for(int i = 0; i < this._minThreads; i++)
    {
      Thread t = new
          Thread(new ThreadStart(new GenPool (this, this).run ));
        ThreadElement e = new ThreadElement (t) ;
        e.Idle = true;
        _availableThreads.Add (e) ;
     }

  Console.WriteLine("Initialized the ThreadPool. "
                    + " Number of Available threads:
                    + this._availableThreads.Count) ;
}
```

这个方法在实例化时创建了线程池必需的线程。默认的构造函数仅指定了一个线程，因为以前没有必要创建更多的线程。这段代码循环执行，创建线程池允许的最多线程，最大数量在 m_maxThreads 字段中指定。下面是带有 4 个参数的构造函数的代码：

```
public GenThreadPoolImpl(int maxThreads, int minThreads,
                         int maxIdleTime, bool debug_)
{
  _maxThreads = maxThreads ;
  _minThreads = minThreads;
  _maxIdleTime = maxIdleTime;
```

```
    this._pendingJobs =
        ArrayList.Synchronized (new ArrayList ());
    this._availableThreads =
        ArrayList.Synchronized (new ArrayList());

    _debug = debug_;
    InitAvailableThreads() ;
  }
```

这个构造函数和前面的构造函数功能相同，只是允许设置调试标记。现在讨论这个类的最后一个成员 AddJob()方法：

```
public void AddJob(Thread job)
{
  if(job == null)return;

  lock(this)
  {
```

上述方法向线程池中添加了一个任务。如果作为参数传递的任务不存在，就退出该方法。否则，就锁定 GenThreadPoolImpl 实例，以防止其他线程或进程添加或删除任务：

```
    _pendingJobs.Add (job);

    int index = findFirstIdleThread() ;

    if(_debug)
      Console.WriteLine("First Idle Thread is " + index);

  if(index == -1)
  {
    if((_maxThreads == -1)
        || ( _availableThreads.Count < _maxThreads))
    {
      if(_debug)
          Console.WriteLine("Creating a new thread");
          Thread t = new Thread(new
              ThreadStart (new GenPool(this, this).run ));
```

任务被添加到 ArrayList 中，在 ArrayList 中，将存储所有等待执行和完成的任务。

FindFirstIdleThread()函数返回包含在 m_availableThreads 中的空闲可用的线程的索引。如果函数返回-1，则表明没有空闲线程，线程池需要创建新的线程。在这个线程中将触发 GenPool 类的 Run()方法。

```
ThreadElement e = new ThreadElement (t);

e.Idle = false;
e.getMyThread(). Start ();

try
{
    availableThreads.Add (e) ;
  }
  catch(OutOfMemoryException)
  {
    Console.WriteLine("Out of Memory");
    availableThreads.Add (e) ;
    Console.WriteLine("Added Job again");
  }

  return;
}

if(_debug)
    Console.WriteLine("No Threads Available .."
                      + this.GetStats().ToString() ) ;
```

ThreadElement 类是另一个帮助类，该类随后再定义。它向标准线程添加一些额外的属性，使线程池有效地管理线程。线程的Start()方法在添加到 m_availableThreads 集合中之前就被激发了：

```
else
{
  try
  {
    if(_debug)
        Console.WriteLine("Using an existing thread...");

    ((ThreadElement)availableThreads[index]).Idle = false;
```

在上面的论述中，我们首先详细地描述了认定线程为空闲的条件，这样才能为其分

配新的任务。首先，将线程显式转换为一个 ThreadElement，并改变它的空闲标记：

```
lock(((ThreadElement)_availableThreads[index]).getMyThread())
{
  Monitor.Pulse((
      (ThreadElement)_availableThreads[index]).getMyThread ());

}
```

此处锁定了线程，使它免受其他进程的影响。接着提醒所有的等待线程，它现在可以用了，因此发出 Monitor.Pulse()指令，然后释放这个锁：

```
  }
catch(Exception ex)
{
  Console.WriteLine ("Error while reusing thread " + ex.Message ) ;

  if(_debug)
  {
    Console.WriteLine("Value of index is " + index );
    Console.WriteLine ("Size of available threads is " +
                       this._availableThreads.Count) ;
    Console.WriteLine ("Available Threads is "
                       + this._availableThreads .IsSynchronized );
    }
  }
}//end of else

}//lock
}//end of method
```

最后，捕获异常，并向命令行输出结果，如果已设置 Me.Debug 标记，就提供更多有用的调试信息。至此就结束了 AddJob()方法的实现过程。现在看一下 GetStats()方法的实现过程：

```
public Stats GetStats()
{
  Stats stats = new Stats ();

  stats.maxThreads = _maxThreads;
  stats.minThreads = _minThreads;
```

```
    stats.maxIdleTime = _maxIdleTime;
    stats.pendingJobs = _pendingJobs.Count ;
    stats.numThreads = _availableThreads.Count;

    stats.jobsInProgress =
        _availableThreads.Count – findIdleThreadCount();

    return stats;
  }
```

GetStats()方法返回一个 Stats()结构，这个结构随后定义。该方法包含最小和最大的线程数，还有在构造函数中设置的其他值。现在看一下 FindIdleThreadCount()方法：

```
public int FindIdleThreadCount()
{
  int idleThreads = 0;

  for (int i = 0; i < _availableThreads.Count; i++)
  {
    if ( ( (ThreadElement)_availableThreads [i] ) . Idle)
      idleThreads++;
  }

  return idleThreads ;
}
```

这个方法在前面类中已经调用过。它只是遍历线程数组，并返回空闲线程的数目。我们也使用了 FindFirstIdleThread()方法，如下所示：

```
public int FindFirstIdleThread()
{
  for (int i = 0; i < _availableThreads.Count; i++)
  {
    if(((ThreadElement)_availableThreads[i]).Idle )
      return i;
  }

  return -1;
}
```

该方法返回线程数组中第一个空闲线程的索引。我们也需要如下的方法：

```
public int FindThread()
{
  for(int i = 0; i < _availableThreads.Count; i++)
  {
    if(((ThreadElement)_availableThreads[i])
      .GetMyThread()
      .Equals (Thread.CurrentThread ))
    return i ;
  }

  return -1;
}
```

这个方法用于确定当前线程在数组列表中的位置。我们还需要如下的方法：

```
public void RemoveThread()
{
  for(int i = 0 ; i < _availableThreads.Count; i++)
  {
    if(((ThreadElement)_availableThreads[i])
        .GetMyThread()
        .Equals (Thread.CurrentThread ))
    {
    _availableThreads.RemoveAt (i) ;
    return ;
    }
  }
}
```

这个方法将从线程数组列表中删除当前线程。当然，这也可以用于从线程池中删除线程，此时这个线程已经结束，并且处于空闲状态的时间已经超过 Me.MaxIdleTime 字段指定的值。现在为这个程序集定义其余类：

```
public class GenPool
{
  private Object _lock;
  private GenThreadPoolImpl _gn;

  public GenPool(Object lock_, GenThreadPoolImpl gn)
  {
    this._lock = lock_;
```

```
        this._gn = gn;
    }
```

GenPool 类执行所有待执行的线程，一旦线程执行完成，并且过了在 MaxIdleTime 中指定的时间，该类将从线程池中删除这些线程。该类检查在 GenThreadPoolImpl(按引用传递给构造函数)上是否有可用的线程，同时锁定作为第一个参数传递的对象的值。在通常情况下，这将和作为第二个参数传递的 GenThreadPoolImpl 对象相同。

```
    public void Run()
    {
      Thread job = null;
      try
{
  while(true)
  {
    while(true)
    {
      lock(this._lock )
        {
          if (_gn. PendingJobs. Count = = 0)
          {
            int index s= _gn.findThread();
            if(index == -1)return;

            ((ThreadElement)_gn.AvailableThreads[index]).Idle =
                  true ;
            break;
          }

          job = (Thread)_gn.PendingJobs[0];
          _gn.PendingJobs.RemoveAt (0) ;
        }//end of lock
```

Run()方法启动一个循环，试图在线程池中找到一个与当前线程相匹配的线程，并开始执行。在上面的代码中，它锁定了作为参数传递给构造函数的对象，如果没有待处理的任务，它就在线程池中查找与当前线程相匹配的线程，如果找不到则返回-1。如果存在待处理的任务，它就获取第一个线程，然后从队列中删除它。

```
      //run the job
    job.Start ();
  }
```

接着它开始在待处理的线程上执行方法，并返回到循环起点。

```
try
{
   lock(this)
   {
      if(_gn.MaxIdleTime == -1)
            Monitor.Wait (this);
      else Monitor.Wait (this, gn.MaxIdleTime) ;
   }
}
```

在循环的下一部分(一旦它没有其他的待处理任务)，它就锁定当前对象，并等待线程被释放以执行它，等待的时间在 MaxIdleTime 中指定。

```
lock(_lock)
{
   if (_gn. PendingJobs. Count == 0)
   {
      if(_gn.MinThreads != -1 && _gn.AvailableThreads.Count >
            _gn.MinThreads)
      {
         _gn.removeThread()
            return;
       }
      }
   }
```

最后，它再次锁定这个对象，此时如果没有待处理的任务，并且此时的线程数大于所要求的最小线程数，就从线程池中删除线程。现在讨论 ThreadElement 类：

```
public class ThreadElement
{
   private bool _idle;
   private Thread _thread;

   public ThreadElement(Thread th)
   {
      this._thread = th;
      this._idle = true;
   }
```

ThreadElement 存储在线程池中，它将一个线程作为其构造函数的参数。在构造这个对象时，它将线程设置为空闲线程。

```
public bool Idle
{
   get
   {
      return this._idle;
   }
   set
   {
      this._idle = value;
   }
}
public Thread GetMyThread(){return this._thread;}
```

上述代码比较简洁易懂。Idle 属性定义了线程的执行什么时候完成，GetMyThread() 方法返回 Thread 对象。现在看一下下面的结构：

```
public struct Stats
{
   public int maxThreads;
   public int minThreads;
   public int maxIdleTime;
   public int numThreads;
   public int pendingJobs;
   public int jobsInProgress;
```

上面定义了前面提到的 Stats 结构，它存储了线程池的所有统计信息。其中的字段都是自描述的，ToString()是其中惟一的方法：

```
public override String ToString()
{
   StringBuilder sb = new StringBuilder ("MaxThreads = ");
   sb.Append(maxThreads) ;
   sb.Append("\nMinThreads = ");
   sb.Append(minThreads);
   sb.Append("\nMaxIdleTime = ");
   sb.Append(maxIdleTime) ;
   sb.Append("\npending Jobs == ") ;
   sb.Append(pendingJobs) ;
```

```
        sb.Append("\nJobs In Progress = ");
        sb.Append(jobsInProgress) ;

        return sb.ToString ();
    }
```

ToString()方法以字符串格式返回这个结构，字符串用 StringBuilder 构建。参数 107 将 StringBuilder 的大小初始化为 107 个字符，因为线程数不可能超过 99 999 个。这样，StringBuilder 无论如何都要调整其自身的大小。这种容量规格能使性能稍有提高。

如果应用程序重复地在不同的线程上触发方法，这个类就可以管理该进程，并帮助确保不会生成太多的线程。除了包含最大和最小线程数之外，它还将重用现有的线程。现在将这个项目编译成 DLL，并在其他项目中使用这个类。下面的代码 TestGenThreadPool.cs 可以测试这个线程池类：

```
using System;
using System.Threading;
using GenThreadPool;

namespace TestGenThreadPool
{
    .
  public class TestPerformance
  {
    public int count;
    private Object _lock = new Object();

    public TestPerformance(IThreadPool pool, int times)
    {
count = 0 ;
DateTime start =System.DateTime .Now;
Console.WriteLine("Start Time for Job is "
                    + System.DateTime .Now) ;

for (int i = 0; i < times; i++)
{
  Thread tl = new Thread(
      new ThreadStart (new Job(this).Run ));
  pool.AddJob(t1) ;
}
```

```
Console.WriteLine("End Time for Job is " +
                  System.DateTime .Now);

Console.WriteLine("Performance using Pool[in ms]: ") ;
Console.WriteLine(""
                  + (System.DateTime.Now - start).ToString());

count= 0 ;
start = System.DateTime.Now;

Console.WriteLine("Start Time for JobThread is " +
                  System. DateTime . Now. ToString () ) ;

for (int i = 0; i < times; i++)
{
   Thread jt = new Thread(new ThreadStart(new Job(this).Run));
   jt.Start() ;
}

while (true)
{
   lock (_lock)
   {
      if (count == times)
         break;
   }
   try
   {
      Thread.Sleep(1000) ;
   }
   catch
   {
   }
}

Console.WriteLine("End Time for JobThread is "
                  + System.DateTime.Now.ToString());
Console.WriteLine("Performance using no Pool[in ms]: ");
Console.WriteLine(""
                  + (System.DateTime.Now - start).ToString());
```

```
}
sealed class JobThread
{
   private Object _lock = new Object();
   private TestPerformance tpf;

   public JobThread(TestPerformance tpf_)
   {
      this.tpf = tpf_;
   }

   public void Run()
   {
      lock(_lock)
      {
         tpf.count++ ;
      }
   }
}

sealed class Job
{
   private Object _lock = new Object();
   private TestPerformance tpf;

   public Job(TestPerformance tpf_)
   {
      this.tpf = tpf_;
   }

      public void Run()
      {
         tpf.count++;
      }
   }
}

class TestPool
{
```

```
    static void Main(string[] args)
    {
      GenThreadPool.IThreadPool tp =
            new GenThreadPoolImpl(200, 300, 300, true);
      TestPerformance p = new TestPerformance (tp, 100) ;
    }
}
```

上述的应用程序仅仅是机械地向线程池的实例中添加新的线程，并将调试标记设置为 true。这段代码虽然比较易懂，但是搞清楚线程池运转情况的最好方法就是在自己的应用程序中试验一下。编译后就可以使用这个类。

## 5.6　本章小结

本章介绍了当在一个相对较短的时间内需要线程时，如何使用线程池。线程池允许重复利用线程。线程被分配一个任务，然后当任务完成时，线程就返回到线程池中，等待下一次分配。我们还讨论了在.NET 应用程序中使用线程池涉及的各种问题。首先讲述了什么是线程池，以及为什么要在应用程序中使用线程。在讲解了线程池的缺陷之后，还谈及了 CLR 在创建线程池中的作用。

随后论述了一些可伸缩性问题，因为 ThreadPool 类不适用于需要触发生存期较长的线程的应用程序。我们探讨了 ThreadPool 管理器类的创建过程，并提到了 SMP 系统如何显著提高应用程序的性能(应用程序必须以线程方式运行)。

# 第6章　调试与跟踪线程

调试和跟踪是开发人员频繁使用、并且必须经常使用的两种技术。前者允许开发人员分析应用程序的变量和代码，并单步运行程序的代码流。后者则可以跟踪应用程序的行为，在侦听器(日志文件，Windows 事件日志记录，或其他类似的文件)中显示信息。这两种技术是创建强大应用程序的基础，因为它们提供了一种简单的方式来监控和了解应用程序是如何工作的。这两种技术之间的主要差别在于，跟踪可以在应用程序运行时使用，而调试器则在设计时使用，即在发布应用程序的最后版本之前使用。

桌面应用程序的开发人员一直都拥有极佳的调试支持，并可以使用断点，查看变量的内容。.NET 也不例外，但应考虑在多线程应用程序中使用断点的内在问题，这也是本章的论题。

除了桌面应用程序之外，开发人员由于缺乏在诸如 ASP 等环境下编写的 Web 应用程序调试器而倍感痛苦。为了了解变量值、代码流以及通常使用调试器完成的每一个普通任务，ASP 开发人员经常不得不在代码中使用 Response.Write()语句、像 Entered the function、Exited from the loop 等诸如此类的反馈消息。然后，测试完 ASP 应用程序后，还需要删除所有不需要的语句。这不是调试程序的最佳方式。

.NET 提供了 4 种有用的类：Trace、 Debug、 BooleanSwitch 和 TraceSwitch，为下一代的 ASP 开发人员提供了调试功能。另外，任何一种.NET 语言都可以使用这些类，并且，每一个选择使用 Visual Studio .NET 创建应用程序的开发人员都可以使用它的可视化工具来执行调试操作。

各种各样的跟踪和调试技术对于使用线程的应用程序特别有用。如果得以良好实现，这些技术允许开发人员跟踪每一个线程的行为，揭示任何一个应用程序的异常，例如意外的资源消耗、故障等。

本章将按照以下顺序分析跟踪和调试技术。

- 使用 Visual Studio .NET 调试分析，并使用它的强大工具。
- 使用.NET 的跟踪类，在代码中实现这些功能。
- 创建使用跟踪的应用程序，把它们装配在一起。

就本章来说，Visual Studio .NET 对于利用上述大部分跟踪和调试功能是必不可少的。然而，通过/d:TRACE=TRUE 选项，一些跟踪功能也可以使用命令行来完成。

## 6.1　创建应用程序代码

当创建应用程序(或其中的一部分)时，通常是先编写代码，然后尝试运行应用程序。有时它会按照期望的那样工作，但是通常并不是这样。当应用程序没有按照期望的方式运行时，就要仔细检查所编写的代码，查找原因所在。在 Visual Studio .NET 中，可以使用调试器在错误方法的附近或之前选择一些断点，来停止应用程序的执行，然后逐行执行代码，检查变量值，确切地了解是什么出了问题。最后，当所有代码都能够正常运行时，就可以构建应用程序的最后版本(不包含调试工具使用的符号)，并发布应用程序。

这种程序在开发过程中，还可以插入跟踪功能。事实上，即使程序运行得非常好，也经常会出现没有预见到的情形(特别是某个外部的，或许是第三方的组件失败)。在这种情况下，如果已在代码中添加了跟踪指令，就可以开启跟踪功能，检查随之产生的日志文件，来了解发生了什么事情。此外，跟踪功能还可以用于找出应用程序在哪里使用了资源或者在哪里花费了很多时间来完成任务。在使用线程的应用程序中，应该使用跟踪功能，因为观察每一个线程的行为，识别竞争条件，找出潜在的死锁或费时的内容可能很困难。

跟踪、调试以及性能技术常称为测试工具。这个术语涉及到监控应用程序的性能、诊断错误和故障的能力，所以支持测试工具的应用程序必须包括以下内容：

- 调试：在应用程序的开发期间修改错误和故障
- 代码跟踪：在应用程序的执行过程中，从侦听器程序中接收信息
- 性能计数器：使用技术来监控应用程序的性能

下面来分析一下.NET Framework 为将测试工具添加到应用程序中提供了什么功能。

## 6.2　调试代码

通常，当测试应用程序，并且看到程序的行为不如所愿时，就要认真地检查所编写的代码。如果使用 Visual Studio .NET 来创建应用程序，它会提供很多令人惊异的工具来可视化地调试应用程序。另外，无论选择哪种语言开发应用程序，都会使用带有同样工具的同一个调试器。此外，调试器的基本功能是从 Visual Basic 6 和 Visual C++ IDE 继承而来，因此绝大多数开发人员都很熟悉这些功能。但是，本书不会在调试器上花

费太多的时间，而是主要论述和线程直接相关的那些功能。

新的调试器提供以下功能：

- 用同样的工具调试由不同的语言开发的不同应用程序，还可以调试用混合语言编写的应用程序
- 调试 SQL Server 存储过程
- 调试.NET Framework 和 Win32 本地代码，所以如果调试的是 Visual C# .NET 应用程序，且线程使用 COM+组件，就可以使用同样的调试器调试这两种应用程序
- 更强大和增强的远程调试器

如果用户有使用 Visual Basic 6 调试器的经验，就知道一些功能已经被去掉了。最相关的功能是修改代码和继续执行的能力。使用 Visual Studio .NET 调试器，这个功能就不能再用了，因为对代码的每次修改都需要重新编译代码。

本节将分析由 Visual Studio .NET IDE 提供的调试工具，该工具在开发多线程应用程序的测试和差错发现阶段会特别有用。

## 6.2.1　Visual Studio .NET 调试器

使用 Visual Studio .NET 调试器，只需要在想检查的代码行附近插入断点，就可以在指定的地方中断应用程序的执行。当应用程序挂起时，调试器提供了很多工具来检查变量的内容，编辑变量的值，检查存储器和调用堆栈等。

### 1. 配置调试器参数

要利用 Visual Studio .NET 调试器，就必须使用 Debug 配置构建应用程序。这将使用象征性的调试信息来填充应用程序，而不是优化代码。当一切运行良好时，就可以删除调试信息，优化代码，之后重新编译代码，选择 Release 配置，最后发布应用程序。

当新的调试器会话开始时，许多资源都加载到内存中。事实上，调试器在内存中填充了各种各样的代码，以调试非托管代码、SQL Server 存储过程等。所以，最好在不需要这些功能的时候删除它们。在 Solution Explorer 中右击项目，选择 Properties，在 Property Pages 对话框中修改调试器的设置。对于 Windows 应用程序，会显示如图 6-1 所示的对话框。

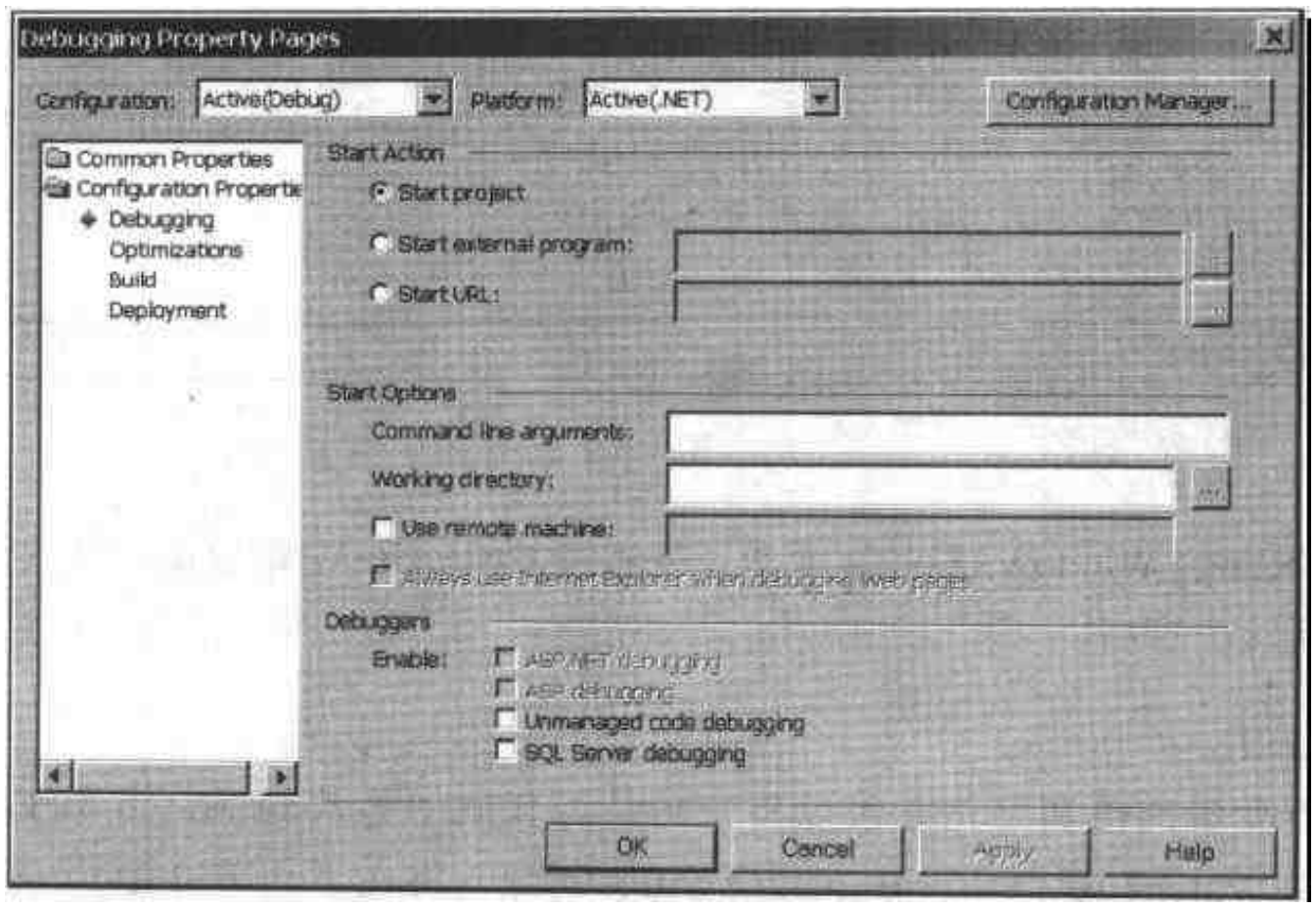


图　6-1

在编译一个 Debug 配置的项目后，输出目录将包括 EXE 或 DLL 文件和一个 PDB (程序数据库)文件。由于 IL 保存了参数和数组中私有成员的值，所以这些变量的原始名称就丢失了，与调试相关的其他信息也会丢失。当为了调试而编译项目时，或在命令行编译器中使用/debug:full 选项时，会同时生成 PDB 文件。在 EXE 或 DLL 文件内部包括一个指向 PDB 文件的绝对路径，如果调试器找不到程序数据库文件，它就开始在同一个应用程序路径和在 Property Pages 对话框中指定的目录中进行搜索。最后，如果调试器不能在任何目录中找到 PDB 文件，它将重新生成一个新的 PDB 文件。

### 2. 调试器窗口

一旦把项目加载到 Visual Studio .NET 中，就可以调试应用程序了，具体方法是，运行项目，等待代码运行到一个断点，或者按 F10 键或 F11 键步进到代码中的方法调用。如果运行的不是应用程序的发布版本，IDE 将显示很多停泊窗口(docked windw)。在调试会话期间，这些窗口会显示变量的值、对象的转储、调用堆栈、分解代码等。现在开始更深入地分析这些调试工具，看看它们是如何帮助调试线程应用程序的。

#### 本地窗口

这个窗口允许检查和修改在所调试方法中的每个在本地定义的变量的内容(包括方法的参数)。例如，调试 TraceSwitchExample 中的 Main()方法，将获得 fs 和 t 这两个变量的内容，如图 6-2 所示。

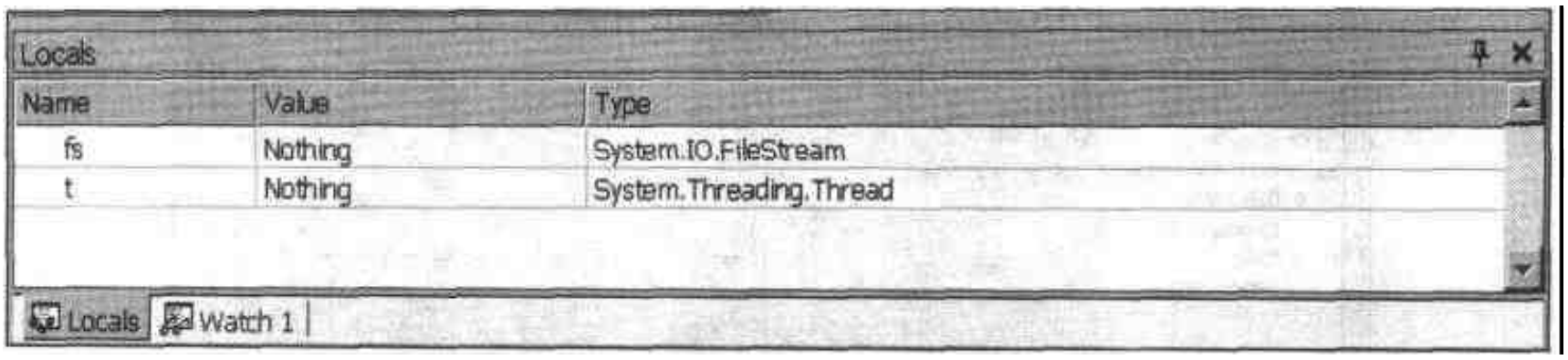


图　6-2

选择 Debug | Window | Locals 菜单，或者按 Ctrl+Alt+V 组合键，释放后再按 L 键，就可以激活这个窗口。

**监视窗口**

要检查变量的值和结构，可以将变量从源代码中拖放到 watch 窗口上。在如图 6-3 所示的 Watch1 窗口中，BooleanSwitch 对象 (将在下一节介绍)被拖放到这个窗口中。

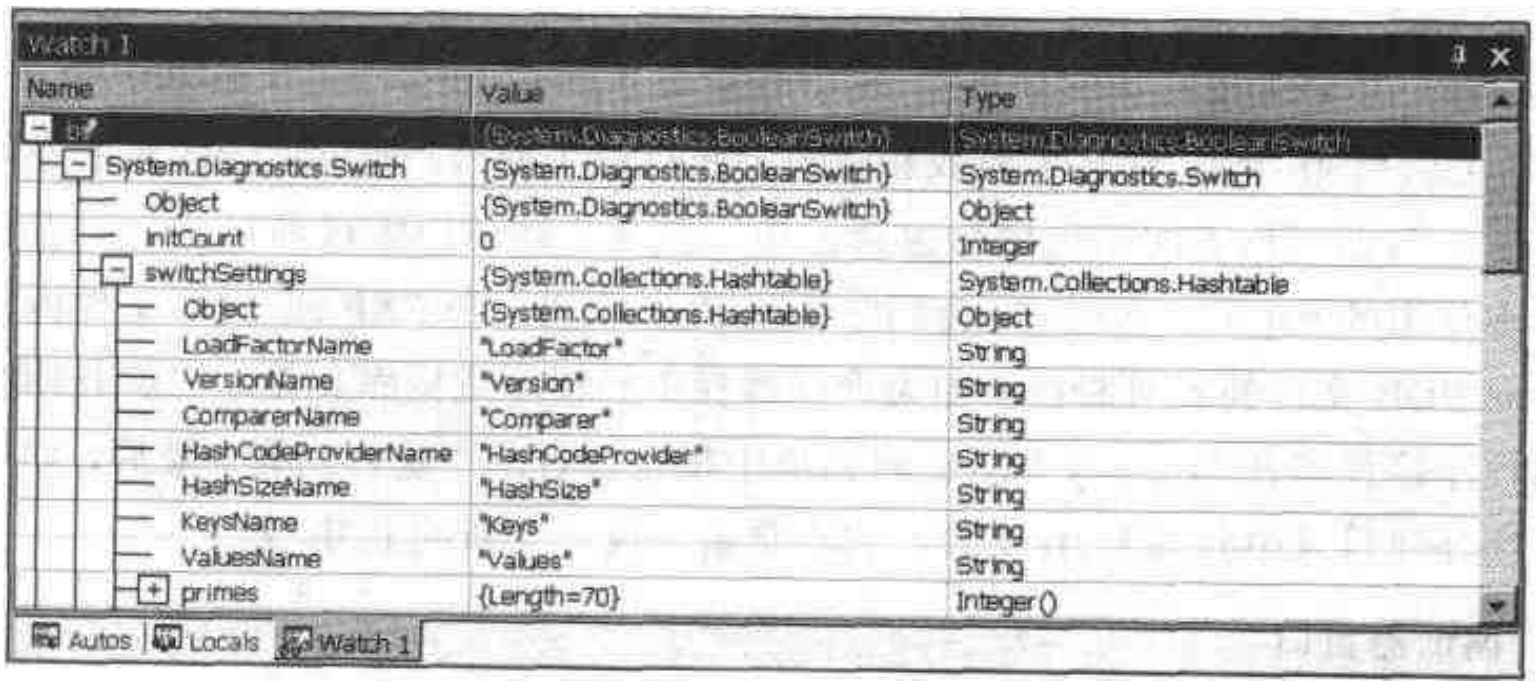


图　6-3

单击加号展开树状节点，就可以检查和修改对象的属性值了。按 Ctrl+Alt+W 组合键，放开，然后按 1 到 4 中的一个键，就可以激活 4 个 Watch 窗口。

**注意：**

也可以在源代码中选择一个变量，并从弹出菜单中选择 Add Watch，把变量添加到 Watch 窗口中。

**命令窗口——即时窗口**

这个窗口提供了一个文本框，从中可以查询变量的内容，修改变量的值。当需要检索变量的内容时，必须在表达式之前加上一个问号。如图 6-4 所示，检查了 BooleanSwitch 对象的 Enabled 属性，将其修改为 False，并重新显示。

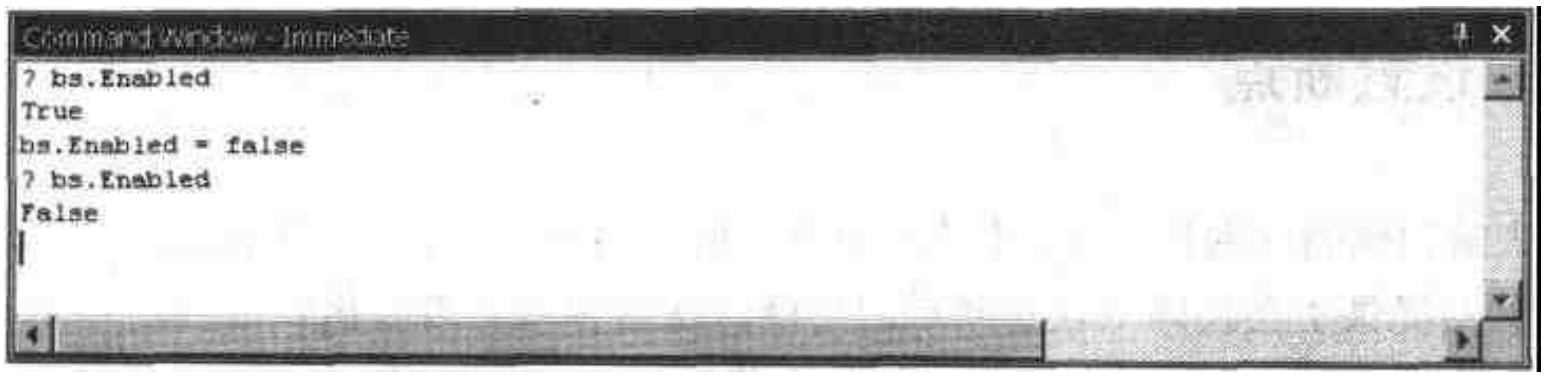


图　6-4

另外，这个窗口还允许使用各种 IDE 命令：例如创建新的文件或项目、查找字符串，以及其他在 Visual Studio .NET 菜单中经常使用的命令。要从即时模式切换到命令模式，只需写入>cmd 语句即可。在命令模式下，IDE 可以通过智能识别功能查找所需的命令。要切换回即时模式，可使用>immed 命令。

选择 Debug | Window | Immediate 命令，或者按 Ctrl+Alt+I 组合键，就可以激活这个窗口。

## 6.2.2　单步执行代码

前面简要描述了一些非常有用的调试窗口，下面开始集中论述代码的调试。Visual Studio .NET 调试器允许开发人员在单行代码和多行代码之间移动，在运行时观察程序的行为。而且，还可以调试未托管的代码和 Microsoft SQL Server 存储过程。调试器提供了 3 种不同的方式来单步执行代码。

- Step Into：按 F11 键，每次仅能在代码中向前执行一步，并进入执行过程中所遇到的方法体(可使用源代码和调试符号)。
- Step Over：按 F10 键，在代码中向前执行一步，将执行所遇到的每个方法，但不进入方法体内部(将方法作为一行代码来执行)。
- Step Out：按 Shift+F1 键，将会执行当前进入的方法体中的所有代码，然后执行调用该方法的代码行的下一行。

每次按这些键执行到下一行代码时，其实都在执行醒目显示的代码。

Visual Studio .NET 调试器提供的另一个有用的功能是 Run To Cursor。在源代码的弹出菜单中选中它，就可以执行在高亮显示的命令行和光标所在的命令行之间的所有代码行。

最后，Visual Studio .NET 调试器提供了一种修改应用程序的执行点的方式。启动调试器，在弹出菜单中选择 Set Next Statement 项，就可以移动应用程序的执行点。但是使用这个功能时一定要小心，因为新旧位置之间的每行代码就不再执行了。

## 6.2.3　设置断点

在大的源代码应用程序中，不能单步执行所有的代码，来到达在调试中感兴趣的方法。调试器提供了在代码中设置断点的功能。从名称就可以看出，断点是一个点，程序执行到这个点时就会被挂起。将光标放在代码行上并按 F9 键，或者单击代码行左边的空白处，就可以在启动调试器会话之前和之后指定断点。该行将以红色醒目地显示出来，让开发人员知道已经在此代码中添加了一个断点，在源窗口的左边空白处还有要添加一个符号。要删除断点，可以在该符号上单击或者再次按 F9 键。

使用如图 6-5 所示的 Breakpoints 窗口，就可以在这个窗口中管理所有设置的断点。

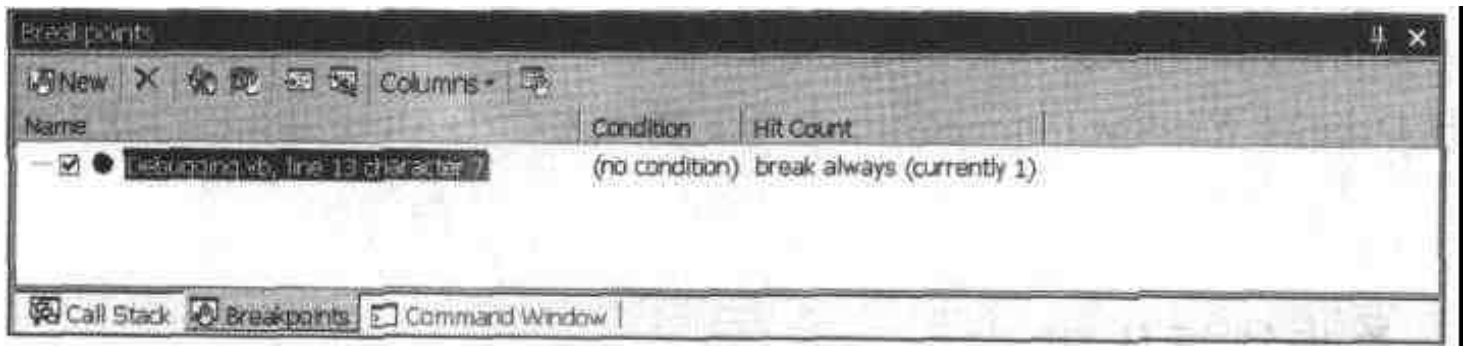


图　6-5

使用这个窗口，可以添加新的断点、删除一个或所有的断点、使所有断点失效、添加和删除窗口的列以及查看断点的属性。

**注意：**

使用断点，可以挂起线程的执行，检查当前堆栈框架的内容。

从上面的窗口中选择断点属性选项，打开一个新的对话框，在此可以指定只有在某个变量的内容改变时，才激活一个断点。必须在断点属性页中选中 has changed 单选按钮，才能指定变量的名称。在线程中这也是有用的，因为它可以检测出某种不可预料的事情的发生。

最后，该对话框中的 Hit Count...按钮允许开发人员仅在断点达到了指定的触发数时才激活断点。这在调试线程时也同样有用，因为这样可以了解多长时间生成一个线程，如图 6-6 所示。

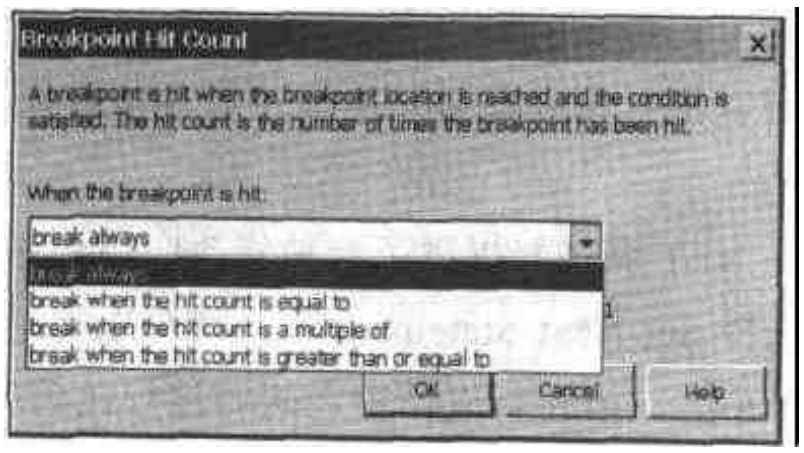


图　6-6

从这个组合框中，可以选择想指派给断点的条件。例如，只有当快要从循环中退出时，才激活循环中的断点。当触发计数等于一个数据项时，可以为组合框旁的文本框赋值，来选择中断。

要在到达断点之前执行所有代码行，就必须按 F5 键，或者选择 Debug | Start 菜单项，或者按标准工具栏上的 Start 按钮。

## 6.2.4　调试线程

Visual Studio .NET 调试器提供了一种特殊的窗口，用于在调试会话期间管理线程。选择 Debug | Windows |Threads，或按 Ctrl+Alt+H 组合键，就可以显示该窗口，如图 6-7 所示。

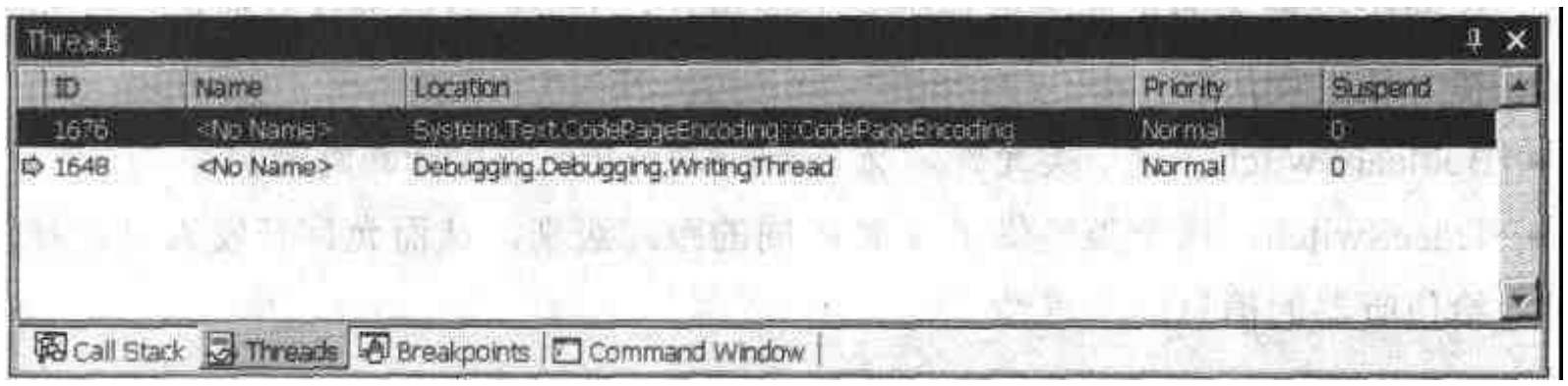


图　6-7

线程窗口包含以下列，如表 6-1 所示。

表 6-1

| 列　名 | 说　明 |
|---|---|
| ID | 由操作系统分配的线程的惟一标识符 |
| Name | 线程的名称。可以使用 Thread 对象的 Name 属性在代码中指定它 |
| Location | 包含当前执行线程的方法或内存地址 |
| Priority | 线程的优先级 |
| Suspend | 用来确定线程被挂起多少次的一个计数器。上图中的计数 0 表示线程从来没有被挂起 |

双击 Threads 窗口中的项，就可以在线程之间切换。此外，右击一个线程，从弹出的菜单中选择 Freeze 菜单项，可以暂停线程的执行。要回滚冻结的线程状态，可选择 Thaw 菜单项。

## 6.3　代码的跟踪

下面要分析的测试工具是跟踪。在多线程应用程序中，这项技术尤其重要。当启动很多任务时，可以跟踪线程的行为以及交互情况。这些用调试器是不可能实现的。.NET Framework 提供了一些有用的类，允许开发人员实现跟踪功能。下面介绍.NET Framework 提供的 System.Diagnostics 命名空间中的跟踪类：

- Trace：这个类有很多向侦听器中写入信息的静态方法。在默认情况下，VS.NET 中的调试输出窗口用作侦听器应用程序，但是由于有 Listeners 集合，还可以添加不同的侦听器，例如文本文件侦听器或 Windows 事件日志记录侦听器。
- Debug：这个类具有与 Trace 类相同的方法，向侦听器应用程序中写入信息。这两个类最大的不同之处在于它们的用法：Trace 类在运行时使用；Debug 类在开发时使用。
- BooleanSwitch：这个类允许定义一个开关来开启或关闭跟踪消息。
- TraceSwitch：这个类提供了 4 种不同的跟踪级别，从而允许开发人员选择发送给侦听器的消息的严重性。

### 6.3.1　Trace 类

本节将分析使用频率最高的 Trace 类的方法。该类由.NET Framework 提供，并封装了所有能实现跟踪功能的必要方法。Trace 类包含在 System.Diagnostics 命名空间中，它提供了很多向侦听器应用程序中发送消息的静态方法。静态方法表示不必从 Trace 类中实例化新的对象，就可以直接使用方法。例如：

```
static void Main()
{
   Trace.WriteLine(t.ThreadState) ;
}
```

上面的代码段使用 WriteLine()方法，向侦听器应用程序输出线程状态，并附随一个回车符。表 6-2 列出了 Trace 类提供的一些静态方法。

表 6-2

| 方　法 | 说　明 |
| --- | --- |
| Assert(*condition, message*) | 当提供给方法的条件为 False 时，显示指定的字符串消息。当没有指定消息文本时，改为显示调用堆栈 |
| Fail(*message*) | 与 Assert()方法相似，当发生故障时，这个方法向调用堆栈中写入指定的文本。由于 Fail()不能在显示错误之前指定条件，所以，该方法与 Assert()方法有所不同。事实上，Fail() 方法通常放在 try-catch-finally 指令的 catch 语句中。还可以把它放在代码中肯定执行不到地方，例如如果确信列举出了所有的可能性，就可以把它放在 switch 语句的 default 部分 |
| Write(*message* \| *object*) | 向侦听器应用程序中写入指定的文本消息或对象名称 |
| WriteIf(*condition, message*) | 如果指定的条件为 True，就向侦听器应用程序中写入指定的消息文本 |
| WriteLine(*message* \| *object*) | 编写指定的消息文本或者对象名称，并在其后附随一个回车符 |
| WriteLineIf(*condition, message*) | 如果指定的条件为 True，编写指定消息文本，后跟一个回车符 |

这些方法的行为依赖于所选择的侦听器应用程序。例如，当指定默认侦听器时，Assert()方法会显示一个消息框。

### 1. 默认的侦听器应用程序

Trace 类提供了一个 Listeners 集合，允许添加新的侦听器应用程序。当没有新的侦听器对象添加到集合中时，Trace 类就会使用默认的侦听器应用程序：Output 调试窗口。这个窗口由 Visual Studio .NET IDE 在调试期间提供。现在看一个简单的示例 TraceExample1：

```
static void Main()
{
   Trace.WriteLine("Entered Main()") ;

   for (int i = 0; i < 6; i++)
      Trace.WriteLine(i) ;

   Trace .WriteLine ("Exiting from Main()");
}
```

这个例子的代码很简单；仅编写了当从 Main()方法中进入和退出时的跟踪信息，在循环中加上变量值。在图 6-8 中，可以看到 Visual Studio .NET Output 侦听器是如何显示信息的。

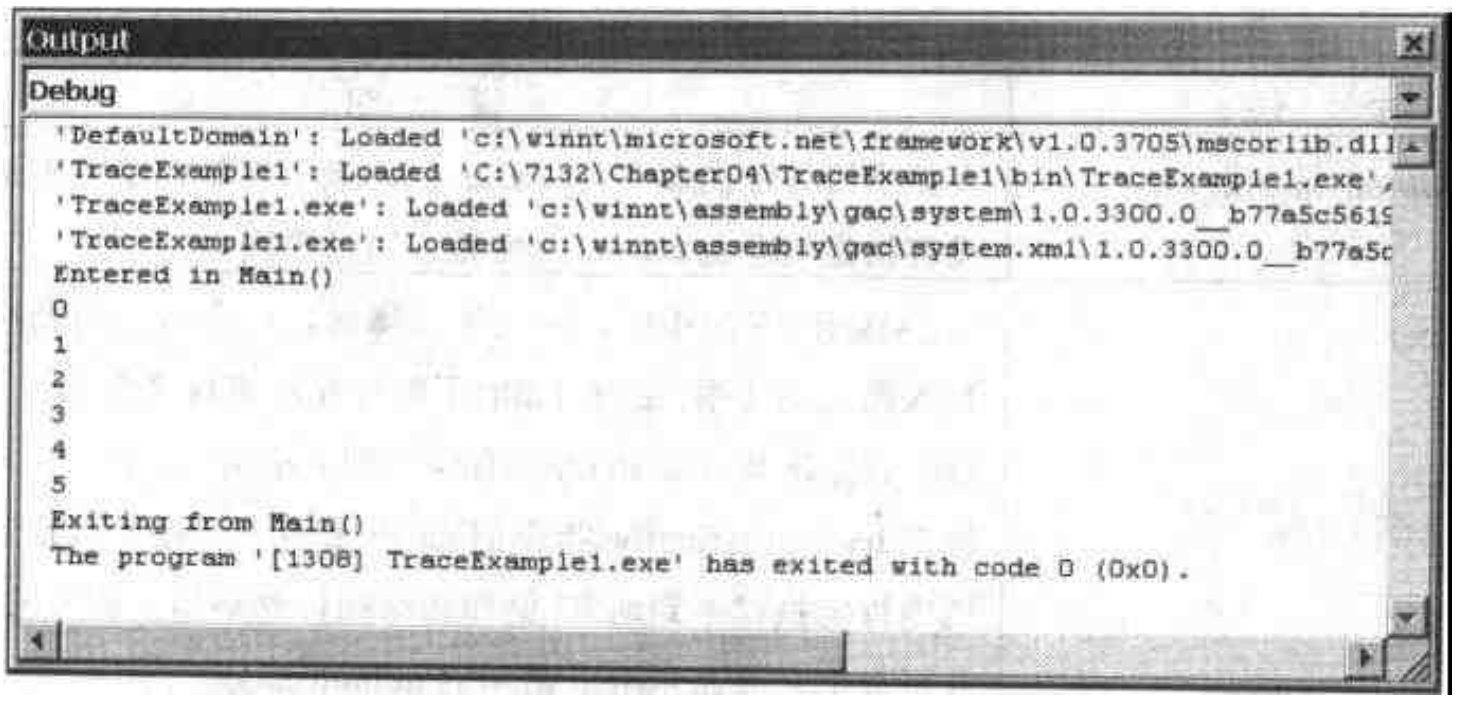


图　6-8

Trace 类还提供了两个有用的方法来声明错误通知：Assert()方法和 Fail()方法。Assert()方法允许开发人员检查作为参数提供的条件，当这个条件为 False 时就向侦听器中写入一条消息。每当出现故障时，Fail()方法就向侦听器中写入一条消息。当没有其他的侦听器添加到集合中时，Assert()方法就显示一个消息框，通知用户断言失败。当 SQL Server 服务被故意停止从而触发一个连接异常时，就可以测试下面的代码片段 TraceAssert.cs：

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Threading;
using System.Diagnostics;

namespace TraceAssert
{
  class Class1
  {
    [STAThread]
    static void Main(string() args)
    {
      // Create a thread
      Thread t;
      t = new Thread(new ThreadStart(DBThread)) ;

      // Start the thread
      t.Start() ;
```

```
}
private static void DBThread()
{
   // Create a connection object
   SqlConnection dbConn = new
      SqlConnection("server=.;database=pubs;uid=sa;pwd=") ;

   // Create a command object to execute a SQL statement
   SqlCommand dbComm = new
         SqlCommand("SELECT * FROM " + "authors", dbConn);

   SqlDataReader dr = null;

   Trace.WriteLine(DateTime.Now + " - Executing SQL statement");

   try
   {
      // Open the connection to the database
      dbConn.Open() ;

      // Assert that the connection opened
      Trace.Assert(dbConn.State == ConnectionState.Open,
            "Error", "Connection failed...") ;

      // Execute the SQL statement
      dr = dbComm.ExecuteReader(CommandBehavior.CloseConnection);

      // Assert that the statement executed OK
      Trace.Assert(dr != null, "Error",
            "The SqlDataReader is null!") ;

      while (dr.Read())
      {
         // Reading records
      }
   }
   catch
   {
      // Log the error to the Trace application
      Trace.Fail("An error occurred in database access");
```

```
            }
            finally
            {
               if ((dr.IsClosed == false) && (dr!=null))
                  dr.Close();
            }
         }
      }
   }
```

在 Main()方法中创建并启动了一个新线程。这个新线程运行包含在 DBThread()方法内的代码。这段代码只是尝试联系 pubs SQL Server 数据库，并检索包含在 authors 表中的所有数据。如果无法获得 SQL Server 服务，就会对代码的异常显示如图 6-9 所示的故障断言。

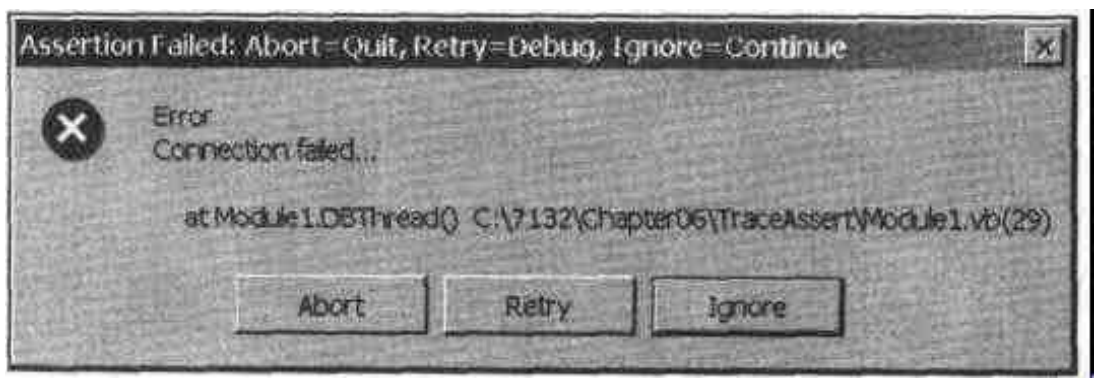


图　6-9

触发上面断言的行是：

```
Trace.Assert (dbConn. State == ConnectionState.Open,
    "Error", "Connection failed...") ;
```

第一个参数检查连接的状态是否为 Open。当连接没有打开时，它就设置为 False，以显示该断言。如本章后面所述，可以使用应用程序配置文件来禁用跟踪消息功能。这样，就可以决定是否在运行期间显示断言消息。

## 6.3.2　使用不同的侦听器应用程序

本节将介绍如何修改默认的侦听器应用程序。Trace 类(和稍后介绍的 Debug 类)提供了 Listeners 集合，该 Listeners 集合包含了一系列的侦听器应用程序。无需添加任何新的侦听器类，DefaultTraceListener 将指向由 Visual Studio .NET 提供的 Output 调试窗口。然而，.NET Framework 提供了另外两个可以用作侦听器应用程序的类,如表 6-3 所示。

表 6-3

| 类 | 说　明 |
| --- | --- |
| EventLogTraceListener | 使用该类，可以向 Windows 事件日志中重定向跟踪消息 |
| TextWriterTraceListener | 使用该类，可以将跟踪消息重定向到文本文件或数据流 |

在多线程应用程序中，如果需要跟踪应用程序在 Visual Studio.NET 之外执行期间的行为，就可以使用上面的侦听器来修改默认侦听器。当然，Output 调试窗口只有在调试期间才可用。使用这两个类，可以选择是将跟踪消息放在 Windows 事件日志中还是放在文本文件中。通常，如果应用程序在配备了事件日志的操作系统中运行，EventLogTraceListener 类就是最佳解决方案。其原因如下：

(1) 事件日志由操作系统管理。

(2) 事件日志允许管理员为日志文件指定安全设置。

(3) 事件日志必须使用 Event Viewer 阅读。以一个可视的界面进行显示比在记事本中以文本文件形式显示要好得多。

改变默认侦听器很简单，下面以 TraceEventLog.cs 为例，代码如下：

```
static void Main(string[ ] args)
{
  // Create a trace listener for the event log.
  EventLogTraceListener eltl = new EventLogTraceListener("TraceLog");

  // Add the event log trace listener to the collection.
  Trace.Listeners.Add(eltl) ;

  // Write output to the event log.
  Trace .WriteLine ( "Entered in Main()");
}
```

首先，必须创建一个新的侦听器对象。在上面的示例中，创建了一个新的 EventLogTraceListener，以便将 Windows 事件日志用作侦听器应用程序。类构造函数接受了一个字符串，在该字符串中可以为已写入记录项的源指定名称。构造函数将自动实例化一个新的 EventLog 对象，并将指定的源名称自动赋给 EventLog 类的 Source 属性。

下一步是使用 Add()方法向 Listeners 集合中添加新的侦听器对象，并给该侦听器对象提供引用。最后，就可以开始写入要重定向到侦听器应用程序的跟踪消息了。

使用 Event Viewer 应用程序打开 Windows 事件日志，就会看到出现在 Application Log 部分的新记录项，如图 6-10 所示。

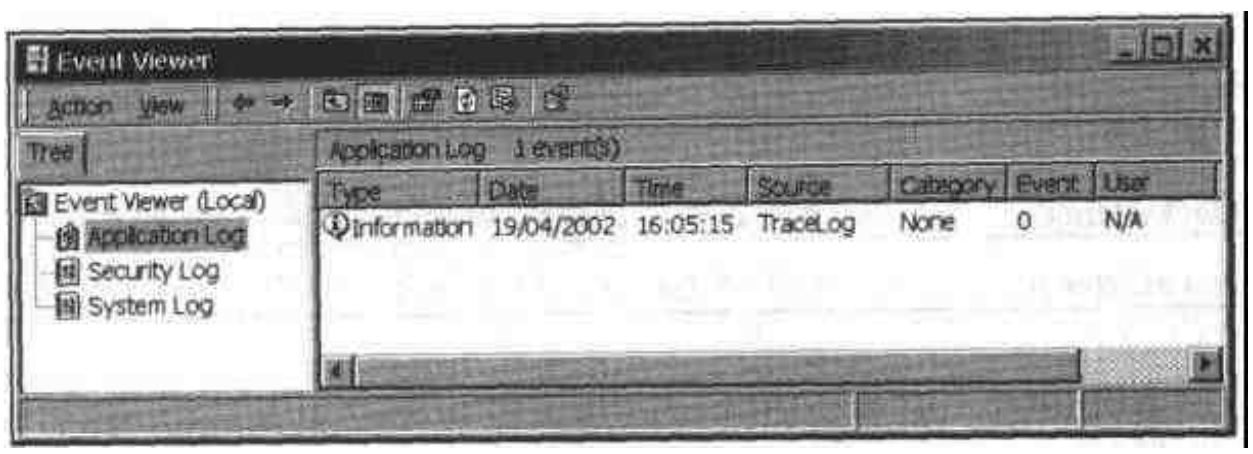


图　6-10

双击在 Application Log 报告中的数据项可以查看消息，如图 6-11 所示。


图　6-11

上面的代码向 Listeners 集合中添加了一个新的侦听器，以便在 Output 调试窗口和事件日志中接收跟踪消息。如果想删除默认侦听器，只使用事件日志应用程序，则必须调用 RemoveAt()方法，其代码如下：

```
static void Main(string[ ] args)
{
  // Create a trace listener for the event log.
  EventLogTraceListener eltl = new EventLogTraceListener("TraceLog");
  // Remove the default listener
  Trace.Listeners.RemoveAt(0) ;

  // Add the event log trace listener to the collection.
```

```
    Trace.Listeners.Add(elti) ;

    // Write output to the event log.
    Trace. WriteLine ( "Entered in Main()");
}
```

### TextWriterTraceListener 类

下面介绍 TextWriterTraceListener 类，以总结侦听器。当必须向文本文件或直接在控制台应用程序中写入跟踪消息时，就可以使用该类。事实上，在 TextWriterTraceListener 对象的创建期间，可以指定 TextWriter 对象或者 Stream 对象。使用 Stream 对象可以指定如何处理文件流的更多细节。下面的代码 TraceConsole.cs 展示了如何在控制台应用程序中跟踪消息。

```
static void Main(string[ ] args)
{
   // Remove the default listener
   Trace.Listeners.RemoveAt(0) ;

   // Add a console listener
   Trace.Listeners.Add(new TextWriterTraceListener(Console.Out)) ;

   // Write a trace message
   Trace. WriteLine ("Entered in Main()");
}
```

在类的构造函数中指定 Console.Out 流，控制台应用程序将显示跟踪信息，如图 6-12 所示。

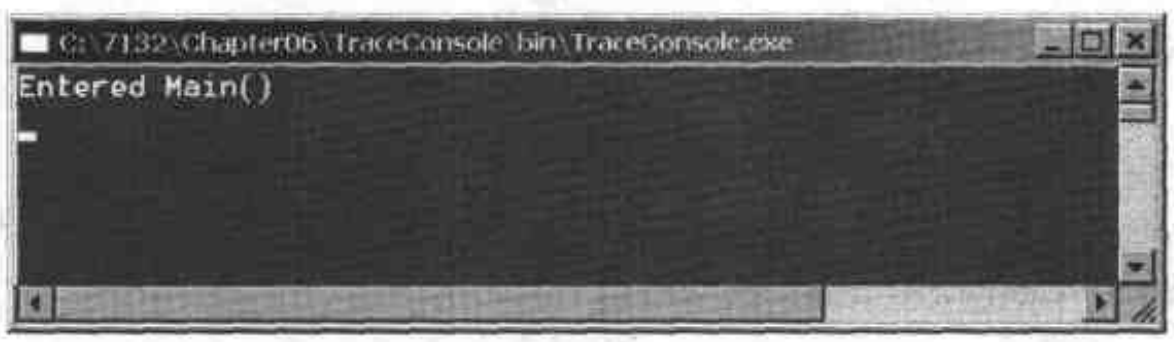


图　6-12

最后，介绍一下如何把文本日志文件添加为侦听器。我们须添加一个新的 TextWriterTraceListener 对象，并在它的构造函数中指定 FileStream 对象。当应用程序结束时，必须使用由 Trace 类提供的静态的 Close()方法来关闭写入所有跟踪消息的日志文件。在下面的 Debugging.cs 代码中，启动了一个能同时跟踪主辅线程消息的线程：

```
private static void WritingThread()
{
  // Trace an info message
  Trace.WriteLine(DateTime.Now + " - Entered WritingThread()");

  // Sleeping for one sec....
  Thread.CurrentThread.Sleep(1000) ;

  // Trace an info message
  Tracc.WriteLine(DateTime.Now + " - Slept for 1 second...");
}
```

该线程使用 WritingThread()方法睡眠一秒钟，之后写入了一些跟踪消息。

现在，创建一个新的 FileStream 对象，创建或者打开 Debugging.log 文件(如果该文件已经存在)。然后，通过在 Add()方法内部创建 TextWriterTraceListener 类的一个新实例，向 Listeners 集合中添加新的侦听器：

```
static void Main(string[ ] args)
{
  // Create a file listener
  FileStream fs = new
      IO.FileStream("C:\Debugging.log", IO.FileMode.OpenOrCreate) ;
  Trace.Listeners.Add(new TextWriterTraceListener(fs)) ;
```

在启动了线程以后，代码等候用户按回车键，然后关闭侦听器应用程序，并把所有的跟踪消息填入到日志文件中。

```
// Write the line only when the switch is on
Trace. WriteLine (DateTime. Now + " - Entered Main()");

// Create a thread
Thread t;
t = new Thread(new ThreadStart(WritingThread));

// Start the thread
t.Start() ;

// Wait for the user carriage return
Console.Read() ;

// Close the file listener flushing the trace messages
```

```
    Trace. Close() ;
}
```

输出结果如下所示：

```
30/04/2002 16:38:15 - Entered Main()
30/04/2002 16:38:15 - Entered into WritingThread()
30/04/2002 16:38:16 - Slept for one second...
```

Trace 类提供了一种用于缩进跟踪消息的属性 IndentLevel。例如，可以对由主线程和辅线程写入的跟踪信息使用不同的缩进级别。为此，可以把下面的命令行添加到上面的代码中：

```
private static void WritingThread()
{
  // Setting indent level
  Trace.IndentLevel = 2 ;

  // Trace an info message
  Trace.WriteLine(DateTime.Now + " - Entered WritingThreadf()");

  // Sleeping for one sec....
  Thread.CurrentThread.Sleep(1000) ;

  // Trace an info message
  Trace.WriteLine(DateTime.Now + " - Slept for 1 second...");
}
```

修改后的代码输出的内容是：

```
30/04/2002 16:40:07 - Entered Main()
        30/04/2002 16:40:07 - Entered into WritingThread()
        30/04/2002 16:40:08 - Slept for one second...
```

注意：

可以分别使用 Indent()和 Unindent()方法来增加或减少缩进级别。

## 6.3.3　跟踪选项

当接近应用程序的部署阶段时，需要从代码中删除所有的跟踪和调试消息。但是，

不必寻找每个跟踪指令并删除它，而可以在应用程序构建期间使用编译标记。在 Visual Studio .NET IDE 中，右击 Solution Explorer 窗口中的项目名称，从弹出的菜单中选择 Properties 项，会打开如图 6-13 所示的对话框。

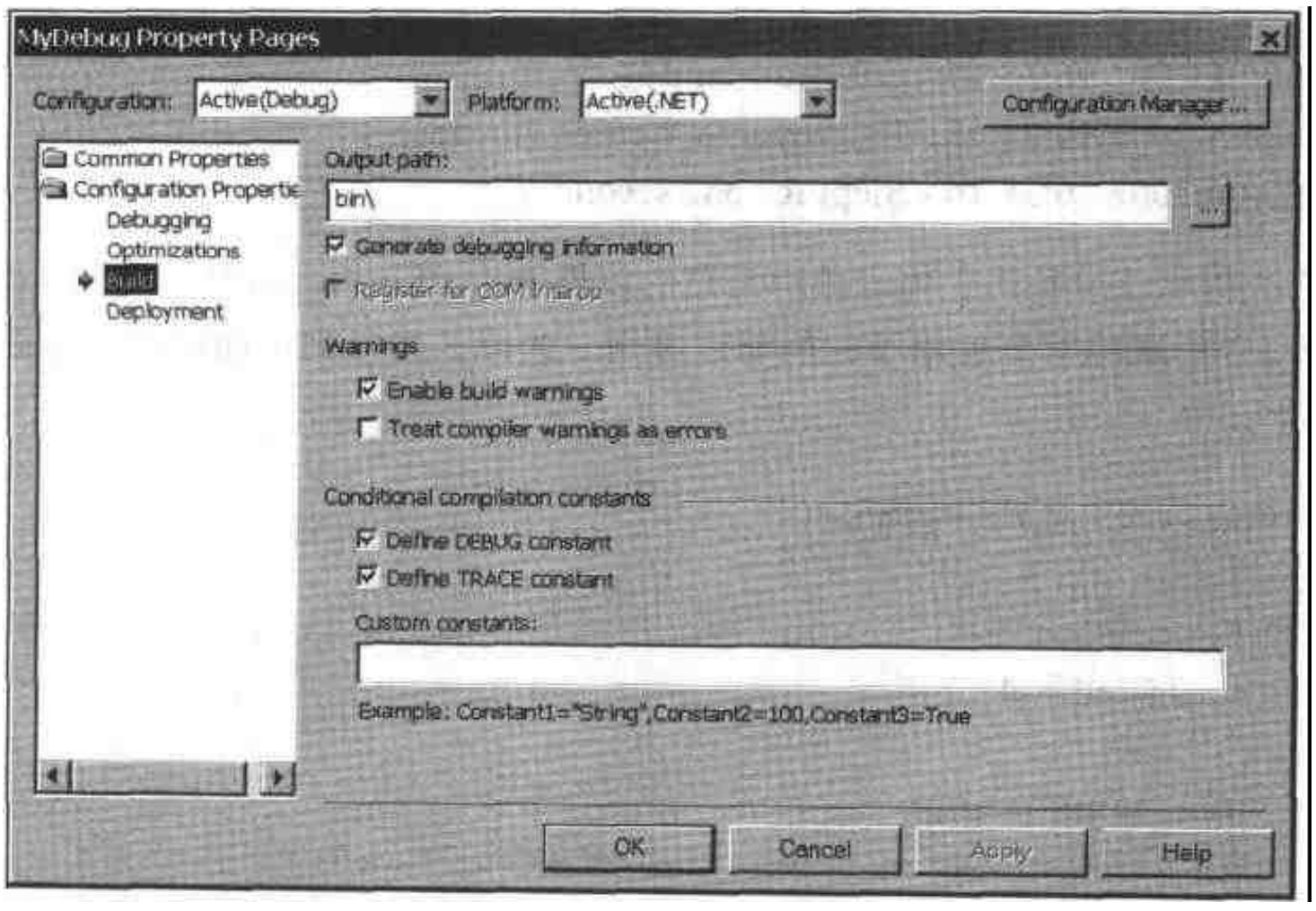


图　6-13

取消对 Define DEBUG constant 和 Define TRACE constant 复选框的选定，重新编译解决方案，所有的跟踪和调试语句就会从应用程序中清除掉。

**注意：**

要删除跟踪功能，甚至可以使用 csc.exe 命令行编译器。当编译时，只须使用 /d:TRACE=FALSE /d:DEBUG=FALSE 选项即可完成该操作。

· 在被跟踪的代码中添加选项，可以在运行期间激活和禁用跟踪消息功能。在应用程序的配置文件中声明一个值，就可以激活跟踪功能，且无需重建整个解决方案。当然，必须构建应用程序来维护跟踪信息，这就意味着应用程序最后的尺寸会较大，这样会降低应用程序的性能，即使关闭了选项也是如此。

.NET Framework 提供了 BooleanSwitch 和 TraceSwitch 类来实现这些选项，首先分析一下 BooleanSwitch 类。

### 1. BooleanSwitch 类

在被跟踪的代码中使用这个类，只须修改应用程序配置文件中的一个值，就可以决定是激活还是禁用跟踪消息功能。WriteLineIf()和 WriteIf()方法可根据 BooleanSwitch 类提供的 Enabled 属性来写入消息。要在应用程序中添加选项，须遵循以下步骤：

(1) 添加一个应用程序配置文件，这可以手动生成，也可以从 Visual Studio .NET 内的 Project 菜单中选择 Add New Item...命令，并从对话框中选择 Text File 模板，选择 App.config 文件名。

(2) 打开配置文件，插入必要的 XML 标记，告诉应用程序选项的名称和值。指定值等于 0 时禁用跟踪功能，值为 1 时激活跟踪功能。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="MySwitch" value="1" />
    </switches>
  </system.diagnostics>
</configuration>
```

(3) 在代码中创建一个新的 BooleanSwitch 对象，该对象的名称与在配置文件中指定的名称相同。也可以使用与 Trace 静态方法相关联的 Enabled 属性。下面继续介绍 Debugging 示例，声明一个可以在代码中任何地方使用的全局 BooleanSwitch 对象：

```
BooleanSwitch bs;

static void Main(string[]args)
{
  // Create a Boolean switch called MySwitch
  bs = new BooleanSwitch("MySwitch",
    "Enable/Disable tracing functionalities");

  // Create a file listener
  FileStream fs = new FileStream("C:\Debugging.log",
      FileMode.OpenOrCreate) ;
  Trace.Listeners.Add(new TextWriterTraceListener(fs)) ;

  // Write the line only when the switch is on
  Trace.WriteLineIf(bs.Enabled, DateTime.Now +
  " - Entered in Main()") ;

  ...
}
```

在 Main()方法中创建这个对象，指定其使用配置文件中的名称，并添加一个简要描

述。只有在配置文件中将 Enabled 属性设置为 1，WriteLineIf()方法才会写入消息。

### 2. TraceSwitch 类

该类是 BooleanSwitch 类的一个增强版本，因为它允许选择是禁用跟踪功能，还是使用基于重要性的层次结构来显示消息。跟踪级别如表 6-4 所示。

表 6-4

| 跟踪级别 | 说　明 |
|---|---|
| 0 | None：禁用跟踪功能 |
| 1 | TraceError: 只有错误消息会写入到侦听器应用程序中 |
| 2 | TraceWarning: 错误和警告消息会写入到侦听器应用程序中 |
| 3 | TraceInformation: 错误、警告和信息消息会写入到侦听器应用程序中 |
| 4 | TraceVerbose: 所有类型的消息都会写入到侦听器应用程序中 |

当出现错误时，可以修改应用程序的配置文件，指定只写入已添加到代码中的错误消息，从而将注意力集中到这些消息上。该配置文件与 BooleanSwitch 示例中的配置文件相同，只是修改了代码，因为必须从 TraceSwitch 类中实例化对象。此外，本例还使用了该类中的枚举值来指定跟踪消息的级别。下面是这个例子 TraceSwitchExample.cs：

```
using System;
using System.Threading;
using System.IO;
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;

namespace TraceSwitchExample
{
  class Class1
  {
    private static TraceSwitch ts;

    [STAThread]
    static void Main(string[] args)
    {
      // Create a boolean switch called MySwitch
      ts = new TraceSwitch("MySwitch", "Four different trace levels");
      // Create a file listener
```

```
FileStream fs = new FileStream(@"C:\Debugging.log",
    FileMode.OpenOrCreate) ;
Trace.Listeners.Add(new TextWriterTraceListener(fs)) ;

// Write the line only when the switch is set
// to TraceInformation level or above.
Trace.WriteLineIf(ts.TraceInfo, DateTime.Now +
    " - Entered in Main()") ;

// Create a thread
Thread t;
t = new Thread(new ThreadStart(DBThread));

// Start the thread
t.Start() ;

// Wait for the user carriage return
Console.Read() ;

// Close the file listener flushing the trace messages
Trace.Close () ;
}
```

首先声明一个全局 TraceSwitch 对象，然后创建一个新的对象，并以配置文件中指定的名称命名该对象。向应用程序中添加一个文本文件日志侦听器。然后，启动与 SQL Server 内部的 pubs 数据库相联系的新线程，以检索 authors 表中的所有记录。

如果忽略了该线程，Open()方法就会引发一个异常，生成一个跟踪错误消息：

```
private static void DBThread()
{
  // Trace an info message
  Trace.WriteLineIf(ts.TraceInfo, DateTime.Now +
          " - Entered in DBThread()") ;

  // Create a connection object
  SqlConnection dbConn = new
      SqlConnection("servers.;database=pubs;uid=sa;pwd=") ;

  // Create a command object to execute a SQL statement
  SqlCommand dbComm = new SqlCommand("SELECT * FROM authors", dbConn) ;
```

```
SqlDataReader dr = null;

try
{
   Trace.WriteLineIf(ts.TraceInfo, DateTime.Now +
         " - Executing SQL statement");
   // Execute the SQL statement
   dr = dbComm.ExecuteReader(CommandBehavior.CloseConnection);

   while (dr. Read ())
   {
      // Reading records
   }
}
catch (Exception ex)
{
   Trace.WriteLineIf(ts.TraceError, DateTime.Now +
         " - Error: " + ex.Message) ;
}
finally
{
   if ((dr.IsClosed == false) && (dr !=null))
      dr.Close() ;
}
}
```

下面是当配置文件中指定的值为 1(即指定为 TraceError)时代码的输出：

19/04/2002 17:52:23 - Error: ExecuteReader requires an open and available Connection. The connection's current state is Closed.
Slept for 1 second...

以下是当配置文件中指定的值为 3(即指定为 TraceInformation)时代码的输出：

19/04/2002 17:54:23 - Entered Main()
19/04/2002 17:54:23 - Entered DBThread()
19/04/2002 17:54:24 - Executing SQL statement
19/04/2002 17:54:24 - Error: ExecuteReader requires an open and available Connection. The connection's current state is Closed.

### 6.3.4　Debug 类

Debug 类提供了与 Trace 类相同的功能。它提供了相同的方法和属性，以及相同的跟踪结果。

**注意：**

使用 Trace 类提供的 Listeners 集合修改侦听器应用程序时，也会修改调试消息的侦听器应用程序。

这两个类的最大不同在于双方所适用的语境不同。Debug 类适用于在调试会话期间添加信息。而在部署应用程序之前，我们将构建应用程序的发布版本，该版本会自动从代码中删除调试信息。因此，在运行期间需要检查应用程序时，就须添加 Trace 类的功能。

## 6.4　DataImport 示例

本节将重点介绍一个实际的例子，以演示前文所讲述的内容。这里的 DataImport 示例是一个典型的应用程序，它会在指定的目录中等待文件，之后把文件导入 SQL Server 数据库。该应用程序的代码以及本书中其他代码都可以在 Wrox 公司的 Web 站点找到。下面描述一下该示例将使用的类：

- FileSystemWatcher：这个类允许开发人员指定要监控的目录，当发生变化时(例如创建或删除了文件)就引发一个事件。该类包含在.NET Framework 类库的 System.IO 命名空间中。
- TextWriterTraceListener：该类实现用户自己的跟踪功能。
- Thread：这个类在前面的内容中已经介绍了多次，它允许启动一个新的线程，将数据导入到数据库中。
- SqlClient 命名空间中管理数据库连接和更新所必需的很多类。

DataImport 应用程序的第一个版本包含一些逻辑错误，使用跟踪功能就能发现它们。这样可以了解日志(跟踪)文件和它们的重要性。

**注意：**

要了解 ADO.NET 类的更多内容，请参考 Wrox 公司出版的 Professional ADO.NET Programming(ISBN 1-861005-27-X) 或 ADO.NET Programmer's Reference (ISBN 1-861005-58-X)。

## 6.4.1　代码

现在开始分析 DataImport 示例的代码：

```
using System;
using System.IO;
using System.Data;
using System.Data.SqlClient;
using System.Threading;
using System.Diagnostics;
```

```
namespace DataImport1
{
  class DataImport
  {
```

首先，为使用 FileSystemWatcher、Thread 和 SQL Server 类，添加所有必要的命名空间：

```
public static BooleanSwitch bs;

[STAThread]
static void Main(string[] args)
{
  // Remove the default listener
  Trace.Listeners.RemoveAt(0) ;

  // Create and add the new listener
  bs = new BooleanSwitch("DISwitch", "DataImport switch");
  Trace.Listeners.Add(new TextWriterTraceListener(
    new FileStream(@"C:\DataImport.log", FileMode.OpenOrCreate)));
```

接着，代码删除了默认侦听器，并创建了一个新的指向 C:\DataImport.log 的 TextWriterTraceListener 对象：

```
// Create a FileSystemWatcher object used to monitor
// the specified directory
FileSystemWatcher fsw = new FileSystemWatcher();

// Set the path to watch and specify the file
```

```
// extension to monitor for
fsw.Path =@"C:\temp";
fsw.Filter = "*.xml";

// No need to go into subdirs
fsw.IncludeSubdirectories = false;

// Add the handler to manage the raised event
// when a new file is created
fsw.Created += new FileSystemEventHandler(OnFileCreated) ;
// Enable the object to raise the event
fsw.EnableRaisingEvents = true;
```

在上面的代码中，创建了一个 FileSystemWatcher 对象，用来监控在 Path 属性中指定的 C:\temp 目录。Filter 属性用来筛选包含在该目录中的每个文件，寻找具有指定的文件扩展名的文件。IncludeSubdirectories 属性确定是否将文件监控延伸到子目录。要接收文件的创建事件，就必须指定由 FileSystemWatcher 类提供的 Created 事件。使用 FileSystemEventHandler，可以指定一个在目标目录中创建新 XML 文件时调用的事件处理程序。最后，代码通过 FileSystemWatcher 对象来触发事件。

```
try
{
  // Call the waitforchanged method in
  // an infinite loop. When the event is raised
  // the OnFileCreated will be contacted.
  WaitForChangedResult res;
  while (true)
  {
    res = fsw.WaitForChanged(WatcherChangeTypes.Created) ;
    Trace.WriteLineIf(bs.Enabled, DateTime.Now +
          " - Found: " + res.Name + " file");
  }
}
```

以上的代码实现了一个等待文件创建事件触发的无限循环。WaitForChangedResult 对象包含有关文件创建的信息。例如，代码使用 Name 属性来跟踪被发现的文件名称。

```
catch (Exception e)
{
  Trace.WriteLineIf(bs.Enabled, DateTime.Now +
```

```
            " - An exception occurred while waiting for file: ");
        Trace .Indent() ;
        Trace.WriteLineIf(bs.Enabled, DateTime.Now + " - " + e.ToString()) ;
        Trace.Unindent() ;
    }
    finally                                                                    "
    {
        fsw.EnableRaisingEvents = false;
        Trace.WriteLineIf(bs.Enabled, DateTime.Now +
                " - Directory monitoring stopped");
        Trace.Close() ;
    }
```

上面的 Main()方法以跟踪有用的消息和异常作为结束。OnFileCreated()静态方法的代码如下：

```
private static void OnFileCreated(Object source,
                                  FileSystemEventArgs e)
{
    try
    {
        // Create a new object from the
        // ImportData class to process the
        // incoming file
        ImportData id = new ImportData() ;
        id.m_strFileName = e.FullPath;

        // Create and start the thread
        Thread t = new Thread(new ThreadStart(id.Import));
        t.Name = "DataImportThread";
        t.Start();
```

在 OnFileCreated 事件处理程序内部，将启动一个新的线程。这个线程将使用定制 ImportData 类中的 Import 方法，将 XML 文件导入到数据库中。由于此时我们知道被发现文件的完整路径(在 FileSystemEventArgs 参数中包含该信息)，而且在 ImportData 类中需要它，所以可以使用由该类提供的 m_strFileName 变量。

```
    catch
    {
        Trace.WriteLineIf(bs.Enabled, DateTime.Now +
            " - An exception occurred while queuing file : ") ;
```

```
    Trace.Indent() ;
    Trace.WriteLineIf(bs.Enabled, DateTime.Now + " - " + e.ToString());
    Trace.Unindent() ;
}
finally
{
    Trace.Flush() ;
}

class ImportData
{
    // Path and filename of the retrieved file
    public string m_strFileName = null;

    public void Import()
    {
        // Declare Sql objects to contact the database
        SqlConnection dbConn = new
            SqlConnection("server=.;database=pubs;uid=sa;pwd=") ;
        SqlDataAdapter da = new SqlDataAdapter(
                "SELECT * FROM authors", dbConn);
        DataSet ds = new DataSet() ;

        SqlCommandBuilder sa = new SqlCommandBuilder(da) ;
```

在 Import()方法中，首先创建和设置所有必要的类来连接 SQL Server pubs 数据库中的 authors 表。SqlConnection 对象允许指定数据库连接参数。SqlDataAdapter 对象使用连接对象连接到数据库，以执行指定为第一个参数的 SQL 语句。最后，SqlCommandBuilder 检查在 SqlDataAdapter 构造函数中指定的 SQL 语句，并自动地创建 INSERT, MODIFY 和 DELETE 语句。当使用 SqlDataAdapter 类提供的 Update()方法用新的信息物理修改数据库时，就需要使用它们。

```
        try
        {
            Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
                " - Filling the DataSet.");
            // Fill a dataset with data within the authors table
            da.Fill(ds) ;
```

在这里，SqlDataAdapter 类中的 Fill()方法把先前指定的 SQL 查询结果填充到它的参数指定的 DataSet 对象中。DataSet 是数据库数据在内存中的表示，因此它会像 authors

表一样被格式化，并用 authors 表中的记录来填充。

```
// Read the XML file filling another dataset
DataSet dsMerge = new DataSet() ;

Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
    " - Reading XML file.");
dsMerge.ReadXml(m_strFileName, XmlReadMode.InferSchema) ;
Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
    " - DataSet filled with data.");
```

这里，代码使用被发现的文件来填充另一个 DataSet 对象。这次使用了 ReadXml() 方法。DataSet 对象的功能非常适合于本例，可以用完全相同的方式来管理由数据库和 XML 文档提供的数据。DataSet 对象在内部保存了记录的 XML 数据表示。

```
// Update the database tracing the
// total time needed to conclude the operation
DateTime time;
time = DateTime.Now;
Trace.WriteLineIf(DataImport.bs.Enabled, time +
    " - Updating database.");
da.Update(dsMerge) ;
DateTime time2;
time2 = DateTime.Now;
Trace.WriteLineIf(DataImport.bs.Enabled, time2 +
        " - Database updated successfully.");
Trace.Indent() ;
Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
    " - Total TIME: " + time2.Subtract(time) + " second/s");
Trace.Unindent() ;
```

最后，代码使用 SqlDataAdapter 类提供的 Update()方法向 authors 表中写入新的记录。注意在本段代码中所使用的跟踪信息；通过添加性能消息提供详细的信息。DateTime 类用于检索更新数据库所需要的全部时间(以秒为单位)：

```
catch (SqlException sex)
{
  Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
      " - A SQL exception occurred during file processing: ") ;
  Trace.Indent() ;
  Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now
```

```
            + " - " + sex.ToString());
    Trace.Unindent() ;
}
catch (Exception ex)
{
    Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
        " - A general exception occurred during file processing: ");
    Trace.Indent() ;
    Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
        " - " + ex.ToString()) ;
    Trace.Unindent() ;
}
finally
{
    Trace.Flush() ;
}
```

然后，在编写用于捕获和处理可能发生的异常代码后，所有的代码就完成了。

## 6.4.2　测试应用程序

要测试应用程序，须遵循以下步骤：

- 创建 C:\temp 目录来保存 XML 文件
- 运行 DataImport 应用程序
- 将 authors.xml 文件复制到 C:\temp 目录中

最终结果是 C:\目录有一个 DataImport.log 文件，其内容如下：

```
01/05/2002 12:23:01 - Found: authors.xml file
01/05/2002 12:23:01 - Filling the DataSet.
01/05/2002 12:23:02 - Reading XML file.
01/05/2002 12:23:02 - DataSet filled with data.
01/05/2002 12:23:02 - Updating database.
01/05/2002 12:23:02 - Database updated successfully.
01/05/2002 12:23:03 - Total TIME: 0 second/s
```

authors.xml 文件不是很大，所以全部时间不足 1 秒。

## 6.4.3　逻辑错误

所有的一切似乎都运行良好，但是显然，并不是每件事情都处理好了。前面只使用了非常小的文件来测试应用程序，所以当应用程序收到文件创建事件并打开文件时，把文件复制到目录的进程就完成了关闭文件的任务。当接收到一个很大的文件时，将会发生什么事情呢？当线程尝试访问 XML 文件，并填充 DataSet 对象时，它会接收到一个由于尝试打开已由复制任务使用的文件而引起的拒绝访问错误。下面复制 huge_authors.xml 文件，再次测试应用程序。由于我们已经使用了跟踪消息，所以在日志文件中会发现下列错误：

```
4/14/2002 1:29:00 PM - Found: huge_authors.xml file
4/14/2002 1:29:00 PM - Filling the DataSet.
4/14/2002 1:29:00 PM - Reading XML file.
4/14/2002 1:29:00 PM - A general exception occurred during file processing:
    4/14/2002 1:29:00 PM - System.IO.IOException: The process cannot access the file
"C:\temp\huge_authors.xml" because it is being used by another process.
   at System.IO.__Error.WinIOError(Int32 errorCode, String str)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare
share, Int32 bufferSize, Boolean useAsync, String msgPath, Boolean bFromProxy)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare
share)
   at System.Xml.XmlDownloadManager.GetStream(Uri uri, ICredentials credentials)
   at System.Xml.XmlUrlResolver.GetEntity(Uri absoluteUri, String role, Type
ofObjectToReturn)
   at System.Xml.XmlTextReader.CreateScanner()
   at System.Xml.XmlTextReader.Init()
   at System.Xml.XmlTextReader.Read()
   at System.Xml.XmlReader.MoveToContent()
   at System.Data.DataSet.ReadXml(XmlReader reader, XmlReadMode mode)
   at System.Data.DataSet.ReadXml(String fileName, XmlReadMode mode)
```

这是一种调试器常常捕获不到的错误，因为用来启动调试器的时间和单步执行代码的时间常常足以复制文件了。或许在用户的机器上不会发生这种情况，这取决于硬盘访问速度和内存数量(即应用程序减缓的程度)。

错误消息提出了一种可能的解决方案，即应在应用程序中添加解决错误的方法。在调用 ReadXml()方法之前，应当尝试用独占访问方式打开文件。如果发生了错误，可以将线程挂起片刻，当能够处理文件时，再次进行尝试。在添加了 GetFileAccess()方法后，DataImport2 中的代码如下所示：

```
private bool GetFileAccess()
{
   Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
         " - Trying to get exclusive access to the "
         + m_strFileName + " file.");

   try
   {
      FileStream fs = File.Open(m_strFileName,
                                FileMode.Append,
                                FileAccess.Write,
                                FileShare.None) ;
      fs.Close () ;
      Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
            " - Access to the " + m_strFileName + " file allowed.");
      return true;
   }
   catch
   {
      Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
            " - Access denied to the " + m_strFileName + " file.");
      return false;
   }
}
```

添加 GetFileAccess()函数会返回 一个布尔值，表示能否获得对文件的独占访问权。该方法在打开文件时，把共享访问属性设置为 None：

```
public void Import()
{
// Declare Sql objects to contact the database
SqlConnection dbConn = new
      SqlConnection("servers.;database=pubs;uid=sa;pwd=") ;
SqlDataAdapter da = new SqlDataAdapter(
      "SELECT * FROM authors", dbConn);
DataSet ds = new DataSet() ;

SqlCommandBuilder sa = new SqlCommandBuilder(da) ;
try
{
```

```
while (GetFileAccess()==false)
{
   Thread.Sleep(5000) ;
   Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
         " - Slept 5 seconds... Try to access to the "
         + m_strPileName + " file, again.");
}

Trace.WriteLineIf(DataImport.bs.Enabled, DateTime.Now +
      " - Filling the DataSet.");
// Fill a dataset with data within the
// authors table
da.Fill(ds) ;
```

ImportData 类提供的 Import()方法将尝试获得对文件的独占访问权。如果文件仍然是由复制任务打开，线程将挂起 5 秒钟。然后调用 GetFileAccess()方法，直到能够打开源文件为止。

现在我们已经清楚了跟踪功能对于了解应用程序在运行期间的行为的重要性。

## 6.5　本章小结

本章介绍了如何使用 Visual Studio .NET 调试器在应用程序执行期间观察其行为。另外，还介绍了调试器提供的强大工具，以用于查看和修改变量的值等。

在本章的第二部分，论述了.NET 提供的跟踪功能及 3 个跟踪类：Trace, Debug 和 Switch。列举了几种最有用的跟踪功能，并集中论述了如何通过修改应用程序配置文件中的值来启动跟踪功能。

最后，列举了一个跟踪技术帮助开发人员查找和改正故障及逻辑错误的实际示例。

# 第7章　联网与线程

本书的前几章深入介绍了 C#和.NET 中的线程，并论述了与编写多线程应用程序有关的各种概念和技术。现在您已经对线程有了充分的了解，本章将利用 C#构建一个简单的多线程客户机/服务器应用程序，并使用一些已介绍过的概念。

有一些应用程序必须高效地使用线程和异步编程，例如网络通信、有效的用户接口和磁盘输入\输出等等。在所有这些情况下，单线程应用程序在等待一个操作完成时，会冻结或者看起来已经崩溃。网络应用程序也是这样，因为其反应时间通常是衡量应用程序优劣最重要的标准，尤其是对于那些连接速度较低的用户来说就更是如此。在本章列举的示例应用程序中，我们将使用 System.Net 命名空间，并简要探究.NET 的联网能力，其主要原因在于网络应用程序的多用户和异步本性使它们成为使用线程的理想环境。本章的主要内容如下：

- 使用 System.Net 命名空间在.NET 中开发网络应用程序
- 开发基于 TCP/IP 协议的、简单的多线程客户机-服务器应用程序
- 使用.NET 固有的功能实现异步操作
- 在客户机和远程服务器之间使用异步消息传输

## 7.1　.NET 中的联网

在.NET Framework 出现以前，开发复杂的基于 Windows 的网络应用程序是一件比较难的事情，具有这种开发能力的人很大程度上局限于使用令人费解的 WinSock 库的高级 C++程序员。当然，Visual Basic 开发人员或许可以利用 WinInet 控件来完成相对简单的任务。其他语言可以使用其他控件来完成。然而，由于控件所提供的简单而有限的服务不支持许多功能，所以开发人员不必作过多的尝试。

.NET Framework 中的 System.Net 命名空间提供了一个有效功能，该功能包装在一个简单而坚固的对象模型中。与打包和抽象化在 System.Net 命名空间中的几乎所有 WinSock 2.0 核心功能一样，这些类使用简单，功能全面。开发人员可以轻松地在从套接字到 HTTP 的任何层上进行开发。并且，与对 WinSock 库的难于使用不同，System.Net 命名空间为开发人员卸下了手工编码诸多必要的资源管理任务的负担。比如，处理重叠 IO 和完成端口。

下面就简要浏览一下 System.Net 命名空间。

## 7.1.1 System.Net 命名空间

System.Net命名空间实际上由System.Net和System.Net.Sockets两个命名空间组成。

在应用程序中，我们主要使用 System.Net.Sockets 命名空间。System.Net 类的分层方法允许应用程序根据需要，利用各种控件来访问网络。除了对套接字的广泛支持以外，System.Net 类还提供了一组和 HTTP 协议一起使用的功能。System.Net 所提供的大部分功能分为 3 个层：应用程序协议、传输协议和 Web 协议。System.Net.Sockets 命名空间基本上由处理传输协议的类和实用程序组成。下面了解一下 System.Net 命名空间中比较重要的类，如表 7-1 所示。

表 7-1

| 类 | 说　明 |
| --- | --- |
| Authorization | 为 Web 服务器提供身份验证消息 |
| Cookie | 提供一套用于管理 cookies 的属性和方法。该类不能被继承 |
| Dns | 简单的域名解析功能 |
| EndPoint | 标识网络地址。这是一个抽象类 |
| GlobalProxySelection | 所有 HTTP 请求的全局默认代理实例 |
| HttpVersion | 定义由HttpWebRequest 和HttpWebResponse 类支持的HTTP 版本号 |
| HttpWebRequest | WebRequest 类的特定于 HTTP 的实现 |
| HttpWebResponse | WebResponse 类的特定于 HTTP 的实现 |
| IPAddress | 网际协议 (IP)地址 |
| IPEndPoint | 由一个 IP 地址和一个端口号组成的网络端点 |
| IPHostEntry | Internet 主机地址信息容器类 |
| NetworkCredential | 为基于密码的身份验证方案(如 basic, digest, NTLM 和 Kerberos 身份验证)提供证书 |
| SocketAddress | 存储来自 EndPoint 派生类的串行化信息 |
| SocketPermission | 建立和接受套接字连接的控制权 |
| WebClient | 为收发来自于使用 URI 标识的资源的数据提供通用方法 |
| WebException | 当通过 HTTP 协议访问资源时，一旦出现错误，就抛出异常 |

(续表)

| 类 | 说　明 |
| --- | --- |
| WebPermission | 访问 HTTP Internet 资源的控制权 |
| WebPermissionAttribute | 为访问 Internet 资源指定权限 |
| WebProxy | 包含了 WebRequest 类的 HTTP 代理设置 |
| WebRequest | 请求一个统一资源标识符 (URI)。该类是抽象类 |
| WebResponse | 提供来自于 URI 的响应。该类是抽象类 |

System.Net 命名空间包含了丰富的类和实用程序，这对于 Web 和网络编程是非常有用的。

## 7.1.2　System.Net.Sockets 命名空间

System.Net.Sockets 命名空间主要用于传输层：套接字层，该层包含了一组类。这些类不仅提供了强大、高效率的套接字堆栈(附属于 Berkeley 套接字)，还卓有成效地降低了套接字编程的复杂性。最后，对 TCP 和 UDP 的内置支持能很好地集成在 System.Net.Sockets 的类中。表 7-2 列出了 System.Net.Sockets 命名空间中的类。

表 7-2

| 类 | 说　明 |
| --- | --- |
| LingerOption | 包含在未定数据出现之后，套接字将保持可用的时间信息(套接字的拖延时间) |
| MulticastOption | 包含 IP 多播数据包的 IP 地址值 |
| NetworkStream | 提供网络访问的底层数据流 |
| Socket | 实现 Berkeley 套接字接口 |
| SocketException | 当套接字出现错误时，系统就会抛出异常 |
| TcpClient | 为 TCP 网络服务提供客户端连接 |
| TcpListener | 侦听来自 TCP 网络客户端的连接。其实质上是 TCP 服务器类 |
| UdpClient | 提供用户数据报协议 (UDP) 网络服务 |

给开发人员提供不同级别的控制，例如像 Socket 类的低层类和像 TcpClient 类的高层类，TcpClient 类可提高生产率，但控制权略有减少。深入论述这些类超出了本书的范围，详细内容可参阅《.NET 网络高级编程》(清华大学出版社引进并已出版)一书。在本章的后面，当设计和开发示例应用程序时，将密切关注上述一些类。

## 7.2　创建示例应用程序

简要介绍了.NET 中的网络编程之后，下面开始论述将在本章中构建的应用程序。这个示例的目的在于创建一个简单的应用程序，让用户熟悉在.NET 中构建联网应用程序过程中线程的使用。该应用程序由两个小的 Windows Form 应用程序组成，其中一个充当服务器程序，另一个充当客户端程序。我们将使用 Visual Studio .NET 设计和实现这些应用程序。

### 7.2.1　设计目标

我们要创建两个相互交互的应用程序。第一个应用程序是一个多线程/多用户的股票报价服务器程序，该程序能够从数据库表中查找股票报价，并将数据异步发回给请求的客户。第二个应用程序是客户端程序，它通过一个股票符号来查询服务器，得到需要的报价信息。所有这些都将异步发生，这样当服务器对请求进行响应时，客户端程序的用户界面不会暂停。

.NET Framework 中有许多方法用于处理异步操作。利用这些方法，就不需要显式生成和管理必须的线程了。

下面概括和总结了构建应用程序时，将要遵守的基本要求：

- 存在两个能在 Internet 上相互通信的独立的应用程序(一个充当服务器程序，另一个为客户端程序)
- 当向服务器查询股票报价时，客户端的用户界面不会由于低速的网络连接或其他的延迟而暂停或停滞
- 服务器可以处理同时发生的众多客户机连接和查询，并可以用异步方式与客户机通信
- 网络设置必须基于应用程序，并可更改

为了理解应用程序中的一般用户交互，下面看一个简单的 UML 顺序图，如图 7-1 所示。

前面从一个极高的层次论述了应用程序的基本设计原则。大多数开发人员现在或许已迫不及待地想了解该程序的代码了。现在就开始构建这两个应用程序，并讨论代码段和概念(与以往一样，代码可以从http://www.wrox.com站点获得)。

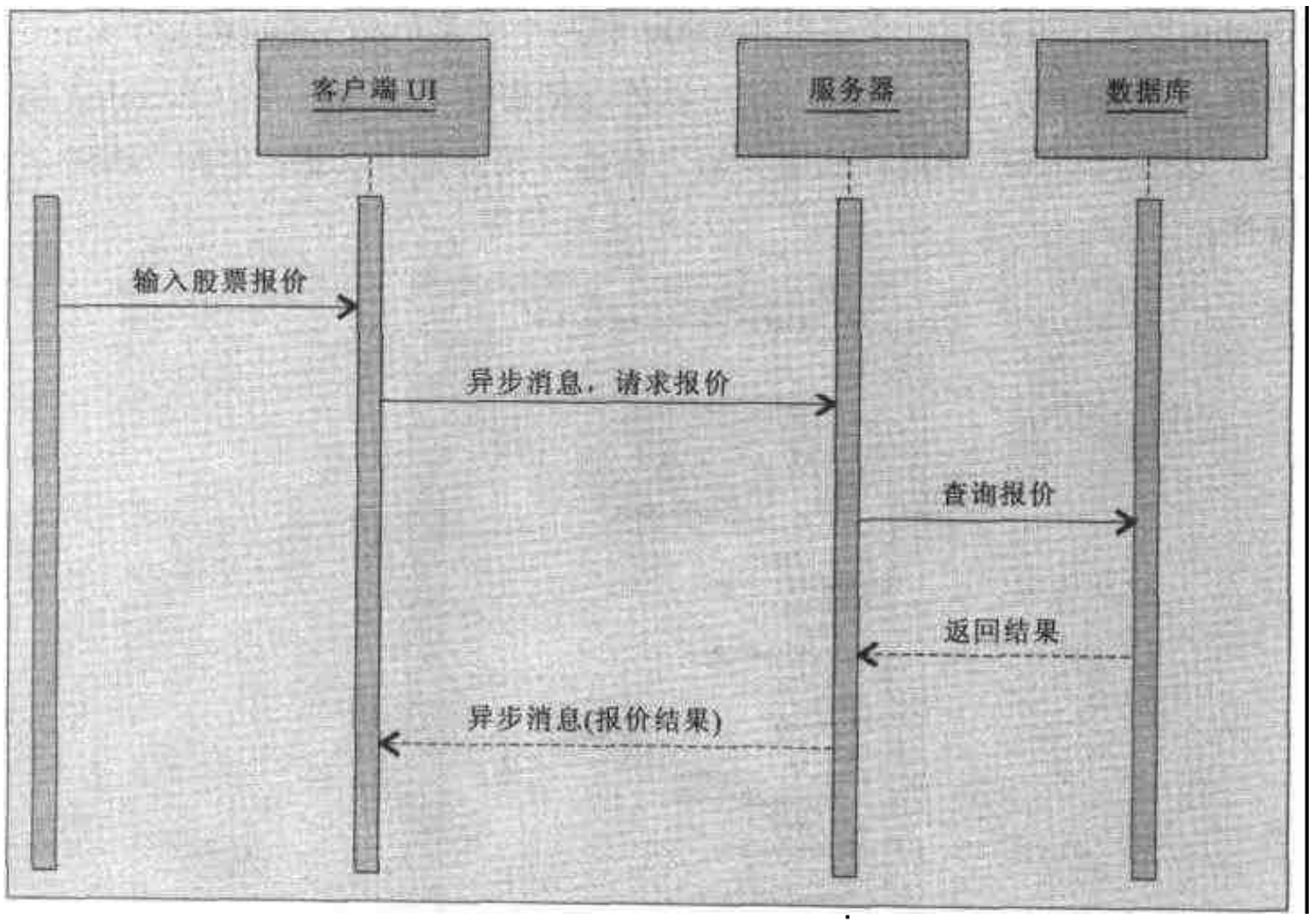


图　7-1

## 7.2.2　构建应用程序

如前所述，本章的示例应用程序包含两个独立的应用程序：一个客户端程序和一个服务器程序。这两个应用程序将通过一个特有的 TCP/IP 端口相互通信，这也可以通过更改应用程序的配置文件来实现(客户端程序和服务器程序需使用相同的配置文件)。现在就开始构建客户端应用程序，该应用程序能够执行向服务器查询股票报价的简单任务。

### 1. 创建客户程序

在开始创建该应用程序之前，先花一些时间来了解客户窗体类的 UML 视图，如图 7-2 所示，该视图包含了客户端应用程序的所有代码。

StockClient 应用程序只有一个 Windows 窗体，它包含了客户端应用程序的所有代码，例如，一些 private 成员变量和一些方法。要创建 StockClient 应用程序，首先要在 Visual Studio .NET 中创建一个新的 Windows 应用程序项目，并命名为 StockClient。在默认的窗体上，在页面上创建 3 个控件：一个 txtStock 文本框、一个 btnGetQuote 按钮，其 Text 属性设置为 Get Quote，以及一个来自于 Visual Studio .NET 工具箱的新 ListView 控件 lstQuotes。将窗体的 Name 和 Text 属性修改为 StockClient。接着，向窗体中添加一

个 MainMenu 控件，并创建一个菜单项&File 和两个子菜单&Connect(称为 mnuConnect)和 E&xit(称为 mnuExit)。最后，除了菜单以外，把窗体上所有控件的 Enabled 属性都设置为 False；这些属性在用户连接到服务器之前都将保持禁用状态，如图 7-3 所示为创建好的新窗体。

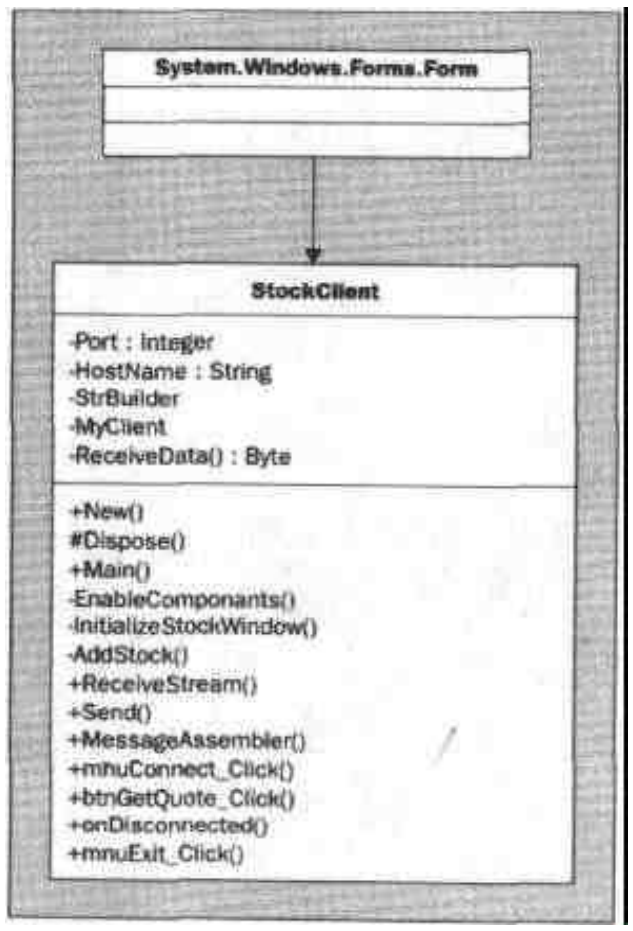


图　7-2

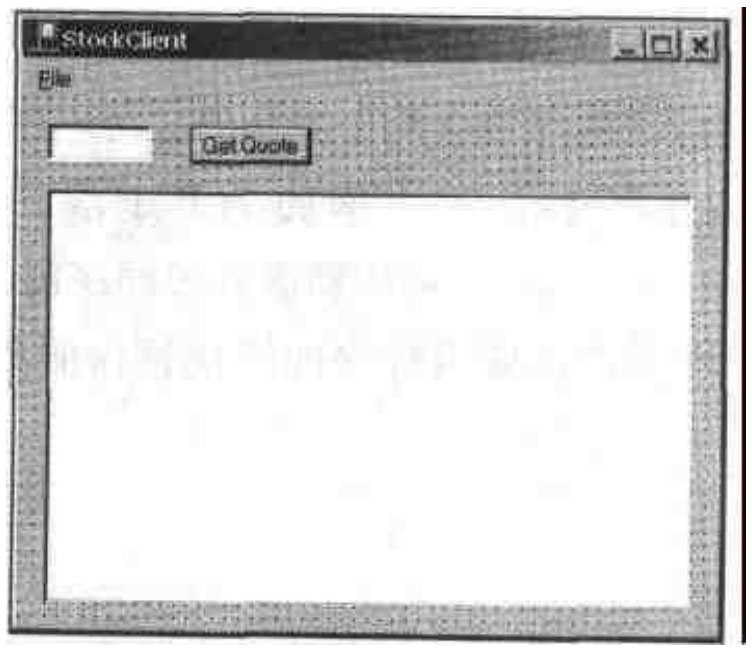


图　7-3

首先设置 using 指令来引用所需的命名空间：

```
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;
```

```
using System.Configuration;
```

我们需要一些 private 成员变量，这些变量将在整个 StockClient 应用程序中使用：

```
private int Port ;
private string HostName;
private const int packetSize = 1024;
private byte[ ] ReceiveData = new byte[packetSize];
private TcpClient MyClient;
private StringBuilder StrBuilder = new StringBuilder();
```

稍后将分析变量和它们的使用，下面改进 ListView 控件，使其跟踪用户输入的所有股票报价。该控件包含 6 个列：每一列对应了股票报价的每一个返回字段。这些字段分别是 Symbol、Price、Change、Bid、Ask 和 Volume。创建一个方法 InitializeStockWindow()，将这些列添加到 ListView 控件中，其代码如下所示：

```
private void InitializeStockWindow()
{
  lstQuotes.View = System.Windows.Forms.View.Details;
  lstQuotes.Columns.Add("Symbol", 60, HorizontalAlignment.Left);
  lstQuotes.Columns.Add("Price" , 50, HorizontalAlignment.Left);
  lstQuotes.Columns.Add("Change", 60, HorizontalAlignment.Left);
  lstQuotes.Columns.Add("Bid", 50, HorizontalAlignment.Left);
  lstQuotes.Columns.Add("Ask", 50, HorizontalAlignment.Left);
  lstQuotes.Columns.Add("Volume",170, HorizontalAlignment.Left) ;
}
```

上面的代码段除了给 ListView 控件分配了 6 个不同宽度的列以外，还启用了该控件的网格线。一旦连接到服务器上，应用程序准备开始从服务器中检索股票报价时，就会调用这个函数。我们还需要一个方法，该方法能根据需要来启用和禁用控件，例如在连接成功时启用控件，如下所示：

```
private void EnableComponents(bool enable )
{
  txtStock.Enabled = enable;
  btnGetQuote.Enabled = enable;
  lstQuotes.Enabled = enable;
}
```

现在创建一个简单的委托 DisconnectedHandler、一个事件 Disconnected 和一个处理该事件的事件处理程序 OnDisconnected()，一旦触发了该事件，就会调用该处理程序。

OnDisconnected()方法会启用 File 菜单中的 Connect 选项，并通过一个消息框显示错误消息。它同时禁用了窗体上的其余输入控件：

```
public delegate void DisconnectedHandler ( object sender ) ;
public event DisconnectedHandler Disconnected;
private void OnDisconnected(object sender)
{
  mnuConnect.Enabled = true;
  MessageBox.Show("The connection was lost!", "Disconnected",
                              MessageBoxButtons.OK, MessageBoxIcon.Error) ;
  EnableComponents(false);
}
```

要将 OnDisconnected()方法绑定到 Disconnected 事件上，需要使用如下代码：

```
Disconnected += new DisconnectedHandler(OnDisconnected) ;
```

Visual Basic .NET 的最佳特性之一是它可以在运行期间动态地为事件绑定及解除绑定事件处理程序。可以使用+=运算符将一个方法分配给一个事件，同样也可以使用-=运算符将事件处理方法与事件分离开来。当需要启动或停止事件的处理程序，或者需要重写处理程序的代码时，动态地为事件分配功能是非常有用的。在 Disconnected 事件中，可以将该事件分配给 OnDisconnected 委托。从技术上讲，可以在事件调用之前的任何时候来完成该操作。然而，通常最好在应用程序执行的早期声明所有的事件处理程序。所以，一旦与服务器建立连接，就在 mnuConnect_Click 事件中声明了事件处理程序。

至于 mnuConnect_Click 事件，双击 File 菜单中的 Connect 子项，进入连接服务器的代码。这里是开始网络编程的地方。首先，需要实例化 TcpClient 对象，它是 System.Net.Sockets 命名空间的一个成员。要实现该操作，需要一个主机地址和一个能使客户机与服务器通信的端口。将该信息存储在一个应用程序配置文件中，将它从应用程序的核心部分中提取出来。.NET 配置文件是结构完备的 XML 文件，并带有.NET Framework 的一个有用的命名空间，即 System.Configuration。下面看一下外部配置文件的内容，在记事本或 Visual Studio .NET 中很容易创建外部配置文件：

```
<configuration>
  <configSections>
    <section name="HostInfo"
             type="System.Configuration.SingleTagSectionHandler" />
  </configSections>
  <HostInfo hostname="localhost" port="6800" />
</configuration>
```

上面的 XML 文件包含一个条目，该条目的两个属性存储了主机信息。这里使用 localhost，端口可以是任何端口(只要它不是保留端口)，可以在 1024～65000 之间选择任何端口号。将该文件保存为 StockClient.exe.config(配置文件名是程序集后跟.config)，将其放到应用程序的编译版本所在的 bin 子目录中。

现在为 mnuConnect_Click 事件处理程序添加下列代码：

```
private void mnuConnect_Click (object sender,System.EventArgs e)
{
    IDictionary HostSettings;
    try
    {
        HostSettings =
            (IDictionary) ConfigurationSettings.GetConfig('HostInfo") ;
        HostName = (string)HostSettings["hostname"];
        Port = (int)HostSettings["port"];
        MyClient = new TcpClient(HostName, Port);
        MyClient.GetStream().BeginRead(receiveData, 0,
        packetSize,new AsyncCallback(ReceiveStream), null);
        EnableComponents(true) ;
        InitializeStockWindow() ;
        mnuConnect.Enabled = false;
        Disconnected += new DisconnectedHandler( OnDisconnected ) ;
    }
    catch
    {
        MessageBox.Show("Error: Unable to establish a connection!",
            "Disconnected", MessageBoxButtons.OK, MessageBoxIcon.Error);
        MyClient.Close();
    }
}
```

上述代码的第一部分从配置文件中读取主机信息。在这个类的开始部分，HostName、Port 和 MyClient 字段已被声明为 private。在此，在配置文件的所有 HostInfo 节点属性之中，声明了一个要读取的局部目录对象。

给构造函数传递 DNS 主机名和一个端口号，实例化 TcpClient 类的一个实例。主机名映射到网络上的一个特有主机(更准确地说是接口)；端口号则标识了要连接的主机上的特有服务。主机名和服务端口的结合通常称为端点，在.NET Framework 中其表示为 EndPoint 类。TcpClient 类的构造函数可以接受一个 IPEndPoint 类实例，也可以被重载以接受一个主机名和一个服务端口号。

**注意：**

可以使用 DNS 类将主机名解析为 IP 地址，接着使用服务端口来构造 IPEndPoint 类。

如果一切操作正确，服务器使用相同主机名和端口在运转，就建立了一个新的连接。一旦连接成功，就必须生成一个后台线程，从服务器中异步获取数据，从而为用户启用输入控件，以接收股票符号。至此，事情开始变得有趣了。

如前所述，应用程序的接收方法必须是异步的。这是客户机运转时没有延迟，可以连续地与用户交互的惟一方式。客户端应用程序在等待数据从服务器传来时，不应处于挂起状态。.NET Framework 提供的解决方法相对简单，且易于实现。首先，要标识 TcpClient 的 NetworkStream 对象。可以通过调用 TcpClient 对象实例中的 GetStream() 方法来实现这一操作，该方法返回用于发送和接收数据的底层 NetworkStream。GetStream()方法使用底层套接字作为其构造函数参数，创建一个 NetworkStream 类的实例。NetworkStream 类继承自 Stream 类，该类提供了很多实现网络通信的方法和属性。一旦有了底层数据流，就可以使用它在网络上发送和接收数据了。与其近亲类 FileStream 和 TextStream 相似，NetworkStream 类也提供了以同步方式发送和接收数据的读写方法。BeginRead()方法和 BeginWrite()方法仅仅是这些方法的异步版本。事实上，.NET Framework 类中大部分以 Begin 开头的方法，如 BeginRead()方法和 BeginGetResponse()方法，在本质上都是异步的，当它们通过委托使用时，不需要程序员提供额外的代码。所以，不必手工生成新的线程，当读取数据的进程运行在后台线程上时，应用程序的主线程就可以响应与 UI 的交互。下面是 BeginRead()方法的签名。

```
public override IAsyncResult BeginRead(
  byte[ ] buffer,
  int offset,
  int size,
  AsyncCallback callback,
  object state
);
```

表 7-3 解释了这个方法的参数。

表 7-3

| 参　数 | 说　明 |
| --- | --- |
| buffer | 数据要到达的 byte 数组数据缓冲区 |
| offset | 开始在 buffer 中存储数据的位置 |
| size | Buffer 的尺寸 |
| callback | 当异步调用完成时要调用的委托 |
| state | 包含由客户机提供的补充信息的对象 |

在继续本章内容之前，先简要论述一下异步调用，因为它是一个非常重要的概念。如前所述，同步操作的问题在于工作线程会暂停，直到一个特定的操作完成，而这并不总是很理想的。异步调用在后台线程中运行，就允许初始线程(调用线程)继续运行，因为.NET 利用委托允许在所有的类和方法上进行异步调用。然而，某些类，比如 NetworkStream 类，包含了内置异步功能的方法，如 BeginRead()方法。委托用于充当进行异步调用的函数的占位符。事实上，委托是类型安全的函数指针。

BeginRead()方法需要的是字节数组，而不是字符串或文本流，所以，它需要更多的处理。前面定义了一个 ReceiveData 变量和一个表示字节数组尺寸的整型常量 PacketSize。现在需要传入接收数据的方法名称——当数据到达时，回调委托将调用该方法。该方法将在后台线程中运行，如果希望与 UI(用户接口)交互，就务必小心。因此，当数据通过网络从服务器发送过来时，需要生成一个后台线程来接收数据，如下所示。

```
MyClient.GetStream() .BeginRead(receiveData, 0, packetSize,
   new AsyncCallback(ReceiveStream), null);
```

下面创建一个 ReceiveStream()方法，当数据到达时，该方法将处理字节数据包中的数据：

```
private void ReceiveStream( IAsyncResult ar )
{
   int ByteCount ;
   try
   {
      ByteCount = MyClient.GetStream().EndRead(ar);
      if(ByteCount < 1)
      {
         // MessageBox.Show("Disconnected")
         Disconnected(this) ;
         return;
      }
      MessageAssembler(receiveData, 0, ByteCount);
      MyClient.GetStream() .BeginRead(receiveData, 0,
           packetSize, new AsyncCallback(ReceiveStream), null);
   }
   catch(Exception ex)
   {
      // Display error message
      object [ ] Params =
```

```
                {("An error has occurred" + ex.ToString()).ToString()};
            Invoke( new InvokeDisplay(this.DisplayData), Params);
        }
    }
```

首先，必须检查字节数组数据包中是否有字节。该数据包中通常有一些数据。可以把这看作是连接的脉冲；只要客户机连接到服务器上，输入数据包中就会有某些数据，不过很少。使用 Stream 对象的 EndRead()方法来检查字节数组的当前尺寸。将 IAsyncResult 的一个实例传递给 EndRead()方法。GetStream()方法的 BeginRead()方法启动了对 ReceiveStream()方法的异步调用，并伴随着一连串隐藏的操作，这些操作是由编译器产生的，用于加速异步操作。接着，ReceiveStream()方法也会排队，等待线程池中的一个线程。如果 ReceiveStream()委托方法抛出一个异常，则新创建的 Async 线程就会终止，在调用者线程中再生成一个异常。图 7-4 进一步阐明了这种情形。

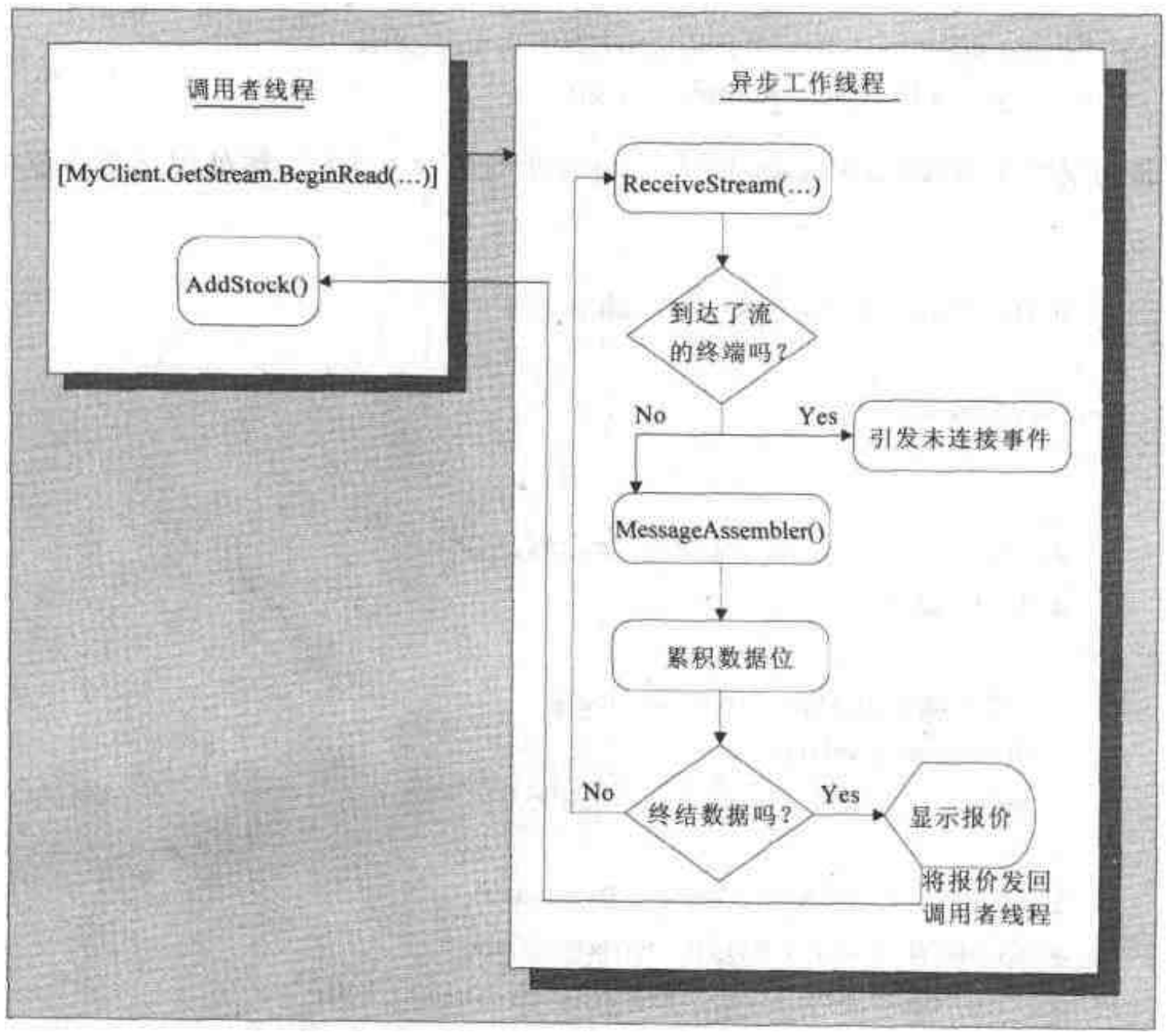


图　7-4

如果从 EndRead()方法返回的数小于 1，就表示连接已断开，于是可以引发 Disconnected 事件，完成处理该情形所要做的工作。然而，如果字节数组中的字节数大于 0，则可以开始接收输入数据了。这时，需要一个帮助方法利用从服务器中检索到的

数据构建一个字符串。

事实上，在.NET 中，可以用 BeginRead()方法的方式异步调用任何方法。只需声明一个委托，并使用 BeginInvoke()方法和 EndInvoke()方法调用该委托。在定义委托时，这两个方法会自动添加。于是，您就远离了异步基础结构的错综复杂，无须担心后台线程和同步了(然而，也不是完全无须担心)。注意 BeginInvoke()方法和 EndInvoke()方法不在 VS.NET IDE 的 IntelliSense 中，这些方法在编译时添加。

现在接着介绍下一部分代码，在这部分代码中，调用了 MessageAssembler()方法。由于 BeginRead()方法的异步性，无法确切得知服务器何时发送数据和数据量。数据或许立刻就全部发送过来，或许分为一百个小片段，每个片段仅有一两个字符。所以，在接收到某种指示数据末尾的信号之前，必须定期读取数据。在这种情况下，可以在消息的末端添加一个符号(#)，该符号将充当 MessageAssembler()方法的触发器代理，指示输入字符串的结束。由此，MessageAssembler 类就可以停止等待更多的数据，并处理数据了。本例还使用 System.Text 命名空间中的 StringBuilder 类，这个类可以对字符串进行高效的处理。下面详细介绍一下 MessageAssembler()方法：

```
private void MessageAssembler(byte [] Bytes, int offset , int count)
{
  for(int ByteCount = 0; ByteCount < count - 1; ByteCount++)
  {
    if(Bytes[ByteCount] == 35)     // Check for '#' to signal the end
    {
      object[ ] Params = {StrBuilder.ToStrinq()};
      Invoke(new InvokeDisplay(this.DisplayData), Params);
      StrBuilder = new StringBuilder();
    }
    else
    {
          StrBuilder.Append((char)Bytes[ByteCount]) ;
    }
  }
}
```

MessageAssembler()方法在数据的字节数组中进行循环，并使用 StringBuilder 类的实例将数据作为字符串的片段拼接起来，直到遇到#字符为止。一旦其遇到#字符，就表明输入字符串结束了，它就停下来，并调用 StringBuilder 实例的 ToString()方法对字符串进行转换。在此无须将字节手工转换为字符串，因为 StringBuilder 类会完成这项工作。接着，调用 DisplayData()方法来处理数据：

```
object [ ] Params = {StrBuilder.ToString()};
Invoke(new InvokeDisplay(this.DisplayData), Params);
```

这是第二次遇到上述代码，它的作用是什么？这个方法运行在后台工作线程中，并与 UI 窗体处于同一线程之中。可以在应用程序的任何地方调用该方法，但由于这种操作不具有线程安全性，所以最好不要这么做。Windows Form 是基于 Win32 单线程单元( Single Threaded Apartments，STA)的，所以不是线程安全的。这意味着一旦窗体实例化，就无法在操作线程之间(包括由异步操作生成的后台线程)安全地来回转换。必须在窗体所在的线程上调用窗体方法。为减缓这一问题的影响，CLR 对 Invoke()方法提供了支持，该方法能够在线程之间编组调用。

如果怀疑上述主张，可以单步执行代码，打开 Threads 窗口，查看代码的线程 ID 号，该ID号表示正在执行的代码所在的当前线程。创建一个委托，并通过窗体的Invoke()方法调用该委托，就可以在窗体的线程中执行该委托，并可以安全地与 Form 控件进行交互。如果没有编组，最初代码会运行良好，并实现了所期待的功能，但这常常会引起应用程序的不稳定，以后会遇到众多问题，有时还会出现不可预知的行为。应用程序生成的线程越多，情况就越糟。所以，如果没有编组线程，就不要与图形用户界面(GUI)通信。另外，委托的签名也必须始终与 Invoke()方法的签名相匹配，所以，必须创建一个对象数组，并将字符串插入其中，这是使用 Invoke()方法的惟一途径。调用 DisplayData()方法来显示数据：

```
private void DisplayData(string stockInfo)
{
  if(stockInfo == " - 1")
  {
    MessageBox.Show("Symbol not found!", "Invalid Symbol",
      MessageBoxButtons.OK, MessageBoxIcon.Error) ;
  }
  else
  {
    AddStock(stockInfo) ;
  }
}
```

在 DisplayData()方法中，只需检查字符串的值是否为 - 1。假如由于提交了一个无效符号而导致被请求的股票报价无法返回，服务器就会被配置为返回 - 1 字符串。当然，在此例中，无效的符号是没有在股票数据库表 tbl_stocks(稍后将看到)中出现的任何符号。将 stockInfo 变量传递给 AddStock()方法，该方法会把它添加到窗体上的 lstQuotes 控件中。

```
private void AddStock(string stockInfo )
{
  string[ ] StockParameter = stockInfo.Split(new char[ ] {","});
  ListViewItem Item = new ListViewItem(StockParameter) ;

  if(double.Parse(StockParameter[2]) > 0)
  {
    Item.ForeColor = Color.Green;
  }
  else if(double.Parse(StockParameter[2]) < 0)
  {
    Item.ForeColor = Color.Red;
  }
  lstQuotes.Items.Add(Item) ;
}
```

把服务器配置为以字符串的形式返回数据，各个值用逗号隔开：

```
symbol, Price, Change, Bid, Ask, Volume
```

首先，使用字符串类的 Split()方法，将这些值彼此分开。接着，创建 ListViewItem 类的一个新实例，并传入新创建的字符串数组作为其构造函数的参数。最后，将 lstQuotes 控件中的股票报价加上彩色，如果股票的价格下跌，整个报价就显示为红色，如果股票价格上涨，报价则显示为绿色。要实现这一操作，仅需要将包含当前股票价格的字符串数组的第二个值转换为 Double，并检查它的值。设置完颜色后，就可以向 lstQuotes 控件中添加一个新的条目了。

现在已几乎完成了客户端代码；仅需要再创建一些小的方法就彻底完成了。首先，需要向 btnGetQuote 的 click 事件添加如下的代码：

```
private void btnGetQuote_Click(object sender,System.EventArgs e)
{
  Send(txtStock.Text.Trim() + "#");
  txtStock.Text = "";
}
```

该方法获得 txtStock 文本框的字符串值，并向字符串的末尾添加了一个#字符，以指示这个字符串的结束，并将它传递给 Send()方法。在 MessageAssembler()方法中，也需要#字符来确定所接收的字符串何时结束。

一旦将数据传递给 Send()方法，该方法就通过把底层的 TcpClient 流传递给 StreamWriter 类的构造函数，来创建一个新的 StreamWriter 类实例，并调用它的 Write()

方法，以流的形式通过套接字将数据发送出去。我们还调用了 Flush()方法来确保数据立即被发送，并且不在缓冲区中驻留直到将来的某个点。

```
private void Send(string sendData )
{
   StreamWriter writer = new StreamWriter(MyClient.GetStream());
   writer.Write(sendData) ;
   writer.Flush() ;
}
```

到此，我们差不多完成了客户程序的创建。但是还需要做一些清除代码的工作。在很大程度上，Windows Form 类通过调用自己的 Dispose()方法和基类的 Dispose()方法完成大部分的清除工作，但是由于.NET 没有确定的垃圾收集，所以最好手工关闭 TcpClient 连接。可以编写一个小的函数来实现该操作，当用户关闭窗体时，将从 SocketClient_Closing()方法中调用该函数：

```
private void StockClient_Closing(object sender , EventArgs e )
{
     closeConnection() ;
}

private void closeConnection()
{
  if(MyClient != null)
  {
       MyClient.Close() ;
       MyClient = null;
   }
}
```

还需要在窗体的 Main()方法中实例化 StockClient 窗体的副本，以启动应用程序：

```
public static void Main()
{
   Application.Run(new StockClient());
}
```

最后，需要在退出菜单项的 Click 事件上调用 Application.Exit()方法，关闭应用程序：

```
private void mnuExit_Click (object sender,System.EventArgs e)
{
```

```
        Application.Exit();
    }
```

至此，应用程序的客户端部分就完成了。

### 2. 创建服务器程序

现在开始创建服务器应用程序。由于目标环境的多客户性，所以在创建 StockServer 应用程序时需要采取稍微不同的方式。我们需要跟踪客户机，知道它们何时连接和断开。在每个客户机上使用一个类的实例，客户机管理将更加有效。所以，需要创建一个单独的客户类，该类表示连接到服务器的客户机，如下面的 UML 类图(见图 7-5)所示。

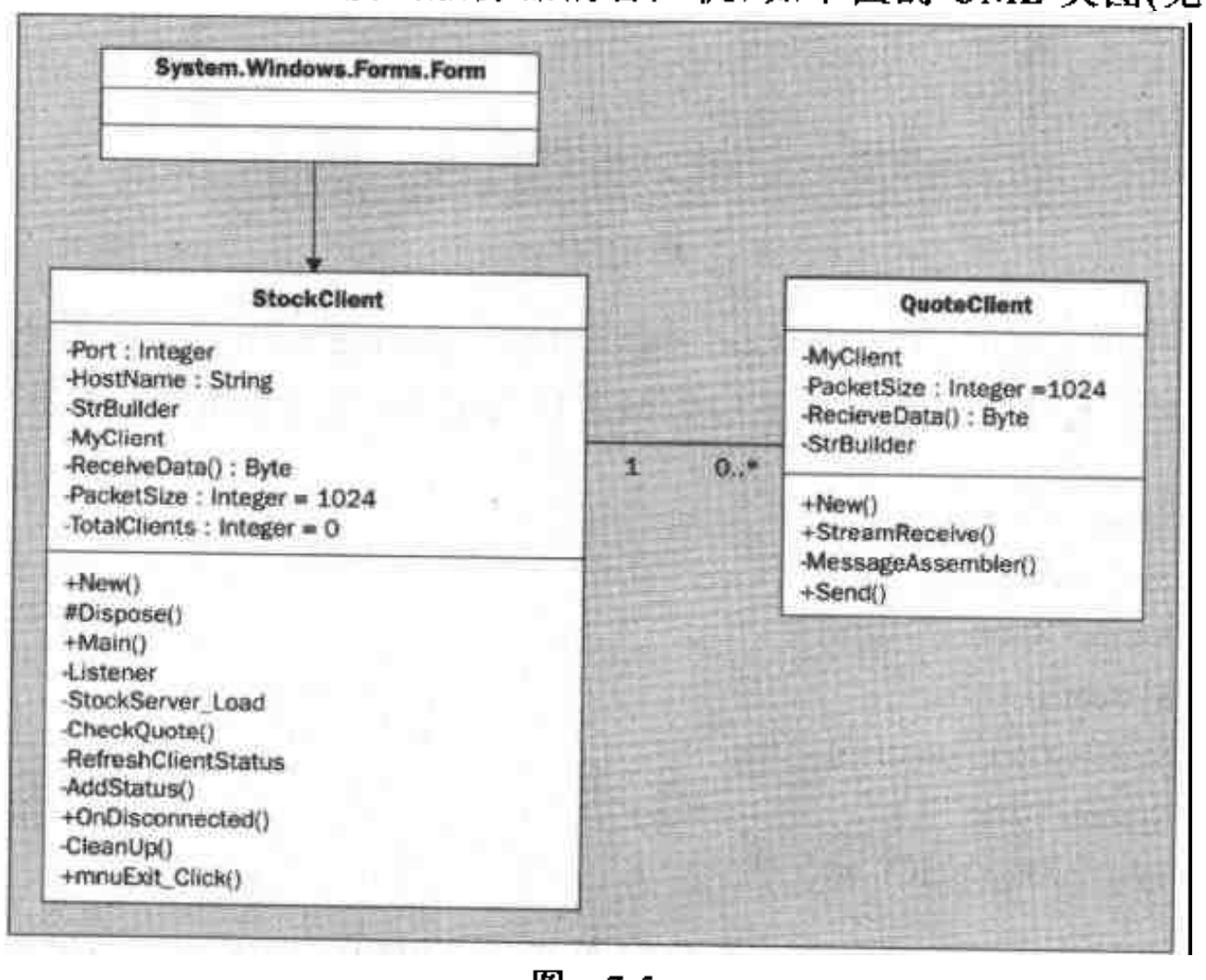


图　7-5

为每个连接到服务器上的新客户创建一个新的 QuoteClient 类实例，所以 StockServer 类和 QuoteClient 类具有一对多的关系。QuoteClient 类总是在一个新生成的线程中被实例化，用于处理刚刚连接上的新客户。QuoteClient 类接受一个 TcpClient 对象作为其构造函数的参数，以处理新客户。稍后将讨论 QuoteClient 类。首先介绍用户界面。就 UI 来说，服务器应用程序比客户端应用程序稍微简单一些。在该用户界面上只有一个显示某些信息的 ListBox 控件和一个具有 Exit 子项的标准 File 菜单。除了那些控件以外，拖放一个新的 StatusBar，并将它的 Anchor 属性修改为 Bottom, Right，将其置入窗体的右下角。将窗体的 Name 和 Text 属性改为 StockServer。该窗体如图 7-6 所示。


图　7-6

还需要一个类文件 QuoteClient.cs。该应用程序将访问 SQL Server 数据库以获取股票报价信息，所以除了需要引用如下的数据命名空间外，还要引用必要的数据命名空间：

```
using System.Threading;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Configuration;
using System.Data;
using System.Data.SqlClient
```

还需要一些 private 变量，这些变量将在整个应用程序中使用。当浏览这个应用程序的代码时，就会看到它们。

```
private Thread ListenerThread ;
private TcpListener MyListener ;
private int Port ;
private TcpClient MyClient ;
private int TotalClients = 0;
private const int PacketSize = 1024;
private byte[ ] ReceiveData = new byte[PacketSize];
```

服务器应用程序一打开就开始运行。因此，应首先向应用程序的 StockServer_Load() 方法中输入某些代码。稍后介绍服务器的 Listener()方法，该方法是服务器应用程序的核心。但首先需要生成一个新的线程，以在后台运行 Listener()方法：

```
private void StockServer_Load(object sender, EventArgs e)
{
   IDictionary HostSettings ;

   try
   {
      HostSettings = ConfigurationSettings.GetConfig("HostInfo");
      Port = int.Parse(HostSettings("port"));
      ListenerThread = new Thread( new ThreadStart(Listener) ) ;
      ListenerThread.Start() ;
      RefreshClientStatus() ;
   }
   catch( Exception ex )
   {
      AddStatus("An error has occurred. The server is not running." +
                  ex.ToString() ) ;
      CleanUp() ;
   }
   finally
   {
      HostSettings = null;
   }
}
```

与客户端应用程序一样，将配置文件中的端口号赋给已定义的 Port 变量。服务器本身就是主机，所以当创建服务器侦听器时，并不需要主机名称。该应用程序是由两个相互独立运行的独立部分组成，所以要确保客户端应用程序和服务器应用程序使用相同的配置文件。假如两个应用程序的端口号不是精确匹配，就不会产生任何功效。假如出现错误，就使用 AddStatus()方法通知用户，并调用 CleanUp()方法做一些手工清理工作。稍后将介绍这两种方法。下面先来看一下 Listener()方法：

```
private void Listener()
{
   try
   {
      MyListener = new TcpListener(Port);
      MyListener.Start() ;

      Object[ ] Message =
      {"Server started. Awaiting new connections..."};
```

```
Invoke(new InvokeStatus(this.AddStatus), Message);

while (true)
{
  QuoteClient NewClient =
      new QuoteClient(MyListener.AcceptTcpClient()) ;
  NewClient.Disconnected +=
      new DisconnectedHandler(onDisconnected) ;
  NewClient.QuoteArrived +=
      new QuoteArrivedHandler(CheckQuote) ;

    Object [ ] ConnectMeseage = {"A new client just connected at"
        + Now().ToShortTimeString()};
    Invoke(new InvokeStatus(this.AddStatus), ConnectMessage);
    TotalClients += 1;
    RefreshClientStatus() ;
  }
}
catch( Exception ex )
{
  object [ ] Message   = {"The server stopped due to an unexpected"
      + "error\r\n" +   ex.ToString()} ;
  Invoke(new InvokeStatus(this.AddStatus), Message);
}
}
```

该方法主要表示服务器的内在引擎，所以它是服务器应用程序的 一个非常重要的组成部分。在初始化端口号时，应调用 TcpListener 类实例的 AcceptTcpClient()方法来接受连接请求。其实 TcpListener 类就是服务器应用程序。它依赖 Socket 类在更高的抽象级上提供 TCP 服务。然而，生成一个新的后台线程来处理 Listener()方法的原因是 AcceptClient()方法，该方法是一个同步方法，在等待连接时，它会使正在运行的线程暂停。所以需要将它作为后台线程来运行。既然该方法运行在后台线程中，就需要使用窗体的 Invoke()方法来编组当前的工作线程和正在运行 UI 控件的线程。同时启动侦听输入数据的异步进程。在本例中，该进程就是来自客户端应用程序的股票报价请求。与客户端应用程序 一样，这里将使用位于 QuoteClient 类中的 StreamReceive()方法：

```
public void StreamReceive (IAsyncResult ar)
{
  int ByteCount;
```

```
try
{
   lock(MyClient.GetStream())
   {
      ByteCount = MyClient.GetStream().EndRead(ar);
   }
   if(ByteCount < 1)
   {
      Disconnected(this) ;
      return;
   }

      MessageAssembler(ReceiveData, 0, ByteCount);
      lock(MyClient.GetStream())
      {
         MyClient.GetStream().BeginRead(ReceiveData, 0, PacketSize,
            new AsyncCallback(StreamReceive), null) ;
      }
   }
   catch( Exception ex )
   {
      Disconnected(this) ;
   }
}
```

该方法和它在客户端应用程序中的方法的主要区别是，用户当前处于一个多线程、多用户环境中，在该环境中，用户无法得到默认的数据流，无法做它想做的事情。在读取流中的数据时，服务器中的另外一个线程或许会尝试将数据发送给同一流，导致资源冲突的机会大增。所以需要使用同步。要实现简单的同步，可以在从流中读取数据时，使用关键字 lock 来锁定请求的流。lock 是最基本的线程同步工具。过度使用 lock 可能会损害应用程序的性能，所以在锁定资源时，务必作出明确的判断。对于较复杂的和定制的线程同步来说，可以使用 System.Threading 命名空间中的其他类，比如 Interlocked，该类允许增加和减少互联锁。除了这点不同以外，服务器应用程序中的 ReceiveStream()方法与客户端应用程序中的 ReceiveStream()方法几乎相同。

服务器应用程序中的 MessageAssembler()方法也极其类似于客户端应用程序中的 MessageAssembler()方法。惟一的不同在于，服务器的 MessageAssembler()方法调用 CheckQuote()方法连接数据库，并通过触发 QuoteArrived 事件检索股票报价。而在前面

的论述中，这是在 Listener()方法中处理的。

```
private void MessageAssembler(byte[ ] Bytes, int offset, int count)
{
   for(int ByteCount = 0; ByteCount < count -1; ByteCount++)
   {
      if(Bytes[ByteCount] == 35) // Check for '#' to signal the end
      {
            QuoteArrived(this, StrBuilder.ToString()) ;
            StrBuilder = new StringBuilder() ;
      }
      else
      {
            StrBuilder.Append((char)Bytes(ByteCount)) ;
      }
   }
}
```

在论述 CheckQuote()方法之前，先简要地介绍一下数据源，服务器是从数据源中检索报价信息的。

首先创建一个名为 StockDB 的 SQL Server 数据库，该数据库包含一个 tbl_stocks 表，其结构如表 7-4 所示。

注意：

数据库的设置、填充脚本连同本书所有的代码，都可以在 Wrox Press Web 站点 http://www.wrox.com 中得到。

表 7-4

| 字段 | 说　明 |
|---|---|
| Symbol | 实际的股票符号 |
| Price | 股票的最后价格 |
| Change | 股票价格的变化 |
| Bid | 股票的最后出价 |
| Ask | 股票的最后标价 |
| Volume | 一次贸易会话中的股票交易总量 |

这就是我们所需要的数据库，现在返回到代码和 CheckQuote()方法上来。CheckQuote()方法驻留在应用程序的主窗体内，当触发 QuoteArrive()方法时，本地事件

处理程序就会调用该方法。该方法可以连接数据库，进行查询以检索报价信息，并将数据发送给用户。可以使用 Visual Studio .NET 中的 SqlConnection 控件，利用向导生成数据库连接字符串，也可以实例化 System.Data.SqlClient 命名空间中的 SqlConnection 类，并手工赋给其一个连接字符串，如下所示：

```
private void CheckQuote(QuoteClient sender,
    string stockSymbol)
{

    // Connection string using SQL Server authentication
    SqlConnection SqlConn =
      new SqlConnection("Initial Catalog=StockDB;" +
      "Data Sources(local);User ID=sa;Passwords");
// Alternative Connection string using Windows Integrated security
// SqlConnection SqlConn =
//     newSqlConnection(" Initial Catalog=StockDB;" +
//             "Data Source=(local);Integrated Security=SSPI" ) ;

string SqlStr =
     "SELECT symbol, price, change, bid, ask, volume " +
     "FROM tbl_stocks WHERE symbols=' " + stockSymbol + " ' " ;

SqlCommand SqlCmd = new SqlCommand(SqlStr, SqlConn) ;

try
{
  SqlCmd.Connection.Open() ;
int Records = 0 ;
StringBuilder TempString = new StringBuilder();

SqlDataReader sqlDataRd = SqlCmd.ExecuteReader();

while(sqlDataRd.Read())
{
  for(int FieldCount = 0; FieldCount < = 5; FieldCount++)
  {
    TempString.Append(
        sqlDataRd.GetValue(FieldCount).ToString() + ",");
    Records += 1;
```

```
    }
}
if(Records== 0)
{
    sender.send("-1#") ;
}
else
{
   TempString.Replace(",", "#", TempString.Length - 1, 1) ;
   sender.send(TempString.ToString()) ;
}
}
catch(SqlException sqlEx)
{
    object[ ] Message = {sqlEx.ToString()};
    Invoke(new InvokeStatus(this.AddStatus), Message);
}
catch(Exception ex )
{
   object[ ] Message =
      {"Unable to retrieve quote information from the Database."};
   Invoke(new InvokeStatus(this.AddStatus), Message);
}
finally
{
   // Close the Connection and the Data Reader
   if(SqlConn.State != ConnectionState.Closed)
      SqlConn.Close() ;
      sqlDataRd.Close() ;

    }
  }
```

还需要一个SQL查询来为客户端应用程序所请求的股票返回表中的所有6个字段：

```
string SqlStr =
          "SELECT symbol, price, change, bid, ask, volume " +
         "FROM tbl_stocks WHERE symbol=' " + stockSymbol + " ' ";
```

有了必要的SQL字符串和连接，就可以实例化SqlCommand和SqlDataReader对象，

从数据库服务器中读取数据了。

最后，创建一个新的 SqlDataReader 类实例，并将其设置为 SqlCommand 对象的 ExecuteReader()方法的结果，之后就可以执行查询了。接着，在返回数据的每个列中进行迭代，并将值添加到 StringBuilder 对象中，每个值用逗号隔开。如果 ExecuteReader()方法没有返回数据，就必须将值为 -1 的字符串发回用户，以通知他们请求的数据不存在。另外，用#字符(表示字符串的终结)代替字符串中的最后一个逗号，并使用 Send()方法将它发回客户。最后，操作执行完后，就必须确保数据库连接关闭。Finally 子句中的代码将检查数据库连接是否依然打开。如果是，就关闭它。

服务器应用程序的 Send()方法驻留在 QuoteClient 类中，该方法的代码与客户端应用程序中的稍有不同。主要的不同是消息将异步地发回给客户。

```
public void send(string sendData)
{
   byte[ ] Buffer =
      System.Text.ASCIIEncoding.ASCII.GetBytes(sendData) ;
   lock(MyClient.GetStream())
   {
      MyClient.GetStream().BeginWrite(Buffer, 0, Buffer.Length,
         null, null) ;
   }
}
```

从接口方面来讲，BeginWrite()方法与 BeginRead()方法极为类似。首先，需要将字符串消息转换为字节数组，使用 System.Text 命名空间中的 ASCII 类，很容易完成这一操作。然后再次锁定流，以防止其他的线程同时把数据写入该流。以上就是将数据异步写入客户端应用程序所需要的全部操作。

## 7.2.3　运行应用程序

在 Visual Studio .NET 中构建每个项目，不要忘记把先前在应用程序可执行文件所在的目录中创建的配置文件包括进来。

现在开始运行经过编译的应用程序。首先需要运行 StockServer.exe，它将启动对客户的侦听，如图 7-7 所示。

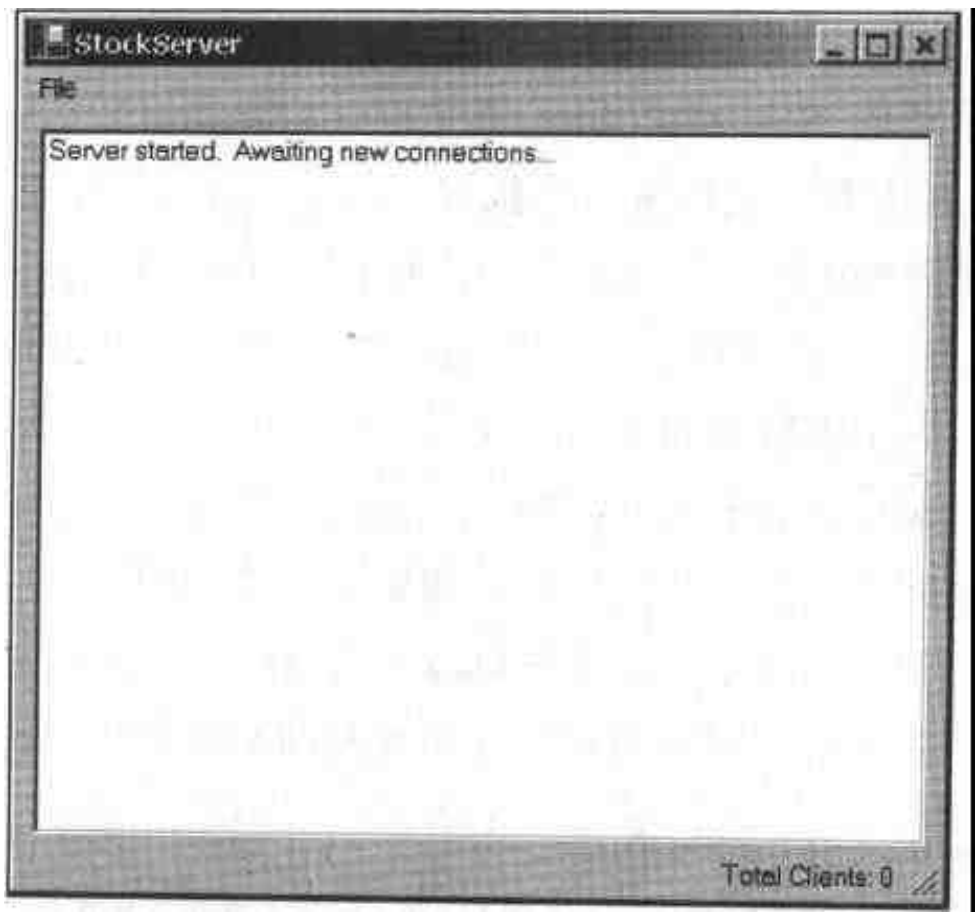


图　7-7

现在运行客户端应用程序的一个实例。窗体上的所有 UI 控件都被禁用，除非用户成功地连接上服务器。所以，单击菜单的 Connect 项，如图 7-8 所示。

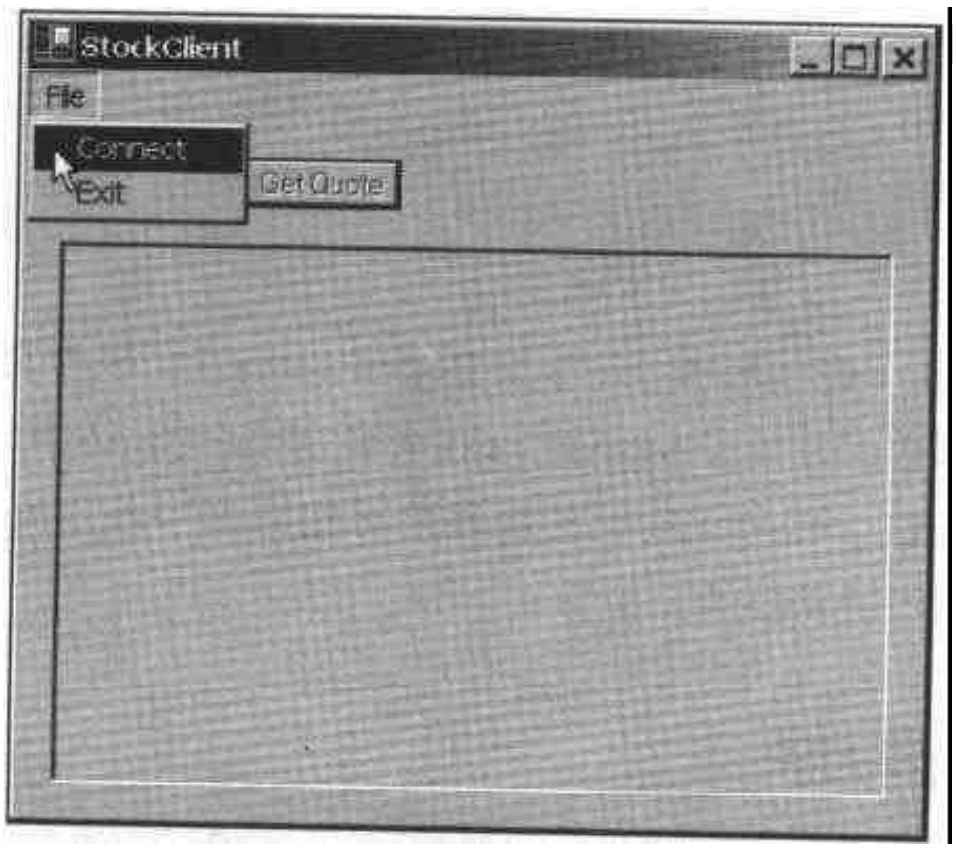


图　7-8

现在，所有控件(除了菜单的 Connect 选项以外)都有效了，并且已使用正确的列实例化了 ListView 控件。从 tbl_stocks 表中输入有效的股票符号。以 CSCO 股票为例，如图 7-9 所示。

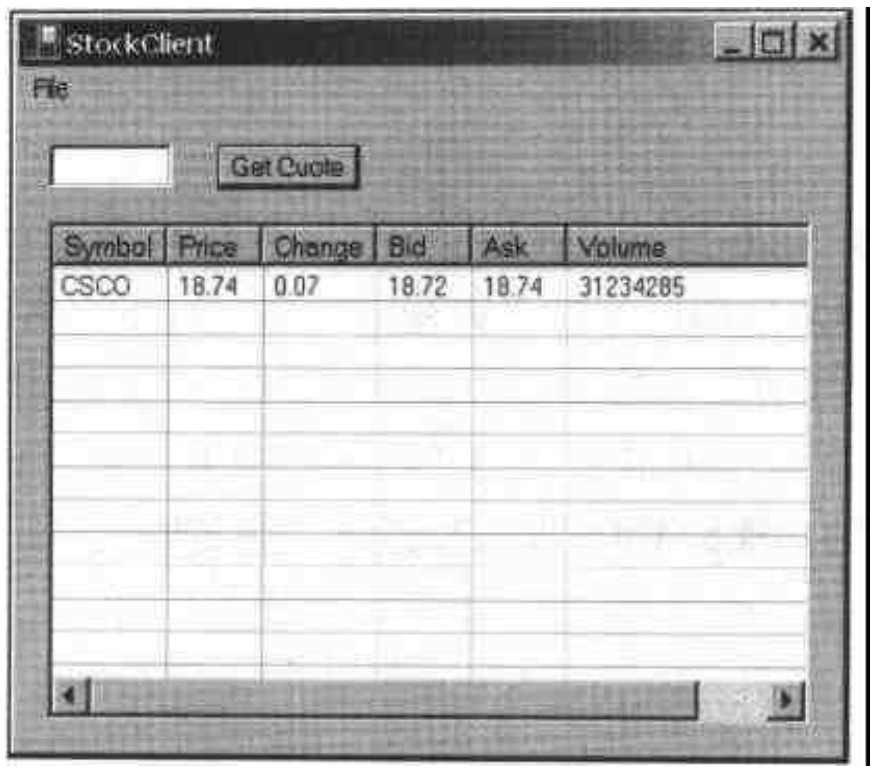


图　7-9

股票服务器成功返回了报价，由于股票价格上涨，所以整行都显示为绿色。再创建 StockClient 类的其他一些实例，看看它们是否正常运行，如图 7-10 所示。

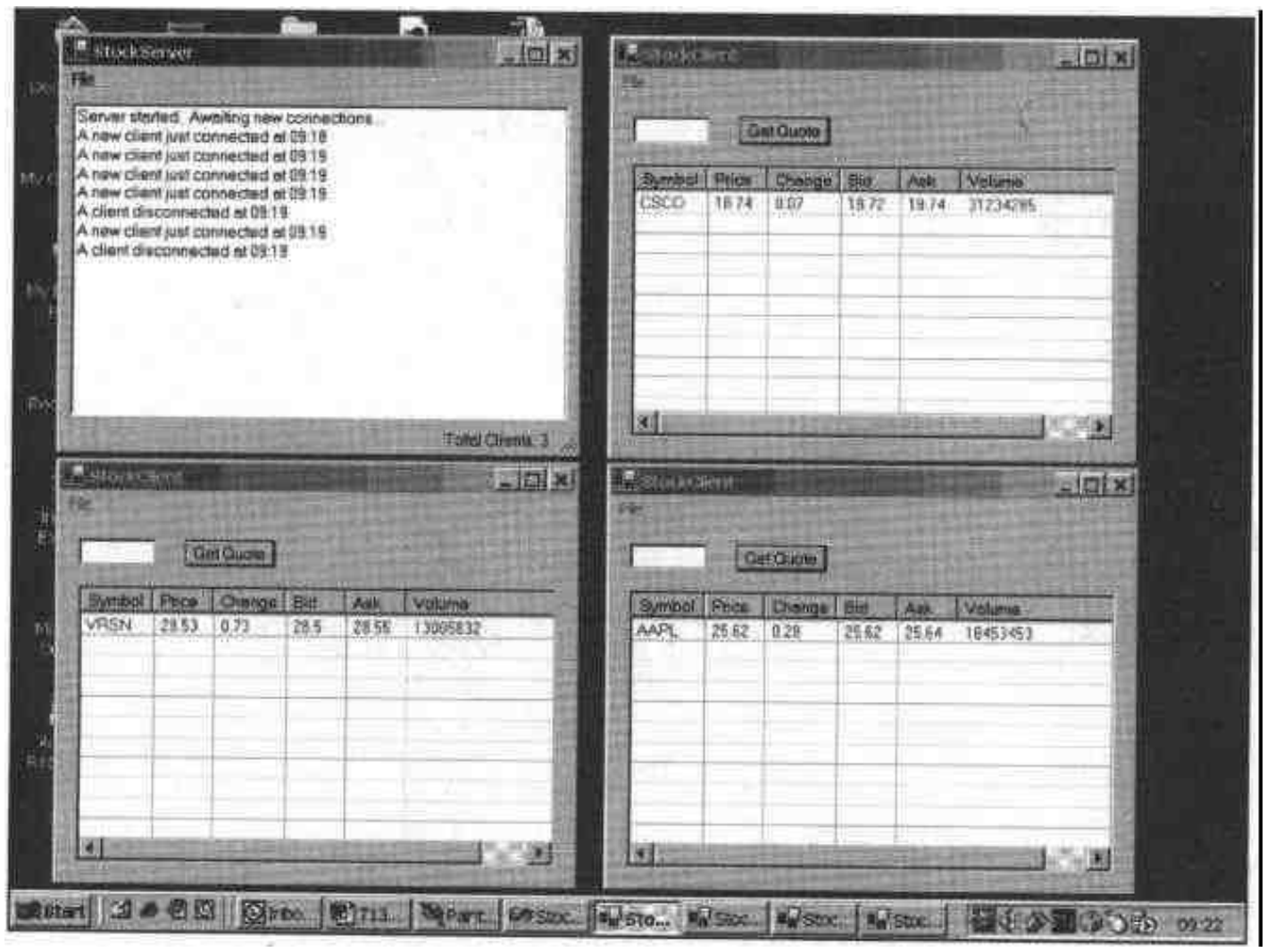


图　7-10

StockClient 应用程序和 StockServer 应用程序相互间运行良好。服务器应用程序跟踪连接与断开连接的客户数，并显示在 ListBox 中。另外，除了以异步方式发送和返回数据以外，多线程服务器应用程序还可以处理众多连接。用户可以步进客户端应用程序和服务器应用程序的代码，更好地了解应用程序的工作流程。

## 7.3　本章小结

使用 Visual Studio .NET 开发多线程网络应用程序是很简单的。它使用一套包罗万象、面向对象的类代替复杂的信息管道和基础结构。对于网络套接字上的更大控制，System.Net.Socket 提供了许多内容丰富的功能。使用.NET 所固有的异步操作支持非常简单，无须编号更多代码，就可以在后台工作线程中运行异步操作。

.NET 的诸多优秀的功能给 C#开发人员提供了更多的支持—— 线程就是其中之一。

# 附录A 支持、勘误表与代码下载

我们一重视您的意见，并想知道您对本书的看法，包括您喜欢和不喜欢的内容，以及您希望我们在下一次完善的地方。您可以通过发送电子邮件(地址为feedback@wrox.com)向我们反馈意见，请确保在反馈信息中提到本书的书名。

## 如何下载本书的示例代码

当您访问 Wrox 公司站点(地址为http://www.wrox.com/)时，通过 Search 工具或书名列表，可以方便地定位到需要的书目。然后，单击 Code 列中的 Download 超链接，或者单击本书的详细页面中的 Download Code 超链接，就可以下载相应的示例代码。

当您下载本书中的代码时，将会看到带有以下 3 个选项的 Web 页面：

- 如果您是 Wrox Developer Community 的成员(即如果您已经在 ASPToday、C#Today 或者 Wroxbase 上注册)，就可以使用一的用户名和密码进行登录以下载代码。
- 如果不是它们的成员，则会向您询问是否愿意注册为会员，以便可以免费下载代码。此外，也可以从 Wrox Press 下载免费的文章。注册为会员后，可以得到本书升级版本和新版本的有关信息。
- 第三个选项是完全绕过注册过程，直接下载代码。

对于本书而言，不注册也能下载代码，但是，如果您愿意注册后下载代码，您的注册信息不会泄漏给第三方。关于这方面的详细条款和条件，可以通过单击下载页面上的相关链接来查看。

从我们的站点上下载的文件都是使用 WinZip 压缩过的文档。保存文件到本地磁盘上的文件夹中后，需要使用一个解压缩程序(例如 WinZip 或 PKUnzip)来解压缩文件。在解压缩文件时，通常将代码解压缩到每一章所在的文件夹中。在解压缩的过程中，应确保解压缩程序(WinZip、PKUnzip，其他)被设置为使用原有的文件夹名。

## 勘误表

我们已经尽量大努力确保本书中的文本和代码没有错误，但是错误仍然在所难免。

如果您发现本书存在错误，例如拼写错误或不正确的代码段，请反馈信息给我们，我们将不胜感激。勘误表的发送可以节约其他读者学习本书的时间，而且能够帮助我们提供更高质量的信息。您的反馈信息将被检查，如果正确，将被粘贴到本书的勘误页面上，或者在本书的后续版本中使用。

要在我们的站点上找到勘误表，请访问http://www.wrox.com/，并通过 Advanced Search 或者书名列表轻松定位到本书页面。然后，单击 Errata 超链接即可，该链接位于本书的详细页面中的封面图解下面。

## 技术支持

如果您希望直接向详细了解本书的专家咨询本书中问题，可以发送电子邮件到support@wrox.com。一个典型的电子邮件应包括下面的内容：

- 在主题栏中必须有本书的书名、ISBN 的后 4 位数字和问题所在的页码。
- 邮件正文中应包括读者的名字、联系信息和问题。

我们将不给您发送垃圾邮件。我们需要您的详细资料，以便节约双方的时间。当您发送电子邮件信息时，要经过下面一系列支持：

- 用户支持：首先，您的信息将被递送到我们的用户支持人员手中，并由他们阅读。对于一些被频繁提到的问题将被归档，并将立即回答有关本书或者 Web 站点的任何常见问题。
- 编辑支持：接着，一些有深度的问题将被送到对本书负责的技术编辑手中，他们在程序设计语言或者特定的产品上有着丰富的经验，能够回答相关主题的详细技术问题。
- 作者支持：最后，如果编辑不能回答您的问题(这种情况很少发生)，他们将请求本书的作者。我们将尽量保护作者免受干扰，以便不影响其写作。然而，我们也非常高兴转寄给他们一些特殊的问题。所有 Wrox 公司的作者都为他们的书提供技术支持。作为回应，他们将发送电子邮件给用户和编辑，进而使所有的读者受益。

**注意：**

Wrox 公司的支持过程仅仅对那些与我们出版的书目内容直接相关的问题提供支持，对于超出常规书目支持的问题，您可以从http://p2p.wrox.com/论坛的公共列表中获得支持信息。

# p2p.wrox.com 站点

为了与作者和其他程序员讨论，请加入 P2P 邮件列表。除了一对一的电子邮件支持系统之外，我们独特的系统还通过邮件列表、论坛和新闻组提供由程序员到程序员的支持。Wrox 的作者、编辑和其他业界的专家也包括在我们的邮件列表中。

无论您是在阅读本书，还是在开发自己的应用程序，都可以在 p2p.wrox.com 站点中找到许多对自己有帮助的邮件列表。和本书特别相关的是站点的.NET 类别中的 aspx 和 aspx_professional 列表。

按照下面的步骤可以订阅一个邮件列表：

(1) 登录http://p2p.wrox.com/站点。

(2) 从左边的主菜单栏中选择一个适当的类别。

(3) 单击希望加入的邮件列表。

(4) 按照说明订阅并填写自己的邮件地址和口令

(5) 回复您收到的确认邮件。

(6) 使用订阅管理程序加入更多的邮件表并设置自己的电子邮件参数。

[General Information]
书名=C#线程参考手册
作者=
页数=247
SS号=11180056
出版日期=